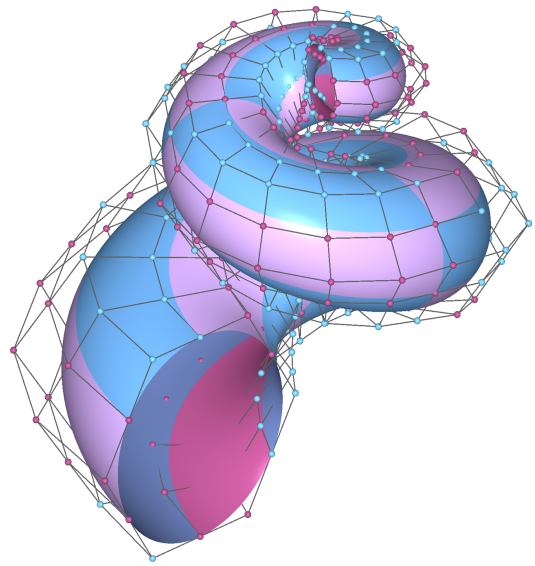
User manual

An OpenGL- and C++-based function library for curve and surface modeling in a large class of extended Chebyshev spaces

© Ágoston Róth, 2018

Babeş–Bolyai University/**Babeş–Bolyai Tudományegyetem**
Department of Mathematics and Computer Science of the Hungarian Line/
Magyar Matematika és Informatika Intézet
ℝ𝕆–400084 **Cluj-Napoca**/**Kolozsvár**, **Romania**/**România**
agoston.roth@gmail.com, agoston.roth@math.ubbcluj.ro

Tutorial | **Updates:** https://sites.google.com/site/agostonroth/algorithm-992



Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers, to redistribute to lists or to use in commercial/non-commercial applications requires prior specific permission and/or a fee.

Contents

1 Theoretical introduction **1**
- 1.1 Preliminaries . . . 1
- 1.2 Proposed algorithms . . . 4
  - 1.2.1 Construction and differentiation of normalized B-basis functions in a large class of EC spaces . . . 5
  - 1.2.2 General dimension and order elevation . . . 8
  - 1.2.3 General B-algorithm . . . 9
  - 1.2.4 General basis transformation . . . 11

2 Full implementation details **13**
- 2.1 Exceptions . . . 14
- 2.2 Smart pointers . . . 16
  - 2.2.1 Type selectors . . . 16
  - 2.2.2 Implicit conversion policies . . . 18
  - 2.2.3 Static checks . . . 18
  - 2.2.4 Ownership policies . . . 19
  - 2.2.5 Storage policies . . . 23
  - 2.2.6 Checking policies . . . 27
  - 2.2.7 Smart pointers . . . 29
  - 2.2.8 Frequently used specialized smart pointers . . . 33
- 2.3 Cartesian coordinates . . . 34
- 2.4 Homogeneous coordinates . . . 39
- 2.5 Texture coordinates . . . 44
- 2.6 Colors . . . 47
- 2.7 Lights . . . 51
- 2.8 Materials . . . 58
- 2.9 Different template and specialized matrices . . . 65
  - 2.9.1 Generic template matrices . . . 65
  - 2.9.2 Real matrices and decompositions . . . 75
  - 2.9.3 Generic and special OpenGL transformation matrices . . . 108
- 2.10 Constants and utility functions . . . 126
- 2.11 Shader programs . . . 129
- 2.12 Generic curves . . . 157
- 2.13 Simple triangle meshes . . . 169
- 2.14 Abstract linear combinations and tensor product surfaces . . . 185
- 2.15 Characteristic polynomials . . . 211
- 2.16 EC spaces . . . 216
- 2.17 B-curves . . . 241
- 2.18 B-surfaces . . . 250

© Ágoston Róth, 2018

3 Usage examples **269**
3.1 Defining and using specialized EC spaces . . . 269
3.2 Using our shader programs . . . 278
3.2.1 Simple (flat) color vertex and fragment shaders . . . 278
3.2.2 Two-sided per pixel lighting . . . 280
3.2.3 Reflection lines combined with two-sided per pixel lighting . . . 283
3.3 Differentiating and rendering basis functions . . . 290
3.4 Defining and using specialized B-curves . . . 294
3.4.1 B-curve generation, order elevation and subdivision . . . 294
3.4.2 B-representation of ordinary integral curves . . . 301
3.5 Defining and using specialized B-surfaces . . . 306
3.5.1 B-surface generation, order elevation and subdivision . . . 306
3.5.2 B-representation of ordinary integral surfaces . . . 318

© Ágoston Róth, 2018

List of Figures

2.1 Tree-view of the header files of the proposed function library . . . 15
2.2 Possible implicit conversion policies ensured by our smart pointers . . . 18
2.3 Possible ownership policies ensured by our smart pointers . . . 20
2.4 Possible storage policies ensured by our smart pointers . . . 23
2.5 Possible checking policies ensured by our smart pointers . . . 27
2.6 Class diagram of 3-dimensional Cartesian coordinates . . . 34
2.7 Class diagram of 3-dimensional homogeneous coordinates . . . 39
2.8 Class diagram of 1-, 2- and 3-dimensional or projective texture coordinates . . . 45
2.9 Class diagram of colors with red, green, blue and alpha channels . . . 47
2.10 Class diagram of directional lights . . . 52
2.11 Class diagram of point lights . . . 53
2.12 Class diagram of spotlights . . . 53
2.13 Class diagram of materials . . . 59
2.14 Class diagrams of different types of template matrices . . . 66
2.15 Diagrams of real matrices and of some auxiliary real row and column matrix related arithmetical binary operators . . . 76
2.16 Diagrams of used real matrix decompositions . . . 88
2.17 Diagram of generic OpenGL transformations . . . 109
2.18 Diagrams of special OpenGL transformations . . . 119
2.19 Class diagram of generic curves . . . 158
2.20 Class diagram of triangular faces . . . 169
2.21 Class diagram of simple triangle meshes . . . 171
2.22 Class diagram of abstract linear combinations . . . 186
2.23 Class diagram of abstract tensor product surfaces . . . 194
2.24 Class diagrams of characteristic polynomials and of their (higher order) zeros . . . 211
2.25 Class diagram of constant-comprising translation invariant EC spaces that can be identified with the solution spaces of constant-coefficient homogeneous linear differential equations defined on intervals for which the corresponding spaces of derivatives are also EC . . . 217
2.26 Class diagram of general B-curves . . . 242
2.27 Class diagrams of general B-surfaces and of ordinary surface coefficients . . . 251

3.1 From top to bottom: evaluation and rendering of the unique normalized B-bases of the EC spaces $\mathbb{P}_{10}^{0,1}$, $\mathbb{T}_{10}^{0,\frac{\pi}{2}}$, $\mathbb{H}_{14}^{0,\pi}$, $\mathbb{AT}_{24}^{0,2\pi}$, $\mathbb{AET}_{27}^{-5\pi/4,5\pi/4}$ and $\mathbb{M}_{14,1,0.2}^{0,\beta}$ that are of types (3.1/269), (3.2/269), (3.3/269), (3.4/269), (3.5/269) and (3.6/269), respectively, where $\beta = \frac{1}{2} \cdot 16.694941067922716 = 8.347470533961358$. (The figure was generated by the source codes listed in Listings 3.10/291 and 3.11/291.) . . . 295
3.2 From left to right: the columns illustrate the order elevation and subdivision of different B-curves in EC spaces $\mathbb{P}_6^{0,1}$, $\mathbb{T}_6^{0,\frac{\pi}{2}}$, $\mathbb{H}_6^{0,\pi}$, $\mathbb{AT}_8^{0,\frac{4\pi}{3}}$ and $\mathbb{M}_{6,1,0.2}^{0,16.694941067922716}$ that are of types (3.1/269), (3.2/269), (3.3/269), (3.4/269) and (3.6/269), respectively. (The figure was generated by the source codes listed in Listings 3.12/294 and 3.13/296.) 302

© Ágoston Róth, 2018

3.3 Control point based exact description (or B-representation) of different arcs of a logarithmic spiral. (The figure was generated by the source codes given in Listings 3.14/302 and 3.15/303.) . . . 306

3.4 (a) Randomly generated initial B-surface, rendered with a simple color material together with its control net. (b) Order elevation of the initial B-surface, rendered with a simple color material together with its initial and order elevated control nets. (c) The order elevated B-surface, rendered with a simple color material together with its control net. (d) The order elevated B-surface, rendered with a simple color material together with its control net and reflection lines. (d) The order elevated B-surface, rendered with the logarithmic umbilic deviation color scheme together with its control net. (e) The order elevated B-surface, rendered with the translated logarithmic scale of the umbilic deviation color scheme together with its control net and reflection lines. (The figure was generated by the source codes given in Listings 3.16/307 and 3.17/308.) . . . 319

3.5 The figure was generated by the source codes given in Listings 3.16/307 and 3.17/308. (a)–(ℓ) The color schemes correspond to the point-wise variations of the x-, y- and z-coordinates, of the length of the normal vectors, of the Gaussian- and mean curvatures, of the Willmore energy and its translated logarithmic counterpart, of the umbilic deviation and its translated logarithmic scale, of the total curvature and its translated logarithmic variant, respectively. (In each case the applied color map behaves like a temperature variation that ranges from the cold dark blue to the hot red, by passing through the colors cyan, green, yellow and orange such that the minimal and maximal values of a fixed energy type correspond to the extremal colors dark blue and red, respectively. For more details, see the lines 10/127–50/128 and 31/195–54/195 of Listings 2.34/127 and 2.44/194, respectively.) . . . 320

3.6 (a) A randomly generated initial B-surface, rendered with a simple color material together with its control net. (b) Subdivision of the given B-surface in direction u. The obtained B-patches are rendered with simple color materials together with the control net of the initial B-surface and with their control nets. (c) The obtained B-patches are rendered with simple color materials together with their control nets. (d) The obtained B-patches are rendered with simple color materials together with their control nets and smooth reflection lines. (The figure was generated by the source codes given in Listings 3.16/307 and 3.17/308.) . . . 321

3.7 (a) A randomly generated initial B-surface, rendered with a simple color material together with its control net. (b) Subdivision of the given B-surface in direction v. The obtained B-patches are rendered with simple color materials together with the control net of the initial B-surface and with their control nets. (c) The obtained B-patches are rendered with simple color materials together with their control nets. (d) The obtained B-patches are rendered with simple color materials together with their control nets and smooth reflection lines. (The figure was generated by the source codes given in Listings 3.16/307 and 3.17/308.) . . . 322

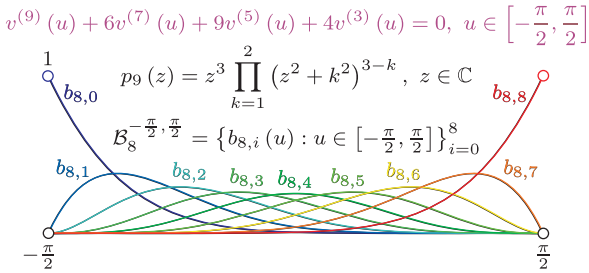
3.8 A possible control point based exact description (or B-representation) of the ordinary exponential-trigonometric integral surface (3.9/320) by means of a 5×6 matrix of B-surface patches. (a) Using alternating simple color materials, the obtained B-surface patches are rendered together with their control nets. (b) Using alternating simple color materials, the obtained B-surface patches are rendered together with their control nets and smooth reflection lines. (The figure was generated by the source codes given in Listings 3.18/321 and 3.19/324.) . . . 334

© Ágoston Róth, 2018

3.9 Isoparametric lines and their tangent vectors in case of a possible B-representation of the ordinary exponential-trigonometric integral surface (3.9/320). Along all $5 \times 6 = 30$ B-surface patches we have generated 5 and 3 isoparametric lines in directions u and v, respectively. The u- and v-isoparametric lines consist of 20 and 10 subdivision points, respectively. (For better visibility, we have rendered the tangent vectors only of some of the isoparametric lines.) (a) Surface patches, rendered with alternating simple color materials together with their control nets and isoparametric lines. (b) u-directional isoparametric lines and their tangent vectors. (c) v-directional isoparametric lines and their tangent vectors. (The figure was generated by the source codes given in Listings 3.18/321 and 3.19/324.) 335

© Ágoston Róth, 2018

© Ágoston Róth, 2018



Extended Chebyshev spaces • Critical length for design • Constant-coefficient homogeneous linear differential equations • Ordinary bases • Normalized B-bases • B-curves • B-surfaces • Construction and differentiation of normalized B-bases • General order (or dimension) elevation • General B-algorithm • General basis transformation • Control point based exact description of ordinary integral curves and surface

1 Theoretical introduction

We propose a platform-independent multi-threaded function library that provides data structures to generate, differentiate and render both the ordinary basis and the normalized B-basis of a user-specified extended Chebyshev (EC) space that comprises the constants and can be identified with the solution space of a constant-coefficient homogeneous linear differential equation defined on a sufficiently small interval. Using the obtained normalized B-bases, our library can also generate, (partially) differentiate, modify and visualize a large family of so-called B-curves and tensor product B-surfaces. Moreover, the library also implements methods that can be used to perform dimension elevation, to subdivide B-curves and B-surfaces by means of de Casteljau-like B-algorithms, and to generate basis transformations for the B-representation of arbitrary integral curves and surfaces that are described in traditional parametric form by means of the ordinary bases of the underlying EC spaces. Independently of the algebraic, exponential, trigonometric or mixed type of the applied EC space, the proposed library is numerically stable and efficient up to a reasonable dimension number and may be useful for academics and engineers in the fields of Approximation Theory, Computer Aided Geometric Design, Computer Graphics, Isogeometric and Numerical Analysis.

1.1 Preliminaries

In order to explain the possibilities of the proposed function library and to provide detailed implementation comments, we will use the following well-known notions. Let $n \geq 1$ be a fixed integer and consider the extended Chebyshev (EC) system

$$\mathcal{F}_n^{\alpha,\beta} = \{\varphi_{n,i}(u) : u \in [\alpha,\beta]\}_{i=0}^{n},\ \varphi_{n,0} \equiv 1,\ -\infty < \alpha < \beta < \infty \tag{1.1}$$

of basis functions in $C^n([\alpha,\beta])$, i.e., by definition [Karlin and Studden, 1966], for any integer $0 \leq r \leq n$, any strictly increasing sequence of knot values $\alpha \leq u_0 < u_1 < \ldots < u_r \leq \beta$, any positive integers (or multiplicities) $\{m_k\}_{k=0}^{r}$ such that $\sum_{k=0}^{r} m_k = n+1$, and any real numbers $\{\xi_{k,\ell}\}_{k=0,\ \ell=0}^{r,\ m_k-1}$ there always exists a unique function

$$f := \sum_{i=0}^{n} \lambda_{n,i}\varphi_{n,i} \in \mathbb{S}_n^{\alpha,\beta} := \left\langle \mathcal{F}_n^{\alpha,\beta} \right\rangle := \operatorname{span} \mathcal{F}_n^{\alpha,\beta},\ \lambda_{n,i} \in \mathbb{R},\ i = 0,1,\ldots,n \tag{1.2}$$

© Ágoston Róth, 2018

that satisfies the conditions of the Hermite interpolation problem

$$f^{(\ell)}(u_k) = \xi_{k,\ell},\ \ell = 0, 1, \ldots, m_k - 1,\ k = 0, 1, \ldots, r. \tag{1.3}$$

In what follows, we assume that the sign-regular determinant of the coefficient matrix of the linear system (1.3/2) of equations is strictly positive for any permissible parameter settings introduced above. Under these circumstances, the vector space $\mathbb{S}_n^{\alpha,\beta}$ of functions is called an EC space of dimension $n+1$. In terms of zeros, this definition means that any non-zero element of $\mathbb{S}_n^{\alpha,\beta}$ vanishes at most n times in the interval $[\alpha, \beta]$. Such spaces and their corresponding spline counterparts have been widely studied, consider, e.g., articles [Carnicer et al., 2004, 2007; Costantini et al., 2005; Lü et al., 2002; Lyche, 1985; Mainar and Peña, 1999, 2004, 2010; Mainar et al., 2001; Mazure and Laurent, 1998; Pottmann, 1993; Róth, 2015a,b; Schumaker, 2007] and many other references therein. (Concerning the definition of EC spaces, the condition $1 \equiv \varphi_{n,0} \in \mathbb{S}_n^{\alpha,\beta}$ would be not necessary, nevertheless we have included the constants in $\mathbb{S}_n^{\alpha,\beta}$ in order to ensure that all its bases can be normalized.)

Hereafter we will also refer to $\mathcal{F}_n^{\alpha,\beta}$ as the ordinary basis of $\mathbb{S}_n^{\alpha,\beta}$ and we will also assume that the (n-dimensional) space $D\mathbb{S}_n^{\alpha,\beta} := \left\{ f^{(1)} : f \in \mathbb{S}_n^{\alpha,\beta} \right\}$ of the derivatives is also EC over the interval $[\alpha, \beta]$. Using [Carnicer and Peña, 1995, Theorem 5.1] and [Carnicer et al., 2004, Theorem 4.1], it follows that under these conditions the vector space $\mathbb{S}_n^{\alpha,\beta}$ also has a unique strictly totally positive normalizable basis, called normalized B-basis

$$\mathcal{B}_n^{\alpha,\beta} = \{b_{n,i}(u) : u \in [\alpha, \beta]\}_{i=0}^{n} \tag{1.4}$$

that apart from the identity

$$\sum_{i=0}^{n} b_{n,i}(u) \equiv 1,\ \forall u \in [\alpha, \beta] \tag{1.5}$$

also fulfills the properties

$$b_{n,0}(\alpha) = b_{n,n}(\beta) = 1, \tag{1.6}$$

$$b_{n,i}^{(j)}(\alpha) = 0,\ j = 0, \ldots, i-1,\ b_{n,i}^{(i)}(\alpha) > 0, \tag{1.7}$$

$$b_{n,i}^{(j)}(\beta) = 0,\ j = 0, 1, \ldots, n-1-i,\ (-1)^{n-i} b_{n,i}^{(n-i)}(\beta) > 0 \tag{1.8}$$

conform [Carnicer and Peña, 1995, Theorem 5.1] and [Mazure, 1999, Equation (3.6)]. (In order to avoid ambiguity, in case of some figures we will also use the notation $\overline{\mathcal{F}}_n^{\alpha,\beta}$ instead of $\mathcal{B}_n^{\alpha,\beta}$.)

All algorithms that will be presented in the forthcoming sections are valid in case of any EC space $\mathbb{S}_n^{\alpha,\beta}$ that fulfills all conditions above, however in case of their C++ and OpenGL based implementation we always assume that $\mathbb{S}_n^{\alpha,\beta}$ can be identified with the solution space of the constant-coefficient homogeneous linear differential equation

$$\sum_{i=0}^{n+1} \gamma_i v^{(i)}(u) = 0,\ \gamma_i \in \mathbb{R},\ u \in [\alpha, \beta] \tag{1.9}$$

of order $n+1$. Such a solution space:

- is translation invariant and it is spanned by those ordinary basis functions that are generated by the (higher order) zeros of the characteristic polynomial

$$p_{n+1}(z) = \sum_{i=0}^{n+1} \gamma_i z^i,\ z \in \mathbb{C} \tag{1.10}$$

  associated with the differential equation (1.9/2) (naturally, in order to ensure that $\varphi_{n,0} \equiv 1 \in \mathbb{S}_n^{\alpha,\beta}$, we will assume that $z = 0$ is at least a first order zero of (1.10/2));

© Ágoston Róth, 2018

- is of class $C^{\infty}([\alpha,\beta])$ and is EC on intervals of sufficiently small length $\beta-\alpha\in(0,\ell_n)$, where the so-called critical length $\ell_n>0$ (i.e., the supremum of the lengths of the intervals on which the given space is EC) can be determined as follows (see [Carnicer et al., 2004, Proposition 3.1]):
  - let
$$W_{[v_{n,0},v_{n,1},\ldots,v_{n,n}]}(u):=\left[v_{n,i}^{(j)}(u)\right]_{i=0,\ j=0}^{n,\ n},\ u\in[\alpha,\beta] \tag{1.11}$$
be the Wronskian matrix of those particular integrals
$$v_{n,i}:=\sum_{k=0}^{n}\rho_{i,k}\varphi_{n,k}\in\mathbb{S}_n^{\alpha,\beta},\ \{\rho_{i,k}\}_{k=0}^{n}\subset\mathbb{R},\ i=0,1,\ldots,n \tag{1.12}$$
of (1.9/2) which correspond to the boundary conditions
$$\begin{cases} v_{n,i}^{(j)}(\alpha) &= 0,\ j=0,\ldots,i-1,\\ v_{n,i}^{(i)}(\alpha) &= 1,\\ v_{n,i}^{(j)}(\beta) &= 0,\ j=0,\ldots,n-1-i,\end{cases} \tag{1.13}$$
i.e., the system $\{v_{n,i}(u):u\in[\alpha,\beta]\}_{i=0}^{n}$ is a bicanonical basis on the interval $[\alpha,\beta]$ such that the Wronskian (1.11/3) at $u=\alpha$ is a lower triangular matrix with positive (unit) diagonal entries;
  - consider the functions (or Wronskian determinants)
$$w_{n,i}(u):=\det W_{[v_{n,i},v_{n,i+1},\ldots,v_{n,n}]}(u),\ i=\left\lfloor\tfrac{n}{2}\right\rfloor+1,\ldots,n, \tag{1.14}$$
$$\theta_{n,i}(u):=(-1)^{n(n+1-i)}\det W_{[v_{n,i},v_{n,i+1},\ldots,v_{n,n}]}(-u),\ i=\left\lfloor\tfrac{n}{2}\right\rfloor+1,\ldots,n, \tag{1.15}$$
define the critical length
$$\ell_n:=\min_{i=\left\lfloor\frac{n}{2}\right\rfloor+1,\ldots,n}\min\{|u-\alpha|:w_{n,i}(u)=0\text{ or }\theta_{n,i}(u)=0,\ u\neq\alpha\} \tag{1.16}$$
(we write $\ell_n=+\infty$ whenever the Wronskian determinants (1.14/3) do not have real zeros other than α, moreover ℓ_n is infinite whenever the characteristic polynomial (1.10/2) has only real roots, otherwise it is finite a number that, in general, also depends on parameters resulting from the differential equation (1.9/2), i.e., on the real and imaginary parts of the complex zeros of the characteristic polynomial (1.10/2)).

We will denote the critical length of the derivative space $D\mathbb{S}_n^{\alpha,\beta}$ by ℓ_n' and, in order to ensure that $\mathbb{S}_n^{\alpha,\beta}$ is an EC space that also possesses a unique normalized B-basis (see [Carnicer et al., 2004, Theorem 4.1 and Corollary 4.1]), *from hereon we always assume that* $\beta\in(\alpha,\alpha+\ell_n')$. (The interval length $\beta-\alpha\in(0,\ell_n')$ can be considered as a shape or tension parameter. In order to avoid ambiguity, instead of ℓ_n and ℓ_n', in some cases, we will also use the notations $\ell\left(\mathbb{S}_n^{\alpha,\beta}\right)$ and $\ell'\left(\mathbb{S}_n^{\alpha,\beta}\right):=\ell\left(D\mathbb{S}_n^{\alpha,\beta}\right)$, respectively.) Following [Carnicer et al., 2004, p. 67], we will also refer to ℓ_n' as *critical length for design.*

Remark 1.1 (Concerning the endpoints of the definition domain) *Since the underlying vector space $\mathbb{S}_n^{\alpha,\beta}$ is translation invariant, it would be sufficient to study the properties of such spaces on intervals of the form $[0,h]$, where $h\in(0,\ell_n')$ and, consequently, the notation $\mathbb{S}_n^{\alpha,\beta}$ could be simplified to $\mathbb{S}_n^{h}$. Nevertheless, due to flexibility and some implementation details which may appear, e.g., in case of the control point based exact description (i.e., B-representation) of ordinary integral curves defined on intervals of appropriate length, we have designed/implemented all proposed algorithms on the interval $[\alpha,\beta]$, where $\beta-\alpha\in(0,\ell_n')$. In this way, during programming, users will have more comfort, e.g., they do not have to deal with phase changes.*

© Ágoston Róth, 2018

As we will see, the proposed algorithms do not assume that the characteristic polynomial (1.10/2) is an either odd or even function, but if this the case, the underlying EC space will also be reflection invariant, i.e., if $\mathbb{S}_n^{\alpha,\beta}$ denotes the solution space of (1.9/2), $v \in \mathbb{S}_n^{\alpha,\beta}$ and $x \in \mathbb{R}$ is fixed, then $v\,(u-x)$ also belongs to $\mathbb{S}_n^{\alpha,\beta}$, moreover the normalized B-basis functions (1.4/2) will share the symmetry property

$$b_{n,i}\,(u) = b_{n,n-i}\,(\alpha+\beta-u)\,,\ \forall u \in [\alpha,\beta]\,,\ i = 0,1,\ldots,\left\lfloor \tfrac{n}{2} \right\rfloor. \tag{1.17}$$

In this case, the critical length (1.16/3) can be determined (see [Carnicer et al., 2004, Proposition 3.2]) by means of the simpler formula

$$\ell_n := \min_{i=\left\lfloor \frac{n}{2} \right\rfloor+1,\ldots,n} \min\left\{|u-\alpha| : w_{n,i}\,(u) = 0,\ u \neq \alpha\right\}. \tag{1.18}$$

Based on both original and existing theoretical results, the main objective of the manuscript is to propose and implement general algorithms into a robust and flexible OpenGL and C++ based multi-threaded function library that can be used:

- to automatically generate and evaluate the derivatives of any order of both the ordinary basis and the normalized B-basis of a not necessarily reflection invariant EC space that comprises the constants, can be identified with the translation invariant solution space of the differential equation (1.9/2) and whose derivative space is also EC;
- to describe, generate, manipulate and render so-called B-curves defined as convex combinations of control points and normalized B-basis functions;
- to generate, manipulate and render so-called B-surfaces defined as tensor products of B-curves;
- to elevate the dimension of the underlying EC space(s) and consequently the order(s) of the B-curve (surface) that is rendered;
- to subdivide B-curves by means of general B-algorithms implied by the normalized B-basis of the given EC space and to extend this subdivision technique to B-surfaces;
- to generate transformations matrices that map the normalized B-bases of the applied EC spaces to their ordinary bases, in order to ensure control point configurations for the exact B-representation of large classes of integral curves and surfaces that are described in traditional parametric form by means of the ordinary bases of the used EC spaces.

1.2 Proposed algorithms

The full implementation details of the proposed function library and test cases that exemplify the usage of our algorithms appear in Chapters 2/13 and 3/269, respectively. After each mathematical notion, theorem or algorithm, this section also includes brief cross references both to the corresponding full implementation details and to the examples that should be considered by the reader. We deliberately omitted the proofs of our theoretical results from this user manual, however they can be found in the paper [Róth, 2019]. In order to formulate the input and output of our algorithms, we define the following control point based integral curves and surfaces.

Definition 1.1 (B-curves) *The convex combination*

$$\mathbf{c}_n\,(u) = \sum_{i=0}^{n} \mathbf{p}_i b_{n,i}\,(u)\,,\ u \in [\alpha,\beta]\,,\ 0 < \beta-\alpha < \ell'\left(\mathbb{S}_n^{\alpha,\beta}\right),\ \mathbf{p}_i = \left[p_i^{\ell}\right]_{\ell=0}^{\delta-1} \in \mathbb{R}^{\delta},\ \delta \geq 2 \tag{1.19}$$

described by means of the normalized B-basis (1.4/2) is called a B-curve of order n, where $[\mathbf{p}_i]_{i=0}^{n}$ denotes its control polygon.

© Ágoston Róth, 2018

Brief implementation details. In case of our implementation B-curves are represented by the class BCurve3 that is derived from the base abstract class LinearCombination3. The diagrams of these data structures are illustrated in Figs. 2.22/186 and 2.26/242, while their declarations and definitions can be found in Listing pairs 2.42/185–2.43/188 and 2.50/241–2.51/244, respectively.

Examples to consider. Listings 3.12/294–3.13/296 and Fig. 3.2/302 of Subsection 3.4.1/294 provide examples for the definition, generation, evaluation, differentiation, order elevation, subdivision and rendering of different types of B-curves. Listings 3.14/302–3.15/303 and Fig. 3.3/306 of Subsection 3.4.2/301 provide an example for the control point based exact description (or B-representation) of integral curves given in traditional (i.e., ordinary) parametric form.

Definition 1.2 (B-surfaces) *Denoting by*

$$\mathcal{B}_{n_r}^{\alpha_r,\beta_r} = \{b_{n_r,i_r}(u_r;\alpha_r,\beta_r) : u_r \in [\alpha_r,\beta_r]\}_{i_r=0}^{n_r},\ 0 < \beta_r - \alpha_r < \ell'\left(\mathbb{S}_{n_r}^{\alpha_r,\beta_r}\right),\ r = 0,1$$

the normalized B-bases of some EC spaces and using the tensor product of curves of type (1.19/4), one can define the B-surface

$$\mathbf{s}_{n_0,n_1}(u_0,u_1) = \sum_{i_0=0}^{n_0}\sum_{i_1=0}^{n_1}\mathbf{p}_{i_0,i_1}b_{n_0,i_0}(u_0;\alpha_0,\beta_0)\,b_{n_1,i_1}(u_1;\alpha_1,\beta_1),\ \mathbf{p}_{i_0,i_1} = \left[p_{i_0,i_1}^{\ell}\right]_{\ell=0}^{2} \in \mathbb{R}^3 \quad (1.20)$$

of order (n_0,n_1), where the matrix $[\mathbf{p}_{i_0,i_1}]_{i_0=0,\,i_1=0}^{n_0,\,n_1}$ forms a control net.

Brief implementation details. The proposed function library represents B-surfaces by the class BSurface3 that is derived from the base abstract class TensorProductSurface3. The diagrams of these types are illustrated in Figs. 2.23/194 and 2.27/251, while their declarations and definitions can be found in Listing pairs 2.44/194–2.45/197 and 2.52/250–2.53/253, respectively.

Examples to consider. Listings 3.16/307–3.17/308 and Figs. 3.4/319–3.7/322 of Subsection 3.5.1/306 provide examples for the definition, generation, evaluation, differentiation, order elevation, subdivision and rendering of different types of B-surfaces. Listings 3.16/307–3.17/308 and Figs. 3.8/334–3.9/335 of Subsection 3.5.2/318 give examples for both the control point based exact description (or B-representation) of integral surfaces given in traditional (i.e., ordinary) parametric form and the generation, evaluation, differentiation and rendering of isoparametric lines of B-surfaces.

1.2.1 Construction and differentiation of normalized B-basis functions in a large class of EC spaces

As already stated in Section 1.1/1, our implementation assumes that the underlying EC space $\mathbb{S}_n^{\alpha,\beta}$ corresponds to the solution space of a constant-coefficient homogeneous linear differential equation of type (1.9/2), where $\beta - \alpha \in (0,\ell'_n)$. This is not necessary for the correctness of the algorithms that will be presented in the forthcoming sections. The only reason for this additional assumption is the fact that in this case we have the possibility to handle a large family of (mixed) EC spaces in a unified way.

For example, if $\mathbf{i} = \sqrt{-1}$, $a,b \in \mathbb{R}$ and $z = a + \mathbf{i}b$ is an mth order ($m \geq 1$) zero of the characteristic polynomial (1.10/2), then based on its real and imaginary parts one has that:

- if $a,b \in \mathbb{R}\setminus\{0\}$, the conjugate complex number $\overline{z}$ is also a root of multiplicity m and consequently one obtains an algebraic-exponential-trigonometric (AET) mixed EC subspace

$$\left\langle \{u^r e^{au}\cos(bu),\, u^r e^{au}\sin(bu) : u \in [\alpha,\beta]\}_{r=0}^{m-1} \right\rangle \subseteq \mathbb{S}_n^{\alpha,\beta}; \quad (1.21)$$

© Ágoston Róth, 2018

- if $a \neq 0$, but $b = 0$, then one has the algebraic-exponential (AE) mixed EC subspace

$$\left\langle \{u^r e^{au} : u \in [\alpha, \beta]\}_{r=0}^{m-1} \right\rangle \subseteq \mathbb{S}_n^{\alpha,\beta}; \tag{1.22}$$

- if $a = 0$, but $b \neq 0$, then one obtains the algebraic-trigonometric (AT) mixed EC subspace

$$\left\langle \{u^r \cos(bu), u^r \sin(bu) : u \in [\alpha, \beta]\}_{r=0}^{m-1} \right\rangle \subseteq \mathbb{S}_n^{\alpha,\beta}; \tag{1.23}$$

- if $a = b = 0$ then one has the polynomial (P) EC subspace

$$\left\langle \{u^r : u \in [\alpha, \beta]\}_{r=0}^{m-1} \right\rangle \subseteq \mathbb{S}_n^{\alpha,\beta}. \tag{1.24}$$

This means that one can easily define ordinary (mixed) basis functions by simply specifying the factorization of the characteristic polynomial (1.10/2), i.e., one can create (mixed) EC spaces at run-time in an interactive way. As we will see, the zeroth and higher order (endpoint) derivatives of the ordinary basis function will play an important role in case of all proposed algorithms. In case of the aforementioned (mixed) EC subspaces one can overload function operators to compute the required derivatives for arbitrarily fixed orders – this possibility also motivates our assumption on the structure of the underlying EC space. Moreover, in real-world engineering, computer-aided design and manufacturing applications, usually one defines traditional parametric curves and surfaces by means of the ordinary basis functions presented above and, in our opinion, it would be nice to have a unified framework in which – apart from general order elevation and subdivision – one is also able to describe exactly important curves and surfaces by using control points and normalized B-basis functions.

In order to have normalizable bases in the vector space $\mathbb{S}_n^{\alpha,\beta}$, we also have to assume that $z = 0$ is at least a first order zero of the characteristic polynomial (1.10/2).

Once we have created an EC space of type $\mathbb{S}_n^{\alpha,\beta}$ by defining its ordinary basis $\mathcal{F}_n^{\alpha,\beta}$ (where $\beta - \alpha \in (0, \ell'_n)$), we also have to generate its unique normalized B-basis $\mathcal{B}_n^{\alpha,\beta}$. In order to achieve this and to be as self-contained as possible, we recall the construction process [Carnicer et al., 2004] of $\mathcal{B}_n^{\alpha,\beta}$. As we will see, the steps of this process can be fully implemented in case of the aforementioned (mixed) EC spaces.

Consider the bicanonical basis $\{v_{n,i}(u) : u \in [\alpha, \beta]\}_{i=0}^n$ formed by the particular integrals (1.12/3) determined by the boundary conditions (1.13/3). Let $W_{[v_{n,n}, v_{n,n-1}, \ldots, v_{n,0}]}(\beta)$ be the Wronskian matrix of the reverse ordered system $\{v_{n,n-i}(u) : u \in [\alpha, \beta]\}_{i=0}^n$ at the parameter value $u = \beta$ and obtain its Doolittle-type LU factorization

$$L \cdot U = W_{[v_{n,n}, v_{n,n-1}, \ldots, v_{n,0}]}(\beta), \tag{1.25}$$

where L is a lower triangular matrix with unit diagonal, while U is a non-singular upper triangular matrix. Calculate the inverse matrices

$$U^{-1} := \begin{bmatrix} \mu_{0,0} & \mu_{0,1} & \cdots & \mu_{0,n} \\ 0 & \mu_{1,1} & \cdots & \mu_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mu_{n,n} \end{bmatrix}, \quad L^{-1} := \begin{bmatrix} \lambda_{0,0} & 0 & \cdots & 0 \\ \lambda_{1,0} & \lambda_{1,1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \lambda_{n,0} & \lambda_{n,1} & \cdots & \lambda_{n,n} \end{bmatrix}$$

and construct the normalized B-basis

$$\mathcal{B}_n^{\alpha,\beta} = \left\{ b_{n,i}(u) = \lambda_{n-i,0} \widetilde{b}_{n,i}(u) : u \in [\alpha, \beta] \right\}_{i=0}^n \tag{1.26}$$

© Ágoston Róth, 2018

defined by

$$\left[\begin{array}{cccc} \widetilde{b}_{n,n}(u) & \widetilde{b}_{n,n-1}(u) & \cdots & \widetilde{b}_0(u) \end{array}\right] := \left[\begin{array}{cccc} v_{n,n}(u) & v_{n,n-1}(u) & \cdots & v_{n,0}(u) \end{array}\right] \cdot U^{-1}$$

and

$$\left[\begin{array}{cccc} \lambda_{0,0} & \lambda_{1,0} & \cdots & \lambda_{n,0} \end{array}\right]^T := L^{-1} \cdot \left[\begin{array}{cccc} 1 & 0 & \cdots & 0 \end{array}\right]^T.$$

If the characteristic polynomial (1.10/2) is an either even or odd function, then the underlying EC space $\mathbb{S}_n^{\alpha,\beta}$ is invariant under reflections, and in this special case one obtains the symmetry (1.17/4), i.e., we only need to determine the half of the basis functions (1.26/6).

Remark 1.2 (An alternative construction) *The non-negative bicanonical basis* $\left\{v_{n,i} : u \in [\alpha, \beta]\right\}_{i=0}^{n}$ *formed by the particular integrals (1.12/3) is a B-basis of the EC space* $\mathbb{S}_n^{\alpha,\beta}$ *(see [Carnicer et al., 2004, Theorem 2.4/(ii)]). Compared with the previously described LU decomposition based method, this B-basis can also be normalized by means of the normalizing coefficients*

$$c_{n,i} := -\sum_{r=0}^{i-1} c_{n,r} v_{n,r}^{(i)}(\alpha), \quad i = 1, \ldots, n, \tag{1.27}$$

where $c_{n,0} = 1$. *This means that the unique normalized B-basis functions of the underlying EC space* $\mathbb{S}_n^{\alpha,\beta}$ *could also be determined as the linear combinations*

$$b_{n,i}(u) := \sum_{i=0}^{n} c_{n,i} v_{n,i}(u), \quad u \in [\alpha, \beta], \quad i = 0, 1, \ldots, n. \tag{1.28}$$

Although the construction process (1.27/7)–(1.28/7) requires less computational effort, several numerical tests show that this alternative method is numerically less stable than (1.25/6)–(1.26/6).

Summarizing the calculations of the current section, we can state the next corollary that will be very important both in the formulation and in the implementation of all proposed algorithms.

Corollary 1.1 (Differentiation of normalized B-basis functions) *In general, the zeroth and higher order differentiation of the constructed normalized B-basis functions (1.26/6) can be reduced to the evaluation of formulas*

$$\begin{aligned} b_{n,n-i}^{(j)}(u) &= \lambda_{i,0} \widetilde{b}_{n,n-i}^{(j)}(u) \\ &= \lambda_{i,0} \sum_{r=0}^{i} \mu_{r,i} v_{n,n-r}^{(j)}(u) \\ &= \lambda_{i,0} \sum_{r=0}^{i} \mu_{r,i} \sum_{k=0}^{n} \rho_{n-r,k} \varphi_{n,k}^{(j)}(u), \quad \forall u \in [\alpha, \beta], \quad i = 0, 1, \ldots, n. \end{aligned} \tag{1.29}$$

Naturally, if the underlying EC space is reflection invariant, then formulas (1.29/7) have to be applied only for indices $i = 0, 1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor$, *since in this special case one also has that*

$$b_{n,i}^{(j)}(u) = (-1)^j b_{n,n-i}^{(j)}(\alpha + \beta - u), \quad \forall u \in [\alpha, \beta], \quad i = 0, 1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor. \tag{1.30}$$

Brief implementation details. Classes CharacteristicPolynomial and ECSpace, that provide interfaces for the factorization management of the general characteristic polynomial (1.10/2) and for the generation of the EC space implied by it, are illustrated in Figs. 2.24/211 and 2.25/217, while their declaration can be found in Listings 2.46/211 and 2.48/216, respectively. The full implementation details of these classes are presented in Listings 2.47/213 and 2.49/220, respectively. For example,

© Ágoston Róth, 2018

the construction process ((1.11/3)–(1.13/3), (1.25/6)–(1.26/6)) of the normalized B-basis functions of the underlying EC space and their differentiation formulas (1.29/7) are implemented in lines 567/230–809/234 and 834/234–911/235 of Listing 2.49/220, respectively.

Examples to consider. Deriving from the base class ECSpace, one can define special (like pure polynomial/trigonometric/hyperbolic and mixed exponential-trigonometric or algebraic-{trigonometric/hyperbolic/exponential-trigonometric}) EC spaces as it is presented by several examples in Listings 3.1/270 and 3.2/273 of Section 3.1/269. In Listings 3.10/291 and 3.11/291 of Section 3.3/290, we also provided examples for the evaluation, differentiation and rendering of both the ordinary basis and the normalized B-basis of different types of EC spaces. A possible output of these source codes is illustrated in Fig. 3.1/295.

1.2.2 General dimension and order elevation

Consider the EC spaces $\mathbb{S}_n^{\alpha,\beta}$ and $\mathbb{S}_{n+1}^{\alpha,\beta}$ such that $1 \in \mathbb{S}_n^{\alpha,\beta} \subset \mathbb{S}_{n+1}^{\alpha,\beta}$ and assume that spaces $D\mathbb{S}_n^{\alpha,\beta}$ and $D\mathbb{S}_{n+1}^{\alpha,\beta}$ of the derivatives are also EC on $[\alpha,\beta]$, i.e., $0 < \beta-\alpha < \min\left\{\ell'\left(\mathbb{S}_n^{\alpha,\beta}\right), \ell'\left(\mathbb{S}_{n+1}^{\alpha,\beta}\right)\right\}$, furthermore let us denote their unique normalized B-bases by $\{b_{n,i}(u) : u \in [\alpha,\beta]\}_{i=0}^{n}$ and $\{b_{n+1,i}(u) : u \in [\alpha,\beta]\}_{i=0}^{n+1}$, respectively. Following the results of [Mazure and Laurent, 1998, Theorem 3.1] and, as a slight difference, considering both endpoints of the definition domain $[\alpha,\beta]$ in order to minimize the maximal differentiation order of the lower and higher order normalized B-basis functions, one can state the next lemma.

Lemma 1.1 (General order elevation, [Mazure and Laurent, 1998]**)** *Using the notations of the section, the nth order B-curve (1.19/4) fulfills the identity*

$$\mathbf{c}_n(u) = \sum_{i=0}^{n} \mathbf{p}_i b_{n,i}(u) \equiv \sum_{i=0}^{n+1} \mathbf{p}_{1,i} b_{n+1,i}(u) =: \mathbf{c}_{n+1}(u), \quad \forall u \in [\alpha,\beta],$$

where $\mathbf{p}_{1,0} = \mathbf{p}_0$, $\mathbf{p}_{1,n+1} = \mathbf{p}_n$ *and*

$$\mathbf{p}_{1,i} = \left(1 - \frac{b_{n,i}^{(i)}(\alpha)}{b_{n+1,i}^{(i)}(\alpha)}\right)\mathbf{p}_{i-1} + \frac{b_{n,i}^{(i)}(\alpha)}{b_{n+1,i}^{(i)}(\alpha)}\mathbf{p}_i, \quad i = 1,\ldots,\left\lfloor\frac{n}{2}\right\rfloor, \tag{1.31}$$

$$\mathbf{p}_{1,n+1-i} = \frac{b_{n,n-i}^{(i)}(\beta)}{b_{n+1,n+1-i}^{(i)}(\beta)}\mathbf{p}_{n-i} + \left(1 - \frac{b_{n,n-i}^{(i)}(\beta)}{b_{n+1,n+1-i}^{(i)}(\beta)}\right)\mathbf{p}_{n+1-i}, \quad i = 1,\ldots,\left\lfloor\frac{n+1}{2}\right\rfloor. \tag{1.32}$$

Although Lemma 1.1/8 is valid for any nested EC spaces that fulfill the conditions $1 \in \mathbb{S}_n^{\alpha,\beta} \subset \mathbb{S}_{n+1}^{\alpha,\beta}$ and for which the derivative spaces $D\mathbb{S}_n^{\alpha,\beta}$ and $D\mathbb{S}_{n+1}^{\alpha,\beta}$ are also EC, in case of our implementation, we always assume that the higher dimensional EC space $\mathbb{S}_{n+1}^{\alpha,\beta}$ can also be identified with the solution space of a constant-coefficient homogeneous linear differential equation.

Brief implementation details. The declaration and definition of the class BCurve3 that represents B-curves of type (1.19/4) can be found in Listings 2.50/241 and 2.51/244, respectively. For example, formulas (1.31/8)–(1.32/8) of the general order elevation are implemented in lines 137/246–160/246 of Listing 2.51/244. Naturally, the results of Lemma 1.1/8 can also be extended to the general order elevation of B-surfaces of type (1.20/5) and our function library ensures this possibility as well: the declaration and definition of the class BSurface3 can be found in Listings 2.52/250 and 2.53/253, respectively, while the order elevation of B-surfaces is implemented in lines 296/258–515/261 of Listing 2.53/253.

© Ágoston Róth, 2018

Examples to consider. Examples for the order elevation of different types of B-curves and B-surfaces can be found in Listing pairs 3.12/294–3.13/296 and 3.16/307–3.17/308, respectively. Figs. 3.2/302 and 3.4/319–3.5/320 illustrate order elevated B-curves and B-surfaces, respectively, that were obtained as possible outputs of these source codes.

1.2.3 General B-algorithm

Theoretically, every normalized B-basis implies a B-algorithm for the subdivision of B-curves like (1.19/4), i.e., for an arbitrarily fixed parameter value $\gamma \in (\alpha, \beta)$ there exists a recursive corner cutting de Casteljau-like algorithm that starts with the initial conditions $\mathbf{p}_i^0(\gamma) \equiv \mathbf{p}_i,\ i = 0, 1, \ldots, n$ and recursively defines the subdivision points

$$\mathbf{p}_i^j(\gamma) = \left(1 - \xi_i^j(\gamma)\right) \cdot \mathbf{p}_i^{j-1}(\gamma) + \xi_i^j(\gamma) \cdot \mathbf{p}_{i+1}^{j-1}(\gamma),\ i = 0, \ldots, n-j,\ j = 1, \ldots, n, \tag{1.33}$$

where the explicit closed forms of the blending functions $\left\{\xi_i^j : [\alpha, \beta] \to [0, 1]\right\}_{i=0,\, j=1}^{n-j,\, n}$, in general, are either not known, or, apart from some very special cases (like Bézier curves), usually have non-linear complicated expressions even in low-dimensional EC spaces.

Using blossoms, B-algorithms were theoretically characterized in [Pottmann, 1993, Theorem 2.4] by means of a non-constructive procedure relying on unevaluated exterior products which, unfortunately, are not very useful concerning implementation. Even the author of [Pottmann, 1993, Theorem 2.4] states that the steps of His theoretical "*construction can be used for an implementation if the maps from the parameter interval to the axis [...] are simple. This is the case for rational Bézier curves where we have projective maps. Then we get exactly Farin's projective version of the de Casteljau algorithm ([Farin, '83]) see also [Farin & Worsey, '91]. The value of the algorithm for general TB-curves lies on the theoretical side.*"

Subdivision related constructive algorithms appear, e.g., in [Mainar and Peña, 1999, Section 3]. These methods use Neville elimination and are based both on *LU* decomposition of non-singular stochastic square matrices and on (unique) bidiagonal decompositions of non-singular lower triangular stochastic matrices [Gasca and Peña, 1996, Theorem 4.5].

Compared with subdivision techniques presented in [Pottmann, 1993, Theorem 2.4] and [Mainar and Peña, 1999, Section 3], in what follows, we propose simple, efficient and easily implementable constructive formulas which are based only on continuity conditions and avoid the evaluation of the non-diagonal entries of the triangular scheme that can be associated with every B-algorithm.

Let $\mathbb{S}_n^{\alpha,\beta}$ be an EC space, where $1 \in \mathbb{S}_n^{\alpha,\beta}$ and $\beta - \alpha \in (0, \ell_n')$. Consider the B-curve (1.19/4) and let $\mathcal{B}_n^{\alpha,\gamma} := \{b_{n,i}(u; \alpha, \gamma) : u \in [\alpha, \gamma]\}_{i=0}^n$ and $\mathcal{B}_n^{\gamma,\beta} := \{b_{n,i}(u; \gamma, \beta) : u \in [\gamma, \beta]\}_{i=0}^n$ be the unique normalized B-bases of the restricted EC spaces $\mathbb{S}_n^{\alpha,\gamma} := \operatorname{span} \mathcal{F}_n^{\alpha,\gamma} := \operatorname{span} \mathcal{F}_n^{\alpha,\beta}\big|_{[\alpha,\gamma]}$ and $\mathbb{S}_n^{\gamma,\beta} := \operatorname{span} \mathcal{F}_n^{\gamma,\beta} := \operatorname{span} \mathcal{F}_n^{\alpha,\beta}\big|_{[\gamma,\beta]}$, respectively. Consider also the diagonal entries

$$\left\{\boldsymbol{\lambda}_i(\gamma) := \mathbf{p}_0^i(\gamma)\right\}_{i=0}^n \text{ and } \left\{\boldsymbol{\varrho}_i(\gamma) := \mathbf{p}_i^{n-i}(\gamma)\right\}_{i=0}^n$$

of the triangular scheme

$$\begin{array}{llll}
\mathbf{p}_0 =: \boldsymbol{\lambda}_0(\gamma) & & & \\
\mathbf{p}_1 & \mathbf{p}_0^1(\gamma) =: \boldsymbol{\lambda}_1(\gamma) & & \\
\mathbf{p}_2 & \mathbf{p}_1^1(\gamma) & \mathbf{p}_0^2(\gamma) =: \boldsymbol{\lambda}_2(\gamma) & \\
\vdots & \vdots & \vdots & \cdots \quad \mathbf{p}_0^n(\gamma) =: \boldsymbol{\lambda}_n(\gamma) =: \boldsymbol{\varrho}_0(\gamma) \\
\mathbf{p}_{n-2} & \mathbf{p}_{n-2}^1(\gamma) & \mathbf{p}_{n-2}^2(\gamma) =: \boldsymbol{\varrho}_{n-2}(\gamma) & \\
\mathbf{p}_{n-1} & \mathbf{p}_{n-1}^1(\gamma) =: \boldsymbol{\varrho}_{n-1}(\gamma) & & \\
\mathbf{p}_n =: \boldsymbol{\varrho}_n(\gamma) & & &
\end{array}$$

© Ágoston Róth, 2018

that can be associated with the recursive process (1.33/9). Blending these points with the functions of the normalized B-bases $\mathcal{B}_n^{\alpha,\gamma}$ and $\mathcal{B}_n^{\gamma,\beta}$, the B-curve (1.19/4) of order n can be subdivided into the left and right arcs

$$\boldsymbol{l}_n(u) := \sum_{i=0}^{n} \boldsymbol{\lambda}_i(\gamma) \cdot b_{n,i}(u;\alpha,\gamma) \equiv \mathbf{c}_n(u), \ \forall u \in [\alpha,\gamma] \tag{1.34}$$

and

$$\mathbf{r}_n(u) := \sum_{i=0}^{n} \boldsymbol{\varrho}_i(\gamma) \cdot b_{n,i}(u;\gamma,\beta) \equiv \mathbf{c}_n(u), \ \forall u \in [\gamma,\beta], \tag{1.35}$$

respectively, that also fulfill the identities

$$\boldsymbol{l}_n^{(j)}(u) = \mathbf{c}_n^{(j)}(u), \ \forall u \in [\alpha,\gamma], \tag{1.36}$$

$$\mathbf{r}_n^{(j)}(u) = \mathbf{c}_n^{(j)}(u), \ \forall u \in [\gamma,\beta] \tag{1.37}$$

for all differentiation orders $j \geq 0$.

We close the current section with a recursive method by means of which one can determine the unknown diagonal subdivision points $\{\boldsymbol{\lambda}_i(\gamma)\}_{i=0}^n$ and $\{\boldsymbol{\varrho}_i(\gamma)\}_{i=0}^n$ even in the absence of the usually unknown blending functions $\left\{\xi_i^j : [\alpha,\beta] \to [0,1]\right\}_{i=0,\,j=1}^{n-j,\,n}$.

Theorem 1.1 (General B-algorithm) *Given an arbitrarily fixed parameter value $\gamma \in (\alpha,\beta)$ and starting with the initial conditions*

$$\boldsymbol{\lambda}_0(\gamma) = \mathbf{c}_n(\alpha) = \mathbf{p}_0, \tag{1.38}$$

$$\boldsymbol{\lambda}_n(\gamma) = \mathbf{c}_n(\gamma) = \boldsymbol{\varrho}_0(\gamma), \tag{1.39}$$

$$\boldsymbol{\varrho}_n(\gamma) = \mathbf{c}_n(\beta) = \mathbf{p}_n, \tag{1.40}$$

the unknown diagonal subdivision points $\{\boldsymbol{\lambda}_i(\gamma)\}_{i=0}^n$ and $\{\boldsymbol{\varrho}_i(\gamma)\}_{i=0}^n$ can be iteratively determined by means of the recursive formulas

$$\boldsymbol{\lambda}_i(\gamma) = \frac{1}{b_{n,i}^{(i)}(\alpha;\alpha,\gamma)} \left(\mathbf{c}_n^{(i)}(\alpha) - \sum_{j=0}^{i-1} \boldsymbol{\lambda}_j(\gamma) \cdot b_{n,j}^{(i)}(\alpha;\alpha,\gamma) \right), \ i = 1,\ldots,\left\lfloor \tfrac{n-1}{2} \right\rfloor, \tag{1.41}$$

$$\boldsymbol{\lambda}_{n-i}(\gamma) = \frac{1}{b_{n,n-i}^{(i)}(\gamma;\alpha,\gamma)} \left(\mathbf{c}_n^{(i)}(\gamma) - \sum_{j=0}^{i-1} \boldsymbol{\lambda}_{n-j}(\gamma) \cdot b_{n,n-j}^{(i)}(\gamma;\alpha,\gamma) \right), \ i = 1,\ldots,\left\lfloor \tfrac{n}{2} \right\rfloor, \tag{1.42}$$

$$\boldsymbol{\varrho}_i(\gamma) = \frac{1}{b_{n,i}^{(i)}(\gamma;\gamma,\beta)} \left(\mathbf{c}_n^{(i)}(\gamma) - \sum_{j=0}^{i-1} \boldsymbol{\varrho}_j(\gamma) \cdot b_{n,j}^{(i)}(\gamma;\gamma,\beta) \right), \ i = 1,\ldots,\left\lfloor \tfrac{n}{2} \right\rfloor, \tag{1.43}$$

$$\boldsymbol{\varrho}_{n-i}(\gamma) = \frac{1}{b_{n,n-i}^{(i)}(\beta;\gamma,\beta)} \left(\mathbf{c}_n^{(i)}(\beta) - \sum_{j=0}^{i-1} \boldsymbol{\varrho}_{n-j}(\gamma) \cdot b_{n,n-j}^{(i)}(\beta;\gamma,\beta) \right), \ i = 1,\ldots,\left\lfloor \tfrac{n-1}{2} \right\rfloor. \tag{1.44}$$

Brief implementation details. In case of B-curves, formulas (1.41/10)–(1.44/10) of the general B-algorithm are implemented in lines 199/247–322/249 of Listing 2.51/244. The subdivision technique presented in Theorem 1.1/10 can also be extended to B-surfaces of the type (1.20/5) as it is implemented in lines 521/262–781/266 of Listing 2.53/253.

Examples to consider. Examples for the subdivision of different types of B-curves and B-surfaces can be found in Listing pairs 3.12/294–3.13/296 and 3.16/307–3.17/308, respectively. Figs. 3.2/302 and 3.6/321–3.7/322 illustrate subdivided B-curves and B-surfaces, respectively, that were obtained as possible outputs of these source codes.

© Ágoston Róth, 2018

1.2.4 General basis transformation

The next theorem describes an efficient way for the evaluation of the matrix of the general basis transformation that maps the normalized B-basis $\mathcal{B}_n^{\alpha,\beta}$ to the ordinary basis $\mathcal{F}_n^{\alpha,\beta}$ of the EC space $\mathbb{S}_n^{\alpha,\beta}$, whenever $\beta-\alpha\in(0,\ell'_n)$.

Theorem 1.2 (Efficient general basis transformation) *The matrix form of the linear transformation that maps the normalized B-basis $\mathcal{B}_n^{\alpha,\beta}$ to the ordinary basis $\mathcal{F}_n^{\alpha,\beta}$ is*

$$\left[\ \varphi_{n,i}(u)\ \right]_{i=0}^{n}=\left[t_{i,j}^{n}\right]_{i=0,\ j=0}^{n,\ n}\cdot\left[\ b_{n,i}(u)\ \right]_{i=0}^{n},\ \forall u\in[\alpha,\beta], \tag{1.45}$$

where $t_{0,j}^n=1,\ j=0,1,\dots,n$ and $t_{i,0}^n=\varphi_{n,i}(\alpha),\ t_{i,n}^n=\varphi_{n,i}(\beta),\ i=0,1,\dots,n$, while the non-trivial entries of the matrix $[t_{i,j}^n]_{i=0,\,j=0}^{n,\,n}$ of the general basis transformation (1.45/11) can be determined by initializing the recursive formulas

$$t_{i,j}^{n}=\frac{1}{b_{n,j}^{(j)}(\alpha)}\left(\varphi_{n,i}^{(j)}(\alpha)-\sum_{k=0}^{j-1}t_{i,k}^{n}b_{n,k}^{(j)}(\alpha)\right),\ j=1,\dots,\left\lfloor\tfrac{n}{2}\right\rfloor, \tag{1.46}$$

and

$$t_{i,n-j}^{n}=\frac{1}{b_{n,n-j}^{(j)}(\beta)}\left(\varphi_{n,i}^{(j)}(\beta)-\sum_{k=0}^{j-1}t_{i,n-k}^{n}b_{n,n-k}^{(j)}(\beta)\right),\ j=1,\dots,\left\lfloor\tfrac{n-1}{2}\right\rfloor, \tag{1.47}$$

with the starting elements $\{t_{i,0}^n=\varphi_{n,i}(\alpha)\}_{i=1}^n$ and $\{t_{i,n}^n=\varphi_{n,i}(\beta)\}_{i=1}^n$, respectively, for all $i=1,\dots,n$.

Using [Róth, 2015a, Corollary 2.1, p. 43], one can also provide ready to use control point configurations for the exact description of those traditional integral parametric curves and (hybrid) surfaces that are specified by coordinate functions given as (products of separable) linear combinations of ordinary basis functions. Namely, by means of general basis transformations, one can implement the control point determining formulas (1.49/11) and (1.51/12) of the next two theorems.

Theorem 1.3 (Exact description of ordinary integral curves) *Using B-curves of the type (1.19/4), the ordinary integral curve*

$$\mathbf{c}(u)=\sum_{i=0}^{n}\boldsymbol{\lambda}_i\varphi_{n,i}(u),\ u\in[\alpha,\beta],\ 0<\beta-\alpha<\ell'_n,\ \boldsymbol{\lambda}_i\in\mathbb{R}^{\delta},\ \delta\geq 2 \tag{1.48}$$

fulfills the identity

$$\mathbf{c}(u)\equiv\mathbf{c}_n(u)=\sum_{j=0}^{n}\mathbf{p}_j b_{n,j}(u),\ \forall u\in[\alpha,\beta],$$

where

$$\left[\ \mathbf{p}_0\ \ \mathbf{p}_1\ \ \cdots\ \ \mathbf{p}_n\ \right]=\left[\ \boldsymbol{\lambda}_0\ \ \boldsymbol{\lambda}_1\ \ \cdots\ \ \boldsymbol{\lambda}_n\ \right]\cdot\left[t_{i,j}^{n}\right]_{i=0,\,j=0}^{n,\,n}. \tag{1.49}$$

Theorem 1.4 (Exact description of ordinary integral surfaces) *Let*

$$\mathcal{F}_{n_r}^{\alpha_r,\beta_r}=\{\varphi_{n_r,i_r}(u_r):u_r\in[\alpha_r,\beta_r]\}_{i_r=0}^{n_r},\ \varphi_{n_r,0}\equiv 1,\ 0<\beta_r-\alpha_r<\ell'\left(\mathbb{S}_{n_r}^{\alpha_r,\beta_r}\right)$$

be the ordinary basis and

$$\mathcal{B}_{n_r}^{\alpha_r,\beta_r}=\{b_{n_r,j_r}(u_r):u_r\in[\alpha_r,\beta_r]\}_{j_r=0}^{n_r}$$

© Ágoston Róth, 2018

be the normalized B-basis of some EC vector space $\mathbb{S}_{n_r}^{\alpha_r,\beta_r}$ *of functions and denote by* $[t_{i_r,j_r}^{n_r}]_{i_r=0,\ j_r=0}^{n_r,\ n_r}$ *the regular square matrix that transforms* $\mathcal{B}_{n_r}^{\alpha_r,\beta_r}$ *to* $\mathcal{F}_{n_r}^{\alpha_r,\beta_r}$*, where* $r=0,1$*. Consider also the ordinary integral surface*

$$\mathbf{s}(u_0,u_1)=\left[\begin{array}{ccc} s^0(u_0,u_1) & s^1(u_0,u_1) & s^2(u_0,u_1)\end{array}\right]^T\in\mathbb{R}^3,\ (u_0,u_1)\in[\alpha_0,\beta_0]\times[\alpha_1,\beta_1], \tag{1.50}$$

where

$$s^k(u_0,u_1)=\sum_{\zeta=0}^{\sigma_k-1}\prod_{r=0}^{1}\left(\sum_{i_r=0}^{n_r}\lambda_{n_r,i_r}^{k,\zeta}\varphi_{n_r,i_r}(u_r)\right),\ \sigma_k\geq 1,\ k=0,1,2.$$

Then, by using B-surfaces of the type (1.20/5), the ordinary surface (1.50/12) fulfills the identity

$$\mathbf{s}(u_0,u_1)\equiv\mathbf{s}_{n_0,n_1}(u_0,u_1)=\sum_{j_0=0}^{n_0}\sum_{j_1=0}^{n_1}\mathbf{p}_{j_0,j_1}b_{n_0,j_0}(u_0)\,b_{n_1,j_1}(u_1),\ \forall(u_0,u_1)\in[\alpha_0,\beta_0]\times[\alpha_1,\beta_1],$$

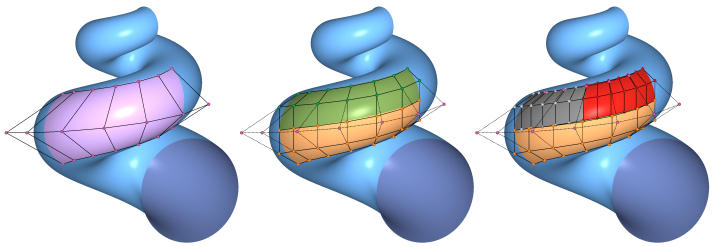
where the control points $\mathbf{p}_{j_0,j_1}=[p_{j_0,j_1}^k]_{k=0}^2\in\mathbb{R}^3$ *are defined by the coordinates*

$$p_{j_0,j_1}^k=\sum_{\zeta=0}^{\sigma_k-1}\prod_{r=0}^{1}\left(\sum_{i_r=0}^{n_r}\lambda_{n_r,i_r}^{k,\zeta}t_{i_r,j_r}^{n_r}\right),\ k=0,1,2. \tag{1.51}$$

Brief implementation details. Formulas (1.46/11)–(1.47/11) of the general basis transformation are implemented in lines 915/235–982/236 of Listing 2.49/220. Using B-curves/surfaces of the type (1.19/4)/(1.20/5) and applying formulas (1.49/11)/(1.51/12), the proposed basis transformation can be used for the control point based exact description (or B-representation) of large families of integral or rational ordinary curves/surfaces that may be important in several areas of applied mathematics, since the investigated large class of EC vector spaces also comprise functions that appear in the traditional (or ordinary) parametric description of famous geometric objects like: *ellipses*; *epi- and hypocycloids*; *epi- and hypotrochoids*; *Lissajous curves*; *torus knots*; *foliums*; *rose curves*; *the witch of Agnesi*; *the cissoid of Diocles*; *Bernoulli's lemniscate*; *Zhukovsky airfoil profiles*; *cycloids*; *hyperbolas*; *helices*; *catenaries*; *Archimedean and logarithmic spirals*; *ellipsoids*; *tori*; *hyperboloids*; *catenoids*; *helicoids*; *ring, horn and spindle Dupin cyclides*; *non-orientable surfaces such as Boy's and Steiner's surfaces and the Klein Bottle of Gray*. Using B-curves of type (1.19/4), the control point based exact description of ordinary integral curves of type (1.48/11) is implemented in lines 350/249–375/250 of Listing 2.51/244, while the B-representation of ordinary integral surfaces of type (1.50/12) is implemented in lines 787/266–826/267 of Listing 2.53/253.

Examples to consider. Listings 3.14/302–3.15/303 and Fig. 3.3/306 of Subsection 3.4.2/301 provide an example for the control point based exact description (or B-representation) of several arcs of the logarithmic spiral (3.8/301). Listings 3.18/321–3.19/324 and Figs. 3.8/334– 3.9/335 of Subsection 3.5.2/318 give an example for the B-representations of the transcendental exponential-trigonometric ordinary integral surface patches (3.9/320).

© Ágoston Róth, 2018



Exceptions • Smart pointers • Cartesian, homogeneous and texture coordinates • Colors, lights, materials • Different template and specialized matrices • Constants and utility functions • Shader programs • Generic curves and simple triangle meshes • Abstract linear combinations and tensor product surfaces • Characteristic polynomials, EC spaces, B-curves and B-surfaces

2

Full implementation details

The entire implementation of the proposed function library is explained in the exhaustively commented listings of the current chapter. Our library assumes that the user has a multi-core CPU and also a GPU that is compatible at least with the desktop variant of OpenGL 3.0. In order to render the geometry, we use vertex buffer objects through the OpenGL Extension Wrangler (GLEW) library and for multi-threading we rely on a C++ compiler that at least supports OpenMP 2.0. Apart from GLEW no other external dependencies are used.

In its current state, the proposed function library consists of two main packages. The first of these is called **Core** and includes data types that represent:

- exceptions (Exception);
- Cartesian (Cartesian3), homogeneous (Homogeneous3) and texture coordinates (TCoordinate4);
- color components (Color4), different types of lights (DirectionalLight, PointLight, Spotlight) and materials (Material);
- mathematical constants, generic rectangular (Matrix<T>, RowMatrix<T>, ColumnMatrix<T>) or triangular template matrices (TriangularMatrix<T>), real matrices (RealMatrix: public Matrix<double>), some real matrix decompositions (PLUDecomposition, FactorizedUnpivotedLUDecomposition, SVDecomposition), generic and derived OpenGL transformations (GLTransformation, Translate, Scale, Rotate, PerspectiveProjection, OrthogonalProjection, LookAt) and Pascal triangles of binomial coefficients (PascalTriangle: public TriangularMatrix<double>);
- generic and specialized smart pointers (SmartPointer<T,TSP,TOP,TICP,TCP>, SP<T>::DefaultPrimitive, SP<T>::Default, SP<T>::Array, SP<T>::DestructiveCopy, SP<T>::NonIntrusiveReferenceCounting) that provide different storage, ownership, implicit conversion and checking policies (StoragePolicy<T>::Default, StoragePolicy<T>::Array, OwnershipPolicy<T>, ImplicitConversionPolicy, CheckingPolicy<T>::NoCheck, CheckingPolicy<T>::RejectNullDereferenceOrIndirection, CheckingPolicy<T>::RejectNull, CheckingPolicy<T>::AssertNullDereferenceOrIndirection, CheckingPolicy<T>::AssertNull) in order to avoid memory leaks and to ensure exception safety (one of the most frequently used smart pointers will be

© Ágoston Róth, 2018

the specialized variant SP<T>::Default that ensures default storage and deep copy policies, disallows implicit conversion and rejects null dereference or indirection);

- generic curves (GenericCurve3) and abstract linear combinations (LinearCombination3);
- triangular faces (TriangularFace), simple triangle meshes (TriangleMesh3) and abstract tensor product surfaces (TensorProductSurface3);
- shader programs[1] (ShaderProgram) written in the OpenGL Shading Language and used for rendering geometries (like control polygons and nets, or generic curves and triangle meshes obtained, e.g., as the images of linear combinations and tensor product surfaces, respectively).

The previously listed classes serve the definition, implementation and testing of the following data types that realize our main objectives and are included in the second main package called **EC**:

- the class CharacteristicPolynomial ensures the factorization management and evaluation of characteristic polynomials of type (1.10);
- EC spaces that comprise the constants and can be identified with the solution spaces of differential equations of type (1.9) will be instances of the class ECSpace;
- B-curves of type (1.19) are represented by the class BCurve3 that is derived from the abstract base class LinearCombination3 and is based on the results of Corollary 1.1, Lemma 1.1 and Theorems 1.1, 1.2–1.3;
- B-surfaces of type (1.20) are represented by the class BSurface3 that is a descendant of the abstract base class TensorProductSurface3 and is based on Theorem 1.4 and on the natural extensions of Corollary 1.1, Lemma 1.1 and Theorem 1.1.

The tree-view of the header files that can be included from our library is illustrated in Fig. 2.1/15 and, in what follows, we detail the implementation of each of the previously listed data types.

2.1 Exceptions

We provide a simple class for possible exception handling as it is illustrated in Listing 2.1/14.

Listing 2.1. Exceptions (**Core/Exceptions.h**)

```
1  #ifndef EXCEPTIONS_H
2  #define EXCEPTIONS_H
3  #include <iostream>
4  #include <string>
5  namespace cagd
6  {
7      class Exception
8      {
9          // overloaded friend output to stream operator
10         friend std::ostream& operator <<(std::ostream &lhs, const Exception &rhs);
11     protected:
12         std::string _reason;
13     public:
14         // special constructor
```

[1] For convenience we have also provided shader programs for simple (flat) color shading, for two-sided per pixel lighting that is able to handle user-defined directional, point and spotlights with uniform front and back materials, and another one for reflection lines that are combined with two-sided per pixel lighting. All figures of the current user manual and of the paper [Róth, 2019] were rendered by using these shader programs.

© Ágoston Róth, 2018

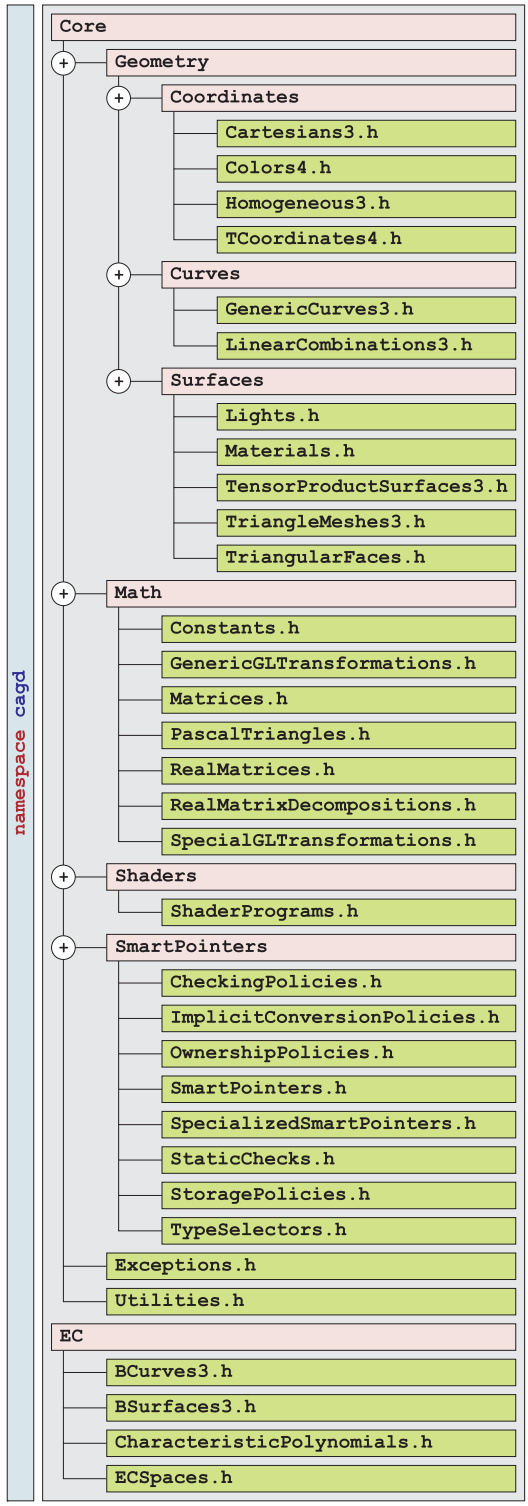


Fig. 2.1: Tree-view of the header files of the proposed function library

© Ágoston Róth, 2018

```
15          Exception(const std::string &reason): _reason(reason)
16          {
17          }

18          const std::string& reason() const
19          {
20              return _reason;
21          }
22      };

23      inline std::ostream& operator <<(std::ostream& lhs, const Exception &rhs)
24      {
25          return lhs << rhs._reason;
26      }
27  }

28  #endif // EXCEPTIONS_H
```

2.2 Smart pointers

Based on [Alexandrescu, 2001] we provide several classes to handle generic smart pointers (SmartPointer) that ensure different storage, ownership, implicit conversion and checking policies in order to avoid memory leaks and to ensure exception safety. The following subsections at first detail these different policy classes, then they also define generic and specialized smart pointers by multiple inheritance.

2.2.1 Type selectors

Type selectors will be used for both enabling/disabling the implicit conversion policy and to determine the copied type in case of the storage policies.

Listing 2.2. Type selectors (**Core/SmartPointers/TypeSelectors.h**)

```
1  #ifndef TYPESELECTORS_H
2  #define TYPESELECTORS_H

3  namespace cagd
4  {
5      // based on the logical value of the compile time constant select_first_type, the class selects one of the
6      // types U and V
7      template <bool select_first_type, typename U, typename V>
8      class TypeSelector
9      {
10     public:
11         typedef U Result;
12     };

13     // specializations for all possible cases when the value of select_first_type is false
14     template <typename U, typename V>
15     class TypeSelector<false, U, V>
16     {
17     public:
18         typedef V Result;
19     };

20     // logical template function that decides whether the given typename T denotes a primitive type or not
21     template <typename T>
22     inline bool primitive(T)
23     {
24         return false; // with the exception of the specializations listed below, the function always returns false
25     }

26     // specializations that belong to primitive types always return true
27     template <>
28     inline bool primitive(int)
```


© Ágoston Róth, 2018

```
29      {
30          return true;
31      }
32      template <>
33      inline bool primitive(unsigned int)
34      {
35          return true;
36      }
37      template <>
38      inline bool primitive(short int)
39      {
40          return true;
41      }
42      template <>
43      inline bool primitive(unsigned short int)
44      {
45          return true;
46      }
47      template <>
48      inline bool primitive(long int)
49      {
50          return true;
51      }
52      template <>
53      inline bool primitive(unsigned long int)
54      {
55          return true;
56      }
57      template <>
58      inline bool primitive(long long int)
59      {
60          return true;
61      }
62      template <>
63      inline bool primitive(unsigned long long int)
64      {
65          return true;
66      }
67      template <>
68      inline bool primitive(signed char)
69      {
70          return true;
71      }
72      template <>
73      inline bool primitive(char)
74      {
75          return true;
76      }
77      template <>
78      inline bool primitive(unsigned char)
79      {
80          return true;
81      }
82      template <>
83      inline bool primitive(float)
84      {
85          return true;
86      }
87      template <>
88      inline bool primitive(double)
89      {
90          return true;
```


© Ágoston Róth, 2018

```
91      }
92      template <>
93      inline bool primitive(long double)
94      {
95          return true;
96      }
97      template <>
98      inline bool primitive(wchar_t)
99      {
100         return true;
101     }
102     // if we have omitted some primitive types, then this file should be appended by following the given examples above...
103 }
104 #endif // TYPESELECTORS_H
```

2.2.2 Implicit conversion policies

The class diagram of implicit conversion policies is illustrated in Fig. 2.2/18. Their implementation can be found in Listing 2.3/18.

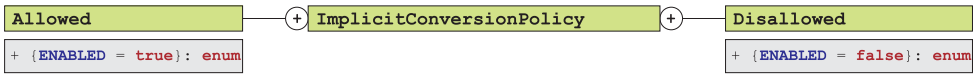


Fig. 2.2: Possible implicit conversion policies ensured by our smart pointers

Listing 2.3. Implicit conversion policies (**Core/SmartPointers/ImplicitConversionPolicies.h**)

```
1  #ifndef IMPLICITCONVERSIONPOLICIES_H
2  #define IMPLICITCONVERSIONPOLICIES_H
3  namespace cagd
4  {
5      // Using implicit conversion policies, one can allow or disallow to convert smart pointers to raw pointers of
6      // stored type.
7      class ImplicitConversionPolicy
8      {
9      public:
10         class Allowed
11         {
12         public:
13             enum {ENABLED = true};
14         };
15         class Disallowed
16         {
17         public:
18             enum {ENABLED = false};
19         };
20     };
21 }
22 #endif // IMPLICITCONVERSIONPOLICIES_H
```

2.2.3 Static checks

In order to include the possible compile time errors related to our smart pointers into the error list generated by the compiler, we need to provide a mechanism that performs static checks by using a macro. The definition of this macro can be found in Listing 2.4/18.

© Ágoston Róth, 2018

Listing 2.4. Static checks (**Core/SmartPointers/StaticChecks.h**)

```
#ifndef STATICCHECKS_H
#define STATICCHECKS_H

namespace cagd
{
    // The non-specialized variant of the class CompileTimeError (i.e., which belongs to the compile time constant
    // false) has only a declaration. Compared to this, the specialization that corresponds to the compile time
    // constant true also provides a definition. Therefore, in case of false values any instantiation will generate a
    // compile time error.
    template <bool> class CompileTimeError;
    template <>     class CompileTimeError<true>{};

    // The macro STATIC_CHECK will generate a compile time error whenever the given expression evaluates to false.
    // In such cases the given message will appear in the list of compile time errors.
    #define STATIC_CHECK(expression, message){\
        CompileTimeError<((expression) != false)> ERROR##message;}
}

#endif // STATICCHECKS_H
```

2.2.4 Ownership policies

Currently one can choose from the no-copy, deep copy, deep primitive copy, destructive copy and non-intrusive reference counting ownership policies. There are other (like copy-on-write, intrusive reference counting and reference linking) ownership policies as well that can be incorporated into the current implementation by following the examples provided in Listing 2.6/20.

Note that, if one applies our smart pointers with deep copy ownership policy to one of their custom types, then the corresponding class should provide a (virtual) constant cloning member function named “clone” that returns a raw pointer to a new dynamically allocated object of the same type obtained by the invocation of the copy constructor of the class (the user is responsible for the correctness of the copy constructor). If such a custom class serves as the base of further derived classes, then the cloning function of the base should be virtual and has to be redeclared and redefined in each of the derived classes. Another possibility for the users is to derive all their custom types from the abstract base class provided in Listings 2.5/19.

Listing 2.5. Abstract base class for deep copy ownership policy in case of custom types

```
#ifndef ABSTRACTBASES_H
#define ABSTRACTBASES_H

namespace cagd
{
    class AbstractBase
    {
    public:
        // pure virtual method that has to redeclared and defined in all derived classes
        virtual AbstractBase* clone() const = 0;

        // virtual destructor
        virtual ~AbstractBase() {}
    };
}

#endif // ABSTRACTBASES_H
```

The class diagram of the provided ownership policies is illustrated in Fig. 2.3/20, while their implementation can be found in Listing 2.6/20.

© Ágoston Róth, 2018

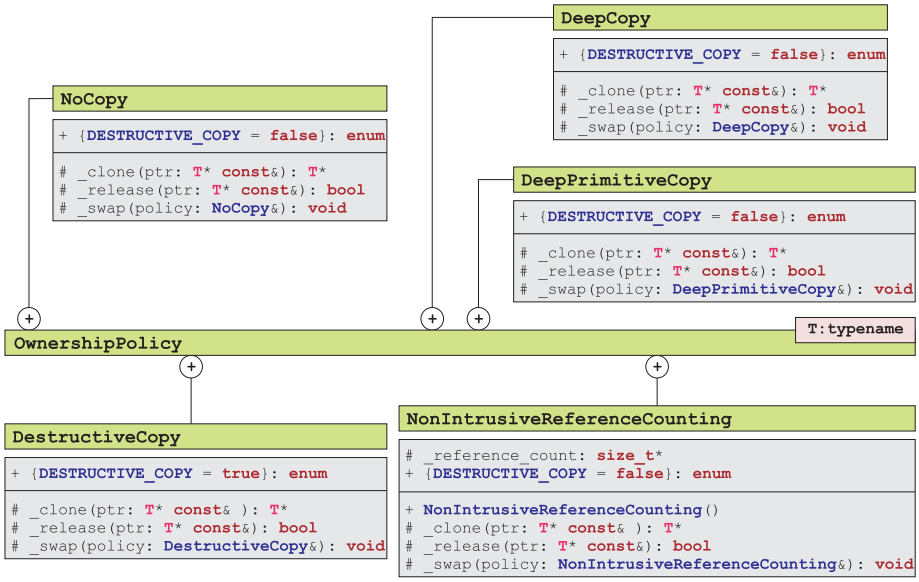


Fig. 2.3: Possible ownership policies ensured by our smart pointers

Listing 2.6. Ownership policies (**Core/SmartPointers/OwnershipPolicies.h**)

```
#ifndef OWNERSHIPPOLICIES_H
#define OWNERSHIPPOLICIES_H

#include "StaticChecks.h"
#include "TypeSelectors.h"
#include <new>

namespace cagd
{
    // Currently one can choose from the no-copy, deep copy, deep primitive copy, destructive copy and
    // non-intrusive reference counting ownership policies.
    template <typename T>
    class OwnershipPolicy
    {
    public:
        // no copy ownership policy
        class NoCopy
        {
        public:
            enum {DESTRUCTIVE_COPY = false};

        protected:
            T*    _clone(T* const& ptr);
            bool _release(T* const& ptr);
            void _swap(NoCopy& policy);
        };

        // deep copy ownership policy for custom types
        class DeepCopy
        {
        public:
            enum {DESTRUCTIVE_COPY = false};

        protected:
            T*    _clone(T* const& ptr);
            bool _release(T* const& ptr);
            void _swap(DeepCopy& policy);
        };
```


© Ágoston Róth, 2018

```
34          // deep copy ownership policy for primitive types
35          class DeepPrimitiveCopy
36          {
37          public:
38              enum {DESTRUCTIVE_COPY = false};
39          protected:
40              T*    _clone(T* const& ptr);
41              bool _release(T* const& ptr);
42              void _swap(DeepPrimitiveCopy& policy);
43          };
44          // destructive copy ownership policy
45          class DestructiveCopy
46          {
47          public:
48              enum {DESTRUCTIVE_COPY = true};
49          protected:
50              T*    _clone(T* & ptr);
51              bool _release(T* const& ptr);
52              void _swap(DestructiveCopy& policy);
53          };
54          // non-intrusive reference counting ownership policy
55          class NonIntrusiveReferenceCounting
56          {
57          private:
58              size_t *_reference_count; // records the total number of references
59          public:
60              enum {DESTRUCTIVE_COPY = false};
61              // default constructor
62              NonIntrusiveReferenceCounting();
63          protected:
64              T*    _clone(T* const& ptr);
65              bool _release(T* const& ptr);
66              void _swap(NonIntrusiveReferenceCounting& policy);
67          };
68      };
69      // implementation of the no-copy ownership policy
70      template <typename T>
71      inline T* OwnershipPolicy<T>::NoCopy::_clone(T* const& ptr)
72      {
73          STATIC_CHECK(false, The_no_copy_ownership_policy_disallows_copying);
74      }
75      template <typename T>
76      inline bool OwnershipPolicy<T>::NoCopy::_release(T* const& ptr)
77      {
78          return true;
79      }
80      template <typename T>
81      inline void OwnershipPolicy<T>::NoCopy::_swap(NoCopy& policy)
82      {
83      }
84      // implementation of the deep copy ownership policy for custom types
85      template <typename T>
86      inline T* OwnershipPolicy<T>::DeepCopy::_clone(T* const& ptr)
87      {
88          return (ptr ? ptr->clone() : nullptr);
89      }
90      template <typename T>
91      inline bool OwnershipPolicy<T>::DeepCopy::_release(T* const& /*ptr*/)
92      {
93          return true;
94      }
```


© Ágoston Róth, 2018

```
95      template <typename T>
96      inline void OwnershipPolicy<T>::DeepCopy::_swap(DeepCopy& /*policy*/)
97      {
98      }

99      // implementation of the deep copy ownership policy for primitive types
100     template <typename T>
101     inline T* OwnershipPolicy<T>::DeepPrimitiveCopy::_clone(T* const& ptr)
102     {
103         if (ptr && primitive(*ptr))
104         {
105             return new (std::nothrow) T(*ptr);
106         }

107         return nullptr;
108     }

109     template <typename T>
110     inline bool OwnershipPolicy<T>::DeepPrimitiveCopy::_release(T* const& ptr)
111     {
112         return true;
113     }

114     template <typename T>
115     inline void OwnershipPolicy<T>::DeepPrimitiveCopy::_swap(DeepPrimitiveCopy& policy)
116     {
117     }

118     // implementation of the destructive copy ownership policy:
119     // during assignment or copying the ownership is transfered to the target smart pointer and the raw pointer
120     // stored by the copied pointer is set to null
121     template <typename T>
122     inline T* OwnershipPolicy<T>::DestructiveCopy::_clone(T* & ptr)
123     {
124         T* aux(ptr);
125         ptr = nullptr;
126         return aux;
127     }

128     template <typename T>
129     inline bool OwnershipPolicy<T>::DestructiveCopy::_release(T* const& /*ptr*/)
130     {
131         return true;
132     }

133     template <typename T>
134     inline void OwnershipPolicy<T>::DestructiveCopy::_swap(DestructiveCopy& /*policy*/)
135     {
136     }

137     // implementation of the non-intrusive reference counting ownership policy:
138     // this ownership policy allows the sharing of a dynamically allocated object's memory address by multiple
139     // smart pointers, records the number of total references, and destroys the referenced object when the total
140     // reference count becomes zero
141     template <typename T>
142     inline OwnershipPolicy<T>::NonIntrusiveReferenceCounting::
143         NonIntrusiveReferenceCounting():
144         _reference_count(new (std::nothrow) size_t(1))
145     {
146     }

147     template <typename T>
148     inline T* OwnershipPolicy<T>::NonIntrusiveReferenceCounting::_clone(T* const& ptr)
149     {
150         if (_reference_count)
151         {
152             *_reference_count += 1; // during copying the total reference count is increased
153         }

154         return ptr;
155     }

156     template <typename T>
157     inline bool OwnershipPolicy<T>::NonIntrusiveReferenceCounting::_release(
```


© Ágoston Róth, 2018

```
158                 T* const& ptr)
159     {
160         if (_reference_count)
161         {
162             // when a reference counting smart pointer goes out of scope the total reference count is decreased
163             *_reference_count -= 1;
164             // if the total reference count becomes 0, we delete the reference tracking pointer and we indicate that
165             // the stored raw pointer should also be deleted
166             if (*_reference_count == 0)
167             {
168                 delete _reference_count, _reference_count = nullptr;
169                 return true;
170             }
171         }
172         // if the total reference count is not 0, the stored raw pointer should be not deleted
173         return false;
174     }
175     template <typename T>
176     inline void OwnershipPolicy<T>::NonIntrusiveReferenceCounting::_swap(
177             NonIntrusiveReferenceCounting& policy)
178     {
179         // The function _swap will be called by a temporary smart pointer in the assignment operator of the
180         // class SmartPointer that will be discussed later in Listing 2.9/29.
181         // After the exchange of the policies, the aforementioned temporary smart pointer will go out of scope
182         // and consequently the number of total references will be further decreased by 1. In order to avoid this,
183         // initially we increase the number of total referecenes by 1.
184         if (policy._reference_count)
185         {
186             *policy._reference_count += 1;
187         }
188         size_t *aux              = _reference_count;
189         _reference_count         = policy._reference_count;
190         policy._reference_count = aux;
191     }
192 }
193 #endif // OWNERSHIPPOLICIES_H
```

2.2.5 Storage policies

The user can choose between the default (i.e., non-array) and array storage policies. Note that the array storage policy can only be combined with the no-copy ownership policy, since we do not know the size of the array referenced by the raw pointer that is stored in such a smart pointer. The class diagram of the aforementioned storage policies are illustrated in Fig. 2.4/23, while their implementations can be found in Listing 2.7/23.

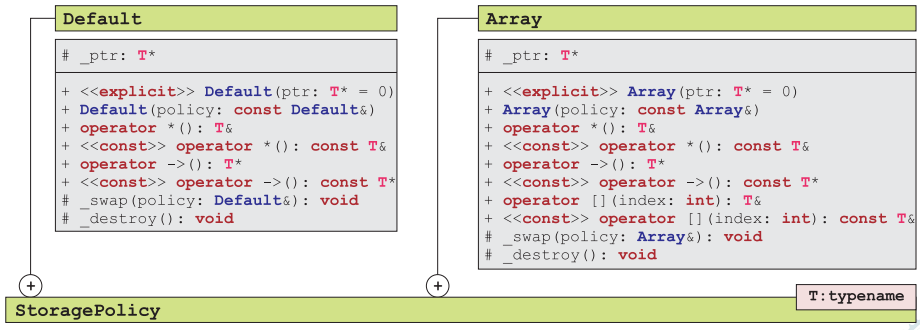


Fig. 2.4: Possible storage policies ensured by our smart pointers

© Ágoston Róth, 2018

Listing 2.7. Storage policies (**Core/SmartPointers/StoragePolicies.h**)

```
#ifndef STORAGEPOLICIES_H
#define STORAGEPOLICIES_H

namespace cagd
{
    // The user can choose between the default (i.e., non-array) and array storage policies.
    template <typename T>
    class StoragePolicy
    {
    public:

        // default storage policiy
        class Default
        {
        protected:
            T* _ptr;

        public:
            // explicit special/default constructor
            explicit Default(T* ptr = nullptr);

            // copy constructor
            Default(const Default& policy);

            // overloaded dereferencing operators
            T& operator *();
            const T& operator *() const;

            // overloaded indirection operators
            T* operator ->();
            const T* operator ->() const;

        protected:
            // it will be called during policy exchanges
            void _swap(Default& policy);

            // it is responsible for the deallocation of the object referenced by the stored pointer _ptr
            void _destroy();
        };

        // array storage policy:
        // it can only be combined with the no-copy ownership policy, since we do not know the size of the array
        // referenced by the stored raw pointer _ptr
        class Array
        {
        protected:
            T* _ptr;

        public:
            // explicit special/default constructor
            explicit Array(T* ptr = nullptr);

            // copy constructor
            Array(const Array& policy);

            // overloaded dereferencing operators
            T& operator *();
            const T& operator *() const;

            // overloaded indexing operators
            T& operator [](int index);
            const T& operator [](int index) const;

            // overloaded indirection operators
            T* operator ->();
            const T* operator ->() const;

        protected:
            // it will be called during policy exchanges
            void _swap(Array& policy);
```


© Ágoston Róth, 2018

```
56              // it is responsible for the deallocation of the array referenced by the stored pointer _ptr
57              void _destroy();
58          };
59      };

60      // Implementation of the default storage policy.

61      // special/default constructor
62      template <typename T>
63      inline StoragePolicy<T>::Default::Default(T* ptr): _ptr(ptr)
64      {
65      }

66      // copy constructor
67      template <typename T>
68      inline StoragePolicy<T>::Default::Default(const Default& /*policy*/)
69      {
70          // the stored raw pointer _ptr will be initialized by the clone function of the applied ownership policy
71      }

72      // overloaded dereferencing operators
73      template <typename T>
74      inline T& StoragePolicy<T>::Default::operator *()
75      {
76          return *_ptr;
77      }

78      template <typename T>
79      inline const T& StoragePolicy<T>::Default::operator *() const
80      {
81          return *_ptr;
82      }

83      // overloaded indirection operators
84      template <typename T>
85      inline T* StoragePolicy<T>::Default::operator ->()
86      {
87          return _ptr;
88      }

89      template <typename T>
90      inline const T* StoragePolicy<T>::Default::operator ->() const
91      {
92          return _ptr;
93      }

94      // it will be called during policy exchanges
95      template <typename T>
96      inline void StoragePolicy<T>::Default::_swap(Default& policy)
97      {
98          T* aux(_ptr);
99          _ptr = policy._ptr;
100         policy._ptr = aux;
101     }

102     // deallocation of the object referenced by the raw pointer _ptr
103     template <typename T>
104     inline void StoragePolicy<T>::Default::_destroy()
105     {
106         if (_ptr)
107         {
108             delete _ptr, _ptr = nullptr;
109         }
110     }

111     // Implementation of the array storage policy.

112     // special/default constructor
113     template <typename T>
114     inline StoragePolicy<T>::Array::Array(T* ptr): _ptr(ptr)
115     {
116     }

117     // copy constructor
```


© Ágoston Róth, 2018

```
118     template <typename T>
119     inline StoragePolicy<T>::Array::Array(const Array& /*policy*/)
120     {
121         // the stored raw pointer _ptr will be initialized by the clone function of the applied ownership policy
122         // do not forget: the array storage policy can only be used together with the no-copy ownership policy,
123         // since we do not know the size of the array that is referenced by the stored raw pointer _ptr
124     }
125     // overloaded dereferencing operators
126     template <typename T>
127     inline T& StoragePolicy<T>::Array::operator *()
128     {
129         return *_ptr;
130     }
131     template <typename T>
132     inline const T& StoragePolicy<T>::Array::operator *() const
133     {
134         return *_ptr;
135     }
136     // overloaded indexing operators
137     template <typename T>
138     T& StoragePolicy<T>::Array::operator [](int index)
139     {
140         return _ptr[index];
141     }
142     template <typename T>
143     inline const T& StoragePolicy<T>::Array::operator [](int index) const
144     {
145         return _ptr[index];
146     }
147     // overloaded indirection operators
148     template <typename T>
149     inline T* StoragePolicy<T>::Array::operator ->()
150     {
151         return _ptr;
152     }
153     template <typename T>
154     inline const T* StoragePolicy<T>::Array::operator ->() const
155     {
156         return _ptr;
157     }
158     // it will be called during policy exchanges
159     template <typename T>
160     inline void StoragePolicy<T>::Array::_swap(Array& policy)
161     {
162         T* aux(_ptr);
163         _ptr = policy._ptr;
164         policy._ptr = aux;
165     }
166     // deallocation of the array referenced by the raw pointer _ptr
167     template <typename T>
168     inline void StoragePolicy<T>::Array::_destroy()
169     {
170         if (_ptr)
171         {
172             delete[] _ptr, _ptr = nullptr;
173         }
174     }
175 }
176 #endif // STORAGEPOLICIES_H
```


© Ágoston Róth, 2018

2.2.6 Checking policies

The user can choose from the “reject null dereference or indirection”, “reject null”, “assert null dereference or indirection” and “assert null” checking policies. The last two of them are based on the usage of the macro assert, they are invoked only under debug mode and will terminate the execution of the program. The possible checking policies are illustrated in Fig. 2.5/27, while their implementations can be found in Listing 2.8/27.



Fig. 2.5: Possible checking policies ensured by our smart pointers

Listing 2.8. Checking policies (**Core/SmartPointers/CheckingPolicies.h**)

```
1  #ifndef CHECKINGPOLICIES_H
2  #define CHECKINGPOLICIES_H

3  #include <cassert>
4  #include <stdexcept>

5  namespace cagd
6  {
7      // The helper class NullPointerException is used by the RejectNullDereferenceOrIndirection and RejectNull
8      // checking policies. (Each instantiation attempt will generate a run-time error.)
9      class NullPointerException: public std::runtime_error
10     {
11     public:
12         NullPointerException() throw(): std::runtime_error("Null pointer exception!") {}
13     };

14     // The user can choose from the:
15     //     - RejectNullDereferenceOrIndirection;
16     //     - RejectNull;
17     //     - AssertNullDereferenceOrIndirection; and
18     //     - AssertNull
19     // checking policies.
20     template <typename T>
21     class CheckingPolicy
22     {
23     public:
24         // The NoCheck policy will not perform checkings during initialization, dereference and indirection.
25         class NoCheck
```


© Ágoston Róth, 2018

```
26          {
27          public:
28              static void onInitialize(const T* ptr) {}
29              static void onDereferenceOrIndirection(const T* ptr) {}
30          };

31          // The RejectNullDereferenceOrIndirection checking policy will throw a NullPointerException whenever
32          // one attempts either to dereference or indirect a smart pointer that stores a null raw pointer.
33          class RejectNullDereferenceOrIndirection
34          {
35          public:
36              static void onInitialize(const T* /*ptr*/) {}

37              static void onDereferenceOrIndirection(const T* ptr)
38              {
39                  if (!ptr)
40                  {
41                      throw NullPointerException();
42                  }
43              }
44          };

45          // The RejectNull checking policy will throw a NullPointerException whenever one bumps into null raw
46          // pointers during the initialization/assignment, dereference and indirection of smart pointers.
47          class RejectNull
48          {
49          public:
50              static void onInitialize(const T* ptr)
51              {
52                  if (!ptr)
53                  {
54                      throw NullPointerException();
55                  }
56              }

57              static void onDereferenceOrIndirection(const T* ptr)
58              {
59                  onInitialize(ptr);
60              }
61          };

62          // Before dereferencing and indirection, the AssertOnDereferenceOrIndirection checking policy verifies in
63          // debug mode the value of the stored raw pointer _ptr.
64          class AssertNullDereferenceOrIndirection
65          {
66          public:
67              static void onInitialize(const T* /*ptr*/) {}

68              static void onDereferenceOrIndirection(const T* ptr)
69              {
70                  assert("Null pointer dereference or indirection!" && ptr);
71              }
72          };

73          // The AssertNull checking policy verifies in debug mode the value of the stored raw pointer _ptr
74          // before every attempt of initialization/assignment, dereferencing and indirection.
75          class AssertNull
76          {
77          public:
78              static void onInitialize(const T* ptr)
79              {
80                  assert("Initializing with null pointer is prohibited!" && ptr);
81              }

82              static void onDereferenceOrIndirection(const T* ptr)
83              {
84                  assert("Null pointer dereference or indirection!" && ptr);
85              }
86          };
87      };
88  }

89  #endif // CHECKINGPOLICIES_H
```


© Ágoston Róth, 2018

2.2.7 Smart pointers

Using the implicit conversion, ownership, storage and checking policies as base classes, we define generic smart pointers by multiple inheritance as it is detailed in Listing 2.9/29.

Listing 2.9. Smart pointers (**Core/SmartPointers/SmartPointers.h**)

```
#ifndef SMARTPOINTERS_H
#define SMARTPOINTERS_H

#include "CheckingPolicies.h"
#include "ImplicitConversionPolicies.h"
#include "OwnershipPolicies.h"
#include "StoragePolicies.h"
#include "TypeSelectors.h"

namespace cagd
{
    template
    <
        typename T,
        class    TSP  = typename StoragePolicy<T>::Default,     // default storage policy
        class    TOP  = typename OwnershipPolicy<T>::DeepCopy,  // default ownership policy
        class    TICP = ImplicitConversionPolicy::Disallowed,   // default implicit conversion policy
        class    TCP  = typename CheckingPolicy<T>::NoCheck     // default checking policy
    >
    class SmartPointer: public TSP, public TOP, public TICP, public TCP
    {
    private:
        // smart pointers with different typename parameters have to be friends, otherwise we cannot perform
        // operations through the corresponding private member _ptr
        template <typename U, class USP, class UOP, class UICP, class UCP>
        friend class SmartPointer;

    // Handling the implicit conversion policy:
    private:
        // represents a private helper class for disallowing implicit conversion
        class _Disallowed
        {
        };

        // if the implicit conversion is enabled, the _ImplicitConversionType below will be defined as T*,
        // otherwise as _Disallowed
        typedef typename TypeSelector<TICP::ENABLED, T*, _Disallowed>::Result
                         _ImplicitConversionType;

    public:
        // if the _ImplicitConversionType type above coincides with _Disallowed, then the type conversion operator
        // below will generate a compile-time error, since there exists no conversion that transforms the type T*
        // to _Disallowed
        operator _ImplicitConversionType() const
        {
            return TSP::_ptr;
        }

    // Handling direct null pointer testing:
    private:
        // represents a private helper class for direct null pointer testing
        class NullPointerTester
        {
        private:
            void operator delete(void*); // it cannot be called
        };

    public:
        operator NullPointerTester*() const
        {
            static class NullPointerTester test;

            if (!TSP::_ptr)
            {
                return nullptr;
```


© Ágoston Róth, 2018

```
58              }

59              return &test;
60          }

61      // Handling the selected ownership policy:
62      private:
63          typedef typename
64                  TypeSelector< TOP::DESTRUCTIVE_COPY,
65                                SmartPointer<T, TSP, TOP, TICP, TCP>,
66                                const SmartPointer<T, TSP, TOP, TICP, TCP> >::Result
67                  _CopiedType;

68          void _swap(SmartPointer& sp);

69      // Declaration of special/default/copy constructors, of overloaded operators and of the destructor:
70      public:
71          // explicit special/default constructor
72          explicit SmartPointer(T* ptr = nullptr);

73          // in order to enable assignments like SmartPointer<const T> = SmartPointer<T>,
74          // we have to enable the conversion from SmartPointer<T> to SmartPointer<const T>
75          operator SmartPointer<const T, TSP, TOP, TICP, TCP>() const;

76          // copy constructor
77          SmartPointer(_CopiedType& sp);

78          // overloaded assignment operator
79          SmartPointer<T, TSP, TOP, TICP, TCP>& operator =(_CopiedType& rhs);

80          // overloaded dereferencing operators
81          T& operator *();
82          const T& operator *() const;

83          // overloaded indirection operators
84          T* operator ->();
85          const T* operator ->() const;

86          // overloaded member comparison operators
87          bool operator ==(const SmartPointer<T, TSP, TOP, TICP, TCP>& rhs) const;
88          bool operator !=(const SmartPointer<T, TSP, TOP, TICP, TCP>& rhs) const;
89          bool operator !() const;

90          template <typename U, class USP, class UOP, class UICP, class UCP>
91          bool operator ==(const SmartPointer<U, USP, UOP, UICP, UCP>& rhs) const;

92          template <typename U, class USP, class UOP, class UICP, class UCP>
93          bool operator !=(const SmartPointer<U, USP, UOP, UICP, UCP>& rhs) const;

94          // overloaded friend comparison operators
95          template <typename U, typename V, class VSP, class VOP, class VICP, class VCP>
96          friend bool operator ==(const U* lhs,
97                                  const SmartPointer<V, VSP, VOP, VICP, VCP>& rhs);

98          template <typename U, class USP, class UOP, class UICP, class UCP, typename V>
99          friend bool operator ==(const SmartPointer<U, USP, UOP, UICP, UCP>& lhs,
100                                 const V* rhs);

101         template <typename U, typename V, class VSP, class VOP, class VICP, class VCP>
102         friend bool operator !=(const U* lhs,
103                                 const SmartPointer<V, VSP, VOP, VICP, VCP>& rhs);

104         template <typename U, class USP, class UOP, class UICP, class UCP, typename V>
105         friend bool operator !=(const SmartPointer<U, USP, UOP, UICP, UCP>& lhs,
106                                 const V* rhs);

107         // destructor
108         ~SmartPointer();
109     };

110     // special/default constructor
111     template <typename T, class TSP, class TOP, class TICP, class TCP>
112     inline SmartPointer<T, TSP, TOP, TICP, TCP>::SmartPointer(T* ptr):
113             TSP(ptr), TOP(), TICP(), TCP()
```


© Ágoston Róth, 2018

```
114     {
115         TCP::onInitialize(ptr);
116     }
117     // in order to enable assignments like SmartPointer<const T> = SmartPointer<T>,
118     // we have to enable the conversion from SmartPointer<T> to SmartPointer<const T>
119     template <typename T, class TSP, class TOP, class TICP, class TCP>
120     inline SmartPointer<T, TSP, TOP, TICP, TCP>::operator
121            SmartPointer<const T, TSP, TOP, TICP, TCP>() const
122     {
123         return SmartPointer<const T, TSP, TOP, TICP, TCP>(TOP::clone(TSP::_ptr));
124     }
125     // copy constructor
126     template <typename T, class TSP, class TOP, class TICP, class TCP>
127     inline SmartPointer<T, TSP, TOP, TICP, TCP>::SmartPointer(_CopiedType& sp):
128            TSP(sp), TOP(sp), TICP(), TCP()
129     {
130         TSP::_ptr = TOP::_clone(sp._ptr);
131     }
132     // exchanging storage and ownership policies
133     template <typename T, class TSP, class TOP, class TICP, class TCP>
134     inline void SmartPointer<T, TSP, TOP, TICP, TCP>::_swap(
135            SmartPointer<T, TSP, TOP, TICP, TCP>& sp)
136     {
137         TSP::_swap(sp);
138         TOP::_swap(sp);
139     }
140     // overloaded assignment operator
141     template <typename T, class TSP, class TOP, class TICP, class TCP>
142     inline SmartPointer<T, TSP, TOP, TICP, TCP>&
143            SmartPointer<T, TSP, TOP, TICP, TCP>::operator =(_CopiedType& rhs)
144     {
145         if (this != &rhs)
146         {
147             if (TOP::_release(TSP::_ptr))
148             {
149                 TSP::_destroy();
150             }
151             SmartPointer aux(rhs);
152             aux._swap(*this);
153         }
154         return *this;
155     }
156     // overloaded dereferencing operators
157     template <typename T, class TSP, class TOP, class TICP, class TCP>
158     inline T& SmartPointer<T, TSP, TOP, TICP, TCP>::operator *()
159     {
160         TCP::onDereferenceOrIndirection(TSP::_ptr);
161         return TSP::operator *();
162     }
163     template <typename T, class TSP, class TOP, class TICP, class TCP>
164     inline const T& SmartPointer<T, TSP, TOP, TICP, TCP>::operator *() const
165     {
166         TCP::onDereferenceOrIndirection(TSP::_ptr);
167         return TSP::operator *();
168     }
169     // overloaded indirection operators
170     template <typename T, class TSP, class TOP, class TICP, class TCP>
171     inline T* SmartPointer<T, TSP, TOP, TICP, TCP>::operator ->()
172     {
173         TCP::onDereferenceOrIndirection(TSP::_ptr);
174         return TSP::operator ->();
175     }
176     template <typename T, class TSP, class TOP, class TICP, class TCP>
177     inline const T* SmartPointer<T, TSP, TOP, TICP, TCP>::operator ->() const
```


© Ágoston Róth, 2018

```
178     {
179         TCP::onDereferenceOrIndirection(TSP::_ptr);
180         return TSP::operator ->();
181     }
182     // overloaded member comparison operators
183     template <typename T, class TSP, class TOP, class TICP, class TCP>
184     inline bool SmartPointer<T, TSP, TOP, TICP, TCP>::operator ==(
185             const SmartPointer<T, TSP, TOP, TICP, TCP>& rhs) const
186     {
187         return (TSP::_ptr == rhs._ptr);
188     }
189     template <typename T, class TSP, class TOP, class TICP, class TCP>
190     inline bool SmartPointer<T, TSP, TOP, TICP, TCP>::operator !=(
191             const SmartPointer<T, TSP, TOP, TICP, TCP>& rhs) const
192     {
193         return (TSP::_ptr != rhs._ptr);
194     }
195     template <typename T, class TSP, class TOP, class TICP, class TCP>
196     inline bool SmartPointer<T, TSP, TOP, TICP, TCP>::operator !() const
197     {
198         return !TSP::_ptr;
199     }
200     template <typename T, class TSP, class TOP, class TICP, class TCP>
201     template <typename U, class USP, class UOP, class UICP, class UCP>
202     inline bool SmartPointer<T, TSP, TOP, TICP, TCP>::operator ==(
203             const SmartPointer<U, USP, UOP, UICP, UCP>& rhs) const
204     {
205         return ((void*)TSP::_ptr == (void*)rhs._ptr);
206     }
207     template <typename T, class TSP, class TOP, class TICP, class TCP>
208     template <typename U, class USP, class UOP, class UICP, class UCP>
209     inline bool SmartPointer<T, TSP, TOP, TICP, TCP>::operator !=(
210             const SmartPointer<U, USP, UOP, UICP, UCP>& rhs) const
211     {
212         return ((void*)TSP::_ptr != (void*)rhs._ptr);
213     }
214     // overloaded friend comparison operators
215     template <typename U, typename V, class VSP, class VOP, class VICP, class VCP>
216     inline bool operator ==(
217             const U* lhs, const SmartPointer<V, VSP, VOP, VICP, VCP>& rhs)
218     {
219         return ((void*)lhs == (void*)rhs._ptr);
220     }
221     template <typename U, class USP, class UOP, class UICP, class UCP, typename V>
222     inline bool operator ==(
223             const SmartPointer<U, USP, UOP, UICP, UCP>& lhs, const V* rhs)
224     {
225         return ((void*)lhs._ptr == (void*)rhs);
226     }
227     template <typename U, typename V, class VSP, class VOP, class VICP, class VCP>
228     inline bool operator !=(
229             const U* lhs, const SmartPointer<V, VSP, VOP, VICP, VCP>& rhs)
230     {
231         return ((void*)lhs != (void*)rhs._ptr);
232     }
233     template <typename U, class USP, class UOP, class UICP, class UCP, typename V>
234     inline bool operator !=(
235             const SmartPointer<U, USP, UOP, UICP, UCP>& lhs, const V* rhs)
236     {
237         return ((void*)lhs._ptr != (void*)rhs);
238     }
239     // destructor
240     template <typename T, class TSP, class TOP, class TICP, class TCP>
241     inline SmartPointer<T, TSP, TOP, TICP, TCP>::~SmartPointer()
```


© Ágoston Róth, 2018

```
242     {
243         if (TOP::_release(TSP::_ptr))
244         {
245             TSP::_destroy();
246         }
247     }
248 }

249 #endif // SMARTPOINTERS_H
```

2.2.8 Frequently used specialized smart pointers

Listing 2.10/33 provides simpler type definitions for the most frequently used smart pointers. For example, the specialized smart pointer SP<T>::Default ensures default storage and deep copy policies, disallows implicit conversion and rejects null dereference or indirection.

Listing 2.10. Specialized smart pointers (**Core/SmartPointers/SpecializedSmartPointers.h**)

```
1  #ifndef SPECIALIZEDSMARTPOINTERS_H
2  #define SPECIALIZEDSMARTPOINTERS_H

3  #include "SmartPointers.h"

4  namespace cagd
5  {
6      template <typename T>
7      struct SP
8      {
9          typedef
10         SmartPointer
11         <
12             T,
13             typename StoragePolicy<T>::Default,
14             typename OwnershipPolicy<T>::DeepPrimitiveCopy,
15             ImplicitConversionPolicy::Disallowed,
16             typename CheckingPolicy<T>::RejectNullDereferenceOrIndirection
17         >
18         DefaultPrimitive;

19         typedef
20         SmartPointer
21         <
22             T,
23             typename StoragePolicy<T>::Default,
24             typename OwnershipPolicy<T>::DeepCopy,
25             ImplicitConversionPolicy::Disallowed,
26             typename CheckingPolicy<T>::RejectNullDereferenceOrIndirection
27         >
28         Default;

29         typedef
30         SmartPointer
31         <
32             T,
33             typename StoragePolicy<T>::Array,
34             typename OwnershipPolicy<T>::NoCopy,
35             ImplicitConversionPolicy::Disallowed,
36             typename CheckingPolicy<T>::RejectNullDereferenceOrIndirection
37         >
38         Array;

39         typedef
40         SmartPointer
41         <
42             T,
43             typename StoragePolicy<T>::Default,
44             typename OwnershipPolicy<T>::DestructiveCopy,
45             ImplicitConversionPolicy::Disallowed,
46             typename CheckingPolicy<T>::RejectNullDereferenceOrIndirection
47         >
```


© Ágoston Róth, 2018

```
48          DestructiveCopy;

49          typedef
50          SmartPointer
51          <
52              T,
53              typename StoragePolicy<T>::Default,
54              typename OwnershipPolicy<T>::NonIntrusiveReferenceCounting,
55              ImplicitConversionPolicy::Disallowed,
56              typename CheckingPolicy<T>::RejectNullDereferenceOrIndirection
57          >
58          NonIntrusiveReferenceCounting;
59      };
60  }

61  #endif // SPECIALIZEDSMARTPOINTERS_H
```

2.3 Cartesian coordinates

In order to implement mathematical formulas related to B-curves/surfaces in a more natural way, we also provide a class for 3-dimensional Cartesian coordinates (Cartesian3) that ensures several useful overloaded arithmetical, logical, indexing and output/input stream operators. By means of these coordinates one can describe and store curve/surface points, higher order (mixed partial) derivatives, (unit) normal vectors associated with surface points, data that have to be interpolated, vertices of control polygons, of nets and of triangle meshes. Thus, these coordinates form the most elementary building blocks for the majority of the forthcoming classes. Fig. 2.6/34 illustrates its diagram, while Listing 2.11/35 provides its implementation.

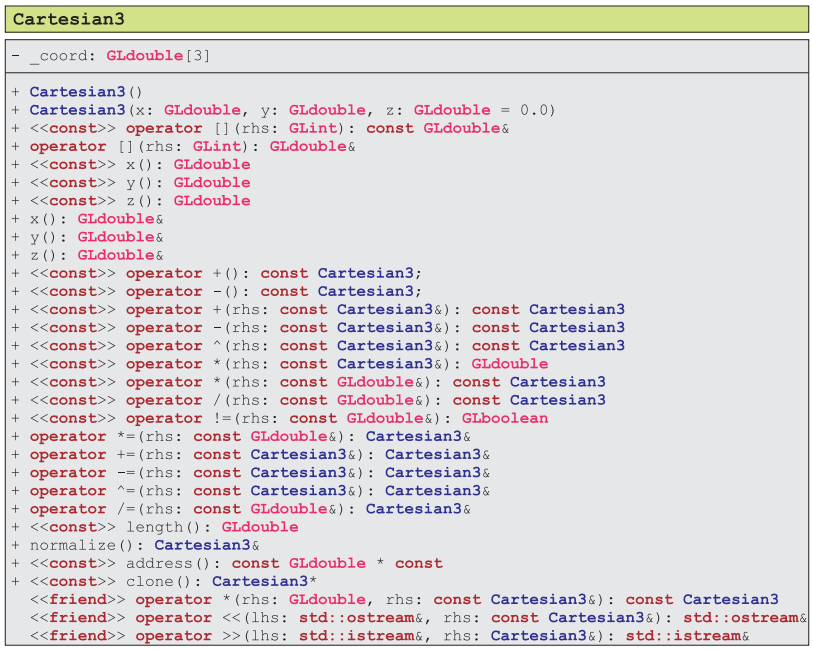


Fig. 2.6: Class diagram of 3-dimensional Cartesian coordinates

© Ágoston Róth, 2018

Listing 2.11. 3-dimensional Cartesian coordinates (**Core/Geometry/Coordinates/Cartesians3.h**)

```
#ifndef CARTESIANS3_H
#define CARTESIANS3_H

#include <GL/glew.h>

#include <cassert>
#include <cmath>
#include <iostream>

namespace cagd
{
    class Cartesian3
    {
    private:
        // coordinates
        GLdouble _coord[3];

    public:
        // default constructor
        Cartesian3();

        // special constructor
        Cartesian3(const GLdouble &x, const GLdouble &y, const GLdouble &z = 0.0);

        // get components by constant references
        const GLdouble& operator [](const GLint &rhs) const;
        const GLdouble& x() const;
        const GLdouble& y() const;
        const GLdouble& z() const;

        // get components by non-constant references
        GLdouble& operator [](const GLint &rhs);
        GLdouble& x();
        GLdouble& y();
        GLdouble& z();

        // sign changing unary operators
        const Cartesian3 operator +() const;
        const Cartesian3 operator -() const;

        // addition
        const Cartesian3 operator +(const Cartesian3 &rhs) const;

        // add to *this
        Cartesian3& operator +=(const Cartesian3 &rhs);

        // subtraction
        const Cartesian3 operator -(const Cartesian3 &rhs) const;

        // subtract from *this
        Cartesian3& operator -=(const Cartesian3 &rhs);

        // cross product v1 (x1, y1, z1) × v2 (x2, y2, z2) = det | i  j  k  |
        //                                                       | x1 y1 z1 |
        //                                                       | x2 y2 z2 |
        const Cartesian3 operator ^(const Cartesian3 &rhs) const;

        // cross product, result is stored by *this
        Cartesian3& operator ^=(const Cartesian3 &rhs);

        // dot product
        GLdouble operator *(const Cartesian3 &rhs) const;

        // scale from right
        const Cartesian3 operator *(const GLdouble &rhs) const;
        const Cartesian3 operator /(const GLdouble &rhs) const;

        // scale *this
        Cartesian3& operator *=(const GLdouble &rhs);
        Cartesian3& operator /=(const GLdouble &rhs);
```


© Ágoston Róth, 2018

```
52          // returns the Euclidean norm of the stored vector
53          GLdouble length() const;
54          // normalizes the stored vector
55          Cartesian3& normalize();
56          // get constant pointer to constant data
57          const GLdouble* address() const;
58          // clone function required by smart pointers based on the deep copy ownership policy
59          Cartesian3* clone() const;
60          // logical operator
61          GLboolean operator != (const GLdouble &rhs) const;
62      };
63      // default constructor
64      inline Cartesian3::Cartesian3()
65      {
66          _coord[0] = _coord[1] = _coord[2] = 0.0;
67      }
68      // special constructor
69      inline Cartesian3::Cartesian3(const GLdouble &x, const GLdouble &y, const GLdouble &z)
70      {
71          _coord[0] = x;
72          _coord[1] = y;
73          _coord[2] = z;
74      }
75      // get components by constant references
76      inline const GLdouble& Cartesian3::operator [](const GLint &rhs) const
77      {
78          assert("Cartesian coordinate index is out of bounds!" && (rhs >= 0 && rhs < 3));
79          return _coord[rhs];
80      }
81      inline const GLdouble& Cartesian3::x() const
82      {
83          return _coord[0];
84      }
85      inline const GLdouble& Cartesian3::y() const
86      {
87          return _coord[1];
88      }
89      inline const GLdouble& Cartesian3::z() const
90      {
91          return _coord[2];
92      }
93      // get components by non-constant references
94      inline GLdouble& Cartesian3::operator [](const GLint &rhs)
95      {
96          assert("Cartesian coordinate index is out of bounds!" && (rhs >= 0 && rhs < 3));
97          return _coord[rhs];
98      }
99      inline GLdouble& Cartesian3::x()
100     {
101         return _coord[0];
102     }
103     inline GLdouble& Cartesian3::y()
104     {
105         return _coord[1];
106     }
107     inline GLdouble& Cartesian3::z()
108     {
109         return _coord[2];
110     }
```


© Ágoston Róth, 2018

```
111     // sign changing unary operators
112     inline const Cartesian3 Cartesian3::operator +() const
113     {
114         return Cartesian3(_coord[0], _coord[1], _coord[2]);
115     }
116     inline const Cartesian3 Cartesian3::operator -() const
117     {
118         return Cartesian3(-_coord[0], -_coord[1], -_coord[2]);
119     }
120     // addition
121     inline const Cartesian3 Cartesian3::operator +(const Cartesian3 &rhs) const
122     {
123         return Cartesian3(_coord[0] + rhs._coord[0],
124                           _coord[1] + rhs._coord[1],
125                           _coord[2] + rhs._coord[2]);
126     }
127     // add to *this
128     inline Cartesian3& Cartesian3::operator +=(const Cartesian3 &rhs)
129     {
130         _coord[0] += rhs._coord[0];
131         _coord[1] += rhs._coord[1];
132         _coord[2] += rhs._coord[2];
133         return *this;
134     }
135     // subtraction
136     inline const Cartesian3 Cartesian3::operator -(const Cartesian3 &rhs) const
137     {
138         return Cartesian3(_coord[0] - rhs._coord[0],
139                           _coord[1] - rhs._coord[1],
140                           _coord[2] - rhs._coord[2]);
141     }
142     // subtract from *this
143     inline Cartesian3& Cartesian3::operator -=(const Cartesian3 &rhs)
144     {
145         _coord[0] -= rhs._coord[0];
146         _coord[1] -= rhs._coord[1];
147         _coord[2] -= rhs._coord[2];
148         return *this;
149     }
150     // cross product
151     inline const Cartesian3 Cartesian3::operator ^(const Cartesian3 &rhs) const
152     {
153         return Cartesian3(_coord[1] * rhs._coord[2] - _coord[2] * rhs._coord[1],
154                           _coord[2] * rhs._coord[0] - _coord[0] * rhs._coord[2],
155                           _coord[0] * rhs._coord[1] - _coord[1] * rhs._coord[0]);
156     }
157     // cross product, result is stored by *this
158     inline Cartesian3& Cartesian3::operator ^=(const Cartesian3 &rhs)
159     {
160         GLdouble x = _coord[1] * rhs._coord[2] - _coord[2] * rhs._coord[1],
161                  y = _coord[2] * rhs._coord[0] - _coord[0] * rhs._coord[2],
162                  z = _coord[0] * rhs._coord[1] - _coord[1] * rhs._coord[0];
163         _coord[0] = x;
164         _coord[1] = y;
165         _coord[2] = z;
166         return *this;
167     }
168     // dot product
169     inline GLdouble Cartesian3::operator *(const Cartesian3 &rhs) const
170     {
171         return _coord[0] * rhs._coord[0] +
172                _coord[1] * rhs._coord[1] +
```


© Ágoston Róth, 2018

```
173                 _coord[2] * rhs._coord[2];
174     }
175     // scale from right
176     inline const Cartesian3 Cartesian3::operator *(const GLdouble &rhs) const
177     {
178         return Cartesian3(_coord[0] * rhs, _coord[1] * rhs, _coord[2] * rhs);
179     }
180     inline const Cartesian3 Cartesian3::operator /(const GLdouble &rhs) const
181     {
182         return Cartesian3(_coord[0] / rhs, _coord[1] / rhs, _coord[2] / rhs);
183     }
184     // scale from left
185     inline const Cartesian3 operator *(const GLdouble& lhs, const Cartesian3 &rhs)
186     {
187         return Cartesian3(lhs * rhs[0], lhs * rhs[1], lhs * rhs[2]);
188     }
189     // scale *this
190     inline Cartesian3& Cartesian3::operator *=(const GLdouble &rhs)
191     {
192         _coord[0] *= rhs;
193         _coord[1] *= rhs;
194         _coord[2] *= rhs;
195         return *this;
196     }
197     inline Cartesian3& Cartesian3::operator /=(const GLdouble &rhs)
198     {
199         _coord[0] /= rhs;
200         _coord[1] /= rhs;
201         _coord[2] /= rhs;
202         return *this;
203     }
204     // returns the Euclidean norm of the stored vector
205     inline GLdouble Cartesian3::length() const
206     {
207         return std::sqrt((*this) * (*this));
208     }
209     // normalizes the stored vector
210     inline Cartesian3& Cartesian3::normalize()
211     {
212         GLdouble l = length();
213         if (l && l != 1.0)
214         {
215             *this /= l;
216         }
217         return *this;
218     }
219     // get constant pointer to constant data
220     inline const GLdouble* Cartesian3::address() const
221     {
222         return _coord;
223     }
224     // clone function required by smart pointers based on the deep copy ownership policy
225     inline Cartesian3* Cartesian3::clone() const
226     {
227         return new (std::nothrow) Cartesian3(_coord[0], _coord[1], _coord[2]);
228     }
229     // logical operator
230     inline GLboolean Cartesian3::operator != (const GLdouble &rhs) const
231     {
232         return (_coord[0] != rhs || _coord[1] != rhs || _coord[2] != rhs);
```


© Ágoston Róth, 2018

```
233     }
234     // output to stream
235     inline std::ostream& operator <<(std::ostream& lhs, const Cartesian3 &rhs)
236     {
237         return lhs << rhs[0] << " " << rhs[1] << " " << rhs[2];
238     }
239     // input from stream
240     inline std::istream& operator >>(std::istream& lhs, Cartesian3 &rhs)
241     {
242         return lhs >> rhs[0] >> rhs[1] >> rhs[2];
243     }
244 }
245 #endif // CARTESIANS3_H
```

2.4 Homogeneous coordinates

Cartesian coordinates are not able to describe points at infinity. In order to overcome this problem, we will use homogeneous coordinates which are used in projective geometry and they have the advantage that the coordinates of points, including points at infinity, can be represented using finite coordinates. Formulas based on homogeneous coordinates are often simpler and more symmetric then their Cartesian counterparts. Moreover, they allow affine transformations and, in general, projective transformations to be easily represented by matrices.

Fig. 2.7/39 illustrates the diagram, while Listing 2.12/39 provides the implementation of the class Homogeneous3, i.e., of 3-dimensional homogeneous coordinates. Note, that we will mostly use this class in case of coordinate and point transformations or to define light positions and directions. Since these operations are graphics related, it is sufficiently enough to store the homogeneous coordinates in an array of single-precision floating-point values.

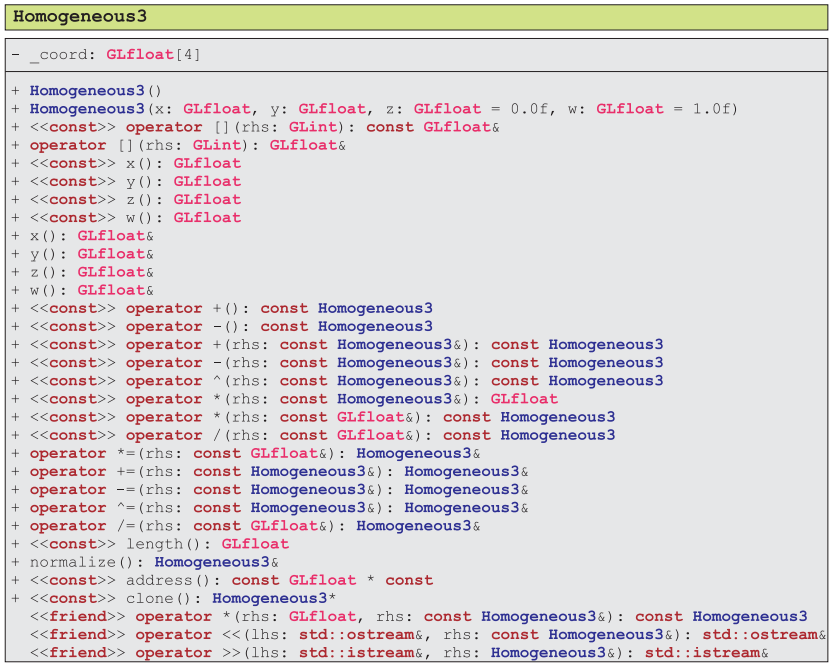


Fig. 2.7: Class diagram of 3-dimensional homogeneous coordinates

© Ágoston Róth, 2018

Listing 2.12. 3-dimensional homogeneous coordinates (**Core/Geometry/Coordinates/Homogeneous3.h**)

```
1  #ifndef HOMOGENEOUS3_H
2  #define HOMOGENEOUS3_H

3  #include <GL/glew.h>

4  #include "Cartesians3.h"

5  #include <cassert>
6  #include <cmath>
7  #include <iostream>
8  #include <limits>
9  #include <new>

10 namespace cagd
11 {
12     class Homogeneous3
13     {
14     private:
15         // coordinates
16         GLfloat _coord[4];

17     public:
18         // default constructor
19         Homogeneous3();

20         // special constructor
21         Homogeneous3(GLfloat x, GLfloat y, GLfloat z = 0.0, GLfloat w = 1.0);

22         // special constructor
23         explicit Homogeneous3(const Cartesian3 &c);

24         // get components by constant references
25         const GLfloat& operator [](const GLint &rhs) const;
26         const GLfloat& x() const;
27         const GLfloat& y() const;
28         const GLfloat& z() const;
29         const GLfloat& w() const;

30         // get components by non-constant references
31         GLfloat& operator [](const GLint &rhs);
32         GLfloat& x();
33         GLfloat& y();
34         GLfloat& z();
35         GLfloat& w();

36         // sign changing unary operators
37         const Homogeneous3 operator +() const;
38         const Homogeneous3 operator -() const;

39         // addition
40         const Homogeneous3 operator +(const Homogeneous3& rhs) const;

41         // adds to *this
42         Homogeneous3& operator +=(const Homogeneous3& rhs);

43         // subtraction
44         const Homogeneous3 operator -(const Homogeneous3& rhs) const;

45         // subtracts from *this
46         Homogeneous3& operator -=(const Homogeneous3& rhs);

47         // cross product
48         const Homogeneous3 operator ^(const Homogeneous3& rhs) const;

49         // cross product, result is stored by *this
50         Homogeneous3& operator ^=(const Homogeneous3& rhs);

51         // dot product
52         GLfloat operator *(const Homogeneous3& rhs) const;

53         // scale from right
```


© Ágoston Róth, 2018

```
54          const Homogeneous3 operator *(GLfloat rhs) const;
55          const Homogeneous3 operator /(GLfloat rhs) const;

56          // scale *this
57          Homogeneous3& operator *=(GLfloat rhs);
58          Homogeneous3& operator /=(GLfloat rhs);

59          // returns the Euclidean norm of the represented Cartesian coordinate
60          GLfloat length() const;

61          // normalizes the represented Cartesian coordinate
62          Homogeneous3& normalize();

63          // get constant pointer to constant data
64          const GLfloat * address() const;

65          // clone function required by smart pointers based on the deep copy ownership policy
66          Homogeneous3* clone() const;
67      };

68      // default constructor
69      inline Homogeneous3::Homogeneous3()
70      {
71          _coord[0] = _coord[1] = _coord[2] = 0.0;
72          _coord[3] = 1.0;
73      }

74      // special constructor
75      inline Homogeneous3::Homogeneous3(GLfloat x, GLfloat y, GLfloat z, GLfloat w)
76      {
77          _coord[0] = x;
78          _coord[1] = y;
79          _coord[2] = z;
80          _coord[3] = w;
81      }

82      // special constructor
83      inline Homogeneous3::Homogeneous3(const Cartesian3 &c)
84      {
85          _coord[0] = (GLfloat)c[0];
86          _coord[1] = (GLfloat)c[1];
87          _coord[2] = (GLfloat)c[2];
88          _coord[3] = 1.0f;
89      }

90      // get components by constant references
91      inline const GLfloat& Homogeneous3::operator [](const GLint &rhs) const
92      {
93          assert("Homogeneous coordinate index is out of bounds!" && (rhs >= 0 && rhs < 4));
94          return _coord[rhs];
95      }

96      inline const GLfloat& Homogeneous3::x() const
97      {
98          return _coord[0];
99      }

100     inline const GLfloat& Homogeneous3::y() const
101     {
102         return _coord[1];
103     }

104     inline const GLfloat& Homogeneous3::z() const
105     {
106         return _coord[2];
107     }

108     inline const GLfloat& Homogeneous3::w() const
109     {
110         return _coord[3];
111     }

112     // get components by non-constant references
113     inline GLfloat& Homogeneous3::operator [](const GLint &rhs)
```


© Ágoston Róth, 2018

```
114     {
115         assert("Homogeneous coordinate index is out of bounds!" && (rhs >= 0 && rhs < 4));
116         return _coord[rhs];
117     }
118     inline GLfloat& Homogeneous3::x()
119     {
120         return _coord[0];
121     }
122     inline GLfloat& Homogeneous3::y()
123     {
124         return _coord[1];
125     }
126     inline GLfloat& Homogeneous3::z()
127     {
128         return _coord[2];
129     }
130     inline GLfloat& Homogeneous3::w()
131     {
132         return _coord[3];
133     }
134     // sign changing unary operators
135     inline const Homogeneous3 Homogeneous3::operator +() const
136     {
137         return Homogeneous3(_coord[0], _coord[1], _coord[2], _coord[3]);
138     }
139     inline const Homogeneous3 Homogeneous3::operator -() const
140     {
141         return Homogeneous3(-_coord[0], -_coord[1], -_coord[2], _coord[3]);
142     }
143     // addition
144     inline const Homogeneous3 Homogeneous3::operator +(const Homogeneous3& rhs) const
145     {
146         return Homogeneous3(rhs._coord[3] * _coord[0] + _coord[3] * rhs._coord[0],
147                             rhs._coord[3] * _coord[1] + _coord[3] * rhs._coord[1],
148                             rhs._coord[3] * _coord[2] + _coord[3] * rhs._coord[2],
149                             _coord[3] * rhs._coord[3]);
150     }
151     // add to *this
152     inline Homogeneous3& Homogeneous3::operator +=(const Homogeneous3& rhs)
153     {
154         _coord[0] = rhs._coord[3] * _coord[0] + _coord[3] * rhs._coord[0];
155         _coord[1] = rhs._coord[3] * _coord[1] + _coord[3] * rhs._coord[1];
156         _coord[2] = rhs._coord[3] * _coord[2] + _coord[3] * rhs._coord[2];
157         _coord[3] = _coord[3] * rhs._coord[3];
158         return *this;
159     }
160     // subtraction
161     inline const Homogeneous3 Homogeneous3::operator -(const Homogeneous3& rhs) const
162     {
163         return Homogeneous3(rhs._coord[3] * _coord[0] - _coord[3] * rhs._coord[0],
164                             rhs._coord[3] * _coord[1] - _coord[3] * rhs._coord[1],
165                             rhs._coord[3] * _coord[2] - _coord[3] * rhs._coord[2],
166                             _coord[3] * rhs._coord[3]);
167     }
168     // subtract from *this
169     inline Homogeneous3& Homogeneous3::operator -=(const Homogeneous3& rhs)
170     {
171         _coord[0] = rhs._coord[3] * _coord[0] - _coord[3] * rhs._coord[0];
172         _coord[1] = rhs._coord[3] * _coord[1] - _coord[3] * rhs._coord[1];
173         _coord[2] = rhs._coord[3] * _coord[2] - _coord[3] * rhs._coord[2];
174         _coord[3] = _coord[3] * rhs._coord[3];
175         return *this;
```


© Ágoston Róth, 2018

```
176     }
177     // cross product
178     inline const Homogeneous3 Homogeneous3::operator ^(const Homogeneous3& rhs) const
179     {
180         return Homogeneous3(_coord[1] * rhs._coord[2] - _coord[2] * rhs._coord[1],
181                             _coord[2] * rhs._coord[0] - _coord[0] * rhs._coord[2],
182                             _coord[0] * rhs._coord[1] - _coord[1] * rhs._coord[0],
183                             _coord[3] * rhs._coord[3]);
184     }
185     // cross product, result is stored by *this
186     inline Homogeneous3& Homogeneous3::operator ^=(const Homogeneous3& rhs)
187     {
188         GLfloat x = _coord[1] * rhs._coord[2] - _coord[2] * rhs._coord[1],
189                 y = _coord[2] * rhs._coord[0] - _coord[0] * rhs._coord[2],
190                 z = _coord[0] * rhs._coord[1] - _coord[1] * rhs._coord[0],
191                 w = _coord[3] * rhs._coord[3];
192         _coord[0] = x;
193         _coord[1] = y;
194         _coord[2] = z;
195         _coord[3] = w;
196         return *this;
197     }
198     // dot product
199     inline GLfloat Homogeneous3::operator *(const Homogeneous3& rhs) const
200     {
201         if (_coord[3] == 0.0f || rhs._coord[3] == 0.0f)
202         {
203             return std::numeric_limits<GLfloat>::max();
204         }
205         return (_coord[0] * rhs._coord[0] +
206                 _coord[1] * rhs._coord[1] +
207                 _coord[2] * rhs._coord[2]) / _coord[3] / rhs._coord[3];
208     }
209     // scale from right
210     inline const Homogeneous3 Homogeneous3::operator *(GLfloat rhs) const
211     {
212         return Homogeneous3(_coord[0] * rhs, _coord[1] * rhs, _coord[2] * rhs, _coord[3]);
213     }
214     // scale from left
215     inline const Homogeneous3 operator *(GLfloat lhs, const Homogeneous3& rhs)
216     {
217         return Homogeneous3(lhs * rhs[0], lhs * rhs[1], lhs * rhs[2], rhs[3]);
218     }
219     // scale from right
220     inline const Homogeneous3 Homogeneous3::operator /(GLfloat rhs) const
221     {
222         return Homogeneous3(_coord[0], _coord[1], _coord[2], _coord[3] * rhs);
223     }
224     // scale *this
225     inline Homogeneous3& Homogeneous3::operator *=(GLfloat rhs)
226     {
227         _coord[0] *= rhs;
228         _coord[1] *= rhs;
229         _coord[2] *= rhs;
230         return *this;
231     }
232     inline Homogeneous3& Homogeneous3::operator /=(GLfloat rhs)
233     {
234         _coord[3] *= rhs;
235         return *this;
236     }
```


© Ágoston Róth, 2018

```
237     // returns the Euclidean norm of the represented Cartesian coordinate
238     inline GLfloat Homogeneous3::length() const
239     {
240         return std::sqrt((*this) * (*this));
241     }

242     // normalize
243     inline Homogeneous3& Homogeneous3::normalize()
244     {
245         GLfloat l = length();

246         if (l && l != 1.0)
247         {
248             *this /= l;
249         }

250         return *this;
251     }

252     // get constant pointer to constant data
253     inline const GLfloat* Homogeneous3::address() const
254     {
255         return _coord;
256     }

257     // clone function required by smart pointers based on the deep copy ownership policy
258     inline Homogeneous3* Homogeneous3::clone() const
259     {
260         return new (std::nothrow) Homogeneous3(_coord[0], _coord[1], _coord[2],
261                                                _coord[3]);
262     }

263     // output to stream
264     inline std::ostream& operator <<(std::ostream& lhs, const Homogeneous3& rhs)
265     {
266         return lhs << rhs[0] << " " << rhs[1] << " " << rhs[2] << " " << rhs[3];
267     }

268     // input from stream
269     inline std::istream& operator >>(std::istream& lhs, Homogeneous3& rhs)
270     {
271         return lhs >> rhs[0] >> rhs[1] >> rhs[2] >> rhs[3];
272     }
273 }

274 #endif // HOMOGENEOUS3_H
```

2.5 Texture coordinates

One can also handle 1-, 2- and 3-dimensional and even projective texture coordinates by using the class TCoordinate4 whose diagram is illustrated in Fig. 2.8/45, while its definition and implementation can be found in Listing 2.13/44.

Listing 2.13. Texture coordinates (**Core/Geometry/Coordinates/TCoordinates4.h**)

```
1  #ifndef TCOORDINATES4_H
2  #define TCOORDINATES4_H

3  #include <GL/glew.h>

4  #include <cassert>
5  #include <iostream>
6  #include <new>

7  namespace cagd
8  {
9      class TCoordinate4
10     {
```


© Ágoston Róth, 2018

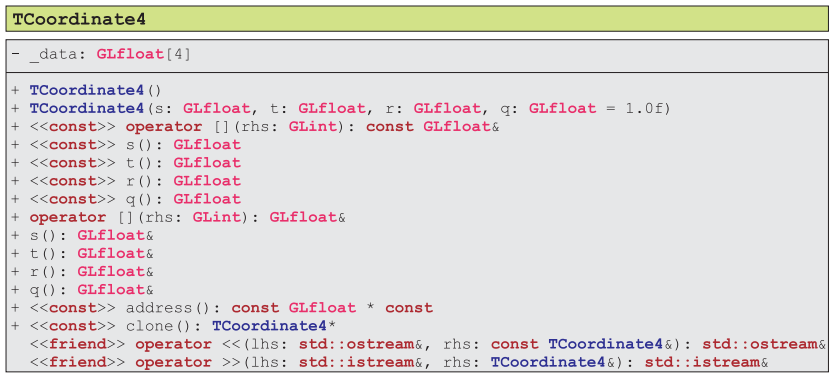


Fig. 2.8: Class diagram of 1-, 2- and 3-dimensional or projective texture coordinates

```
11      private:
12          // four dimensional (projective) texture coordinates (s, t, r, q)
13          GLfloat _data[4];
14      public:
15          // default constructor
16          TCoordinate4();
17          // special constructor
18          TCoordinate4(GLfloat s, GLfloat t, GLfloat r = 0.0, GLfloat q = 1.0);
19          // get components by constant references
20          const GLfloat& operator[](const GLint &rhs) const;
21          const GLfloat& s() const;
22          const GLfloat& t() const;
23          const GLfloat& r() const;
24          const GLfloat& q() const;
25          // get components by non-constant references
26          GLfloat& operator[](const GLint &rhs);
27          GLfloat& s();
28          GLfloat& t();
29          GLfloat& r();
30          GLfloat& q();
31          // get constant pointer to constant data
32          const GLfloat* address() const;
33          // clone function required by smart pointers based on the deep copy ownership policy
34          TCoordinate4* clone() const;
35      };
36      // default constructor
37      inline TCoordinate4::TCoordinate4()
38      {
39          _data[0] = _data[1] = _data[2] = 0.0;
40          _data[3] = 1.0;
41      }
42      // special constructor
43      inline TCoordinate4::TCoordinate4(GLfloat s, GLfloat t, GLfloat r, GLfloat q)
44      {
45          _data[0] = s;
46          _data[1] = t;
47          _data[2] = r;
48          _data[3] = q;
49      }
50      // get components by constant references
51      inline const GLfloat& TCoordinate4::operator [](const GLint &rhs) const
52      {
```


© Ágoston Róth, 2018

```
53          assert("Texture coordinate index is out of bounds!" && (rhs >= 0 && rhs < 4));
54          return _data[rhs];
55      }
56      inline const GLfloat& TCoordinate4::s() const
57      {
58          return _data[0];
59      }
60      inline const GLfloat& TCoordinate4::t() const
61      {
62          return _data[1];
63      }
64      inline const GLfloat& TCoordinate4::r() const
65      {
66          return _data[2];
67      }
68      inline const GLfloat& TCoordinate4::q() const
69      {
70          return _data[3];
71      }
72      // get components by non-constant references
73      inline GLfloat& TCoordinate4::operator [](const GLint &rhs)
74      {
75          assert("Texture coordinate index is out of bounds!" && (rhs >= 0 && rhs < 4));
76          return _data[rhs];
77      }
78      inline GLfloat& TCoordinate4::s()
79      {
80          return _data[0];
81      }
82      inline GLfloat& TCoordinate4::t()
83      {
84          return _data[1];
85      }
86      inline GLfloat& TCoordinate4::r()
87      {
88          return _data[2];
89      }
90      inline GLfloat& TCoordinate4::q()
91      {
92          return _data[3];
93      }
94      // get constant pointer to constant data
95      inline const GLfloat* TCoordinate4::address() const
96      {
97          return _data;
98      }
99      // clone function required by smart pointers based on the deep copy ownership policy
100     inline TCoordinate4* TCoordinate4::clone() const
101     {
102         return new (std::nothrow) TCoordinate4(*this);
103     }
104     // overloaded output to stream operator
105     inline std::ostream& operator <<(std::ostream& lhs, const TCoordinate4 &rhs)
106     {
107         return lhs << rhs.s() << " " << rhs.t() << rhs.r() << " " << rhs.q();
108     }
109     // overloaded input from stream operator
110     inline std::istream& operator >>(std::istream& lhs, TCoordinate4 &rhs)
111     {
112         return lhs >> rhs[0] >> rhs[1] >> rhs[2] >> rhs[3];
113     }
```


© Ágoston Róth, 2018

```
114 }
115 #endif // TCOORDINATES4_H
```

2.6 Colors

In some of our examples we will use predefined color objects that provide fixed values for red, green, blue and alpha channels. The definition and implementation of the class Color4 can be found in Listing 2.14/47, while its diagram is presented in Fig. 2.9/47. Color instances can also be used to describe ambient, diffuse and specular light intensities and to define reflection coefficients in case of materials.



Fig. 2.9: Class diagram of colors with red, green, blue and alpha channels

Listing 2.14. Colors (**Core/Geometry/Coordinates/Colors4.h**)

```
1  #ifndef COLORS4_H
2  #define COLORS4_H

3  #include <GL/glew.h>

4  #include <algorithm>
5  #include <cassert>
6  #include <new>

7  namespace cagd
8  {
9      class Color4
10     {
11     private:
12         // red, green, blue and alpha color components
13         GLfloat _data[4];

14     public:
15         // default constructor
16         Color4();
```


© Ágoston Róth, 2018

```
17          // special constructor
18          Color4(GLfloat r, GLfloat g, GLfloat b, GLfloat a = 1.0f);

19          // get components by constant references
20          const GLfloat& operator [](const GLint &rhs) const;
21          const GLfloat& r() const;
22          const GLfloat& g() const;
23          const GLfloat& b() const;
24          const GLfloat& a() const;

25          // get components by non-constant references
26          GLfloat& operator [](const GLint &rhs);
27          GLfloat& r();
28          GLfloat& g();
29          GLfloat& b();
30          GLfloat& a();

31          // overloaded binary arithmetical operators
32          const Color4 operator +(const Color4 &rhs) const;
33          const Color4 operator -(const Color4 &rhs) const;
34          const Color4 operator *(const Color4 &rhs) const;
35          const Color4 operator *(const GLfloat &rhs) const;
36          const Color4 operator /(const Color4 &rhs) const;
37          const Color4 operator /(const GLfloat &rhs) const;

38          Color4& operator +=(const Color4 &rhs);
39          Color4& operator -=(const Color4 &rhs);
40          Color4& operator *=(const Color4 &rhs);
41          Color4& operator *=(const GLfloat &rhs);
42          Color4& operator /=(const Color4 &rhs);
43          Color4& operator /=(const GLfloat &rhs);

44          // get constant pointer to constant data
45          const GLfloat * address() const;

46          // clone function required by smart pointers based on the deep copy ownership policy
47          Color4* clone() const;
48      };

49      // default constructor
50      inline Color4::Color4()
51      {
52          _data[0] = _data[1] = _data[2] = 0.0f;
53          _data[3] = 1.0f;
54      }

55      // special constructor
56      inline Color4::Color4(GLfloat r, GLfloat g, GLfloat b, GLfloat a)
57      {
58          _data[0] = std::max(0.0f, std::min(r, 1.0f));
59          _data[1] = std::max(0.0f, std::min(g, 1.0f));
60          _data[2] = std::max(0.0f, std::min(b, 1.0f));
61          _data[3] = std::max(0.0f, std::min(a, 1.0f));
62      }

63      // get components by non-constant references
64      inline GLfloat& Color4::r()
65      {
66          return _data[0];
67      }

68      inline GLfloat& Color4::g()
69      {
70          return _data[1];
71      }

72      inline GLfloat& Color4::b()
73      {
74          return _data[2];
75      }

76      inline GLfloat& Color4::a()
77      {
```


© Ágoston Róth, 2018

```
78          return _data[3];
79      }
80      inline GLfloat& Color4::operator [](const GLint &rhs)
81      {
82          assert("Color component index is out of bounds!" && (rhs >= 0 && rhs < 4));
83          return _data[rhs];
84      }
85      // get components by constant references
86      inline const GLfloat& Color4::r() const
87      {
88          return _data[0];
89      }
90      inline const GLfloat& Color4::g() const
91      {
92          return _data[1];
93      }
94      inline const GLfloat& Color4::b() const
95      {
96          return _data[2];
97      }
98      inline const GLfloat& Color4::a() const
99      {
100         return _data[3];
101     }
102     inline const GLfloat& Color4::operator [](const GLint &rhs) const
103     {
104         assert("Color component index is out of bounds!" && (rhs >= 0 && rhs < 4));
105         return _data[rhs];
106     }
107     // overloaded binary arithmetical operators
108     inline const Color4 Color4::operator +(const Color4 &rhs) const
109     {
110         return Color4(_data[0] + rhs._data[0],
111                       _data[1] + rhs._data[1],
112                       _data[2] + rhs._data[2],
113                       _data[3] + rhs._data[3]);
114     }
115     inline const Color4 Color4::operator -(const Color4 &rhs) const
116     {
117         return Color4(_data[0] - rhs._data[0],
118                       _data[1] - rhs._data[1],
119                       _data[2] - rhs._data[2],
120                       _data[3] - rhs._data[3]);
121     }
122     inline const Color4 Color4::operator *(const Color4 &rhs) const
123     {
124         return Color4(_data[0] * rhs._data[0],
125                       _data[1] * rhs._data[1],
126                       _data[2] * rhs._data[2],
127                       _data[3] * rhs._data[3]);
128     }
129     inline const Color4 Color4::operator *(const GLfloat &rhs) const
130     {
131         return Color4(_data[0] * rhs,
132                       _data[1] * rhs,
133                       _data[2] * rhs,
134                       _data[3] * rhs);
135     }
136     inline const Color4 operator *(const GLfloat &lhs, const Color4 &rhs)
137     {
138         return Color4(lhs * rhs[0], lhs * rhs[1], lhs * rhs[2], lhs * rhs[3]);
139     }
```


© Ágoston Róth, 2018

```
140     inline const Color4 Color4::operator /(const Color4 &rhs) const
141     {
142         return Color4(_data[0] / rhs._data[0],
143                       _data[1] / rhs._data[1],
144                       _data[2] / rhs._data[2],
145                       _data[3] / rhs._data[3]);
146     }

147     inline const Color4 Color4::operator /(const GLfloat &rhs) const
148     {
149         return Color4(_data[0] / rhs,
150                       _data[1] / rhs,
151                       _data[2] / rhs,
152                       _data[3] / rhs);
153     }

154     inline Color4& Color4::operator +=(const Color4 &rhs)
155     {
156         for (int i = 0; i < 4; i++)
157         {
158             _data[i] += rhs._data[i];
159             _data[i] =  std::max(0.0f, std::min(_data[i], 1.0f));
160         }

161         return *this;
162     }

163     inline Color4& Color4::operator -=(const Color4 &rhs)
164     {
165         for (int i = 0; i < 4; i++)
166         {
167             _data[i] -= rhs._data[i];
168             _data[i] =  std::max(0.0f, std::min(_data[i], 1.0f));
169         }

170         return *this;
171     }

172     inline Color4& Color4::operator *=(const Color4 &rhs)
173     {
174         for (int i = 0; i < 4; i++)
175         {
176             _data[i] *= rhs._data[i];
177             _data[i] =  std::max(0.0f, std::min(_data[i], 1.0f));
178         }

179         return *this;
180     }

181     inline Color4& Color4::operator *=(const GLfloat &rhs)
182     {
183         for (int i = 0; i < 4; i++)
184         {
185             _data[i] *= rhs;
186             _data[i] =  std::max(0.0f, std::min(_data[i], 1.0f));
187         }

188         return *this;
189     }

190     inline Color4& Color4::operator /=(const Color4 &rhs)
191     {
192         for (int i = 0; i < 4; i++)
193         {
194             _data[i] /= rhs[i];
195             _data[i] =  std::max(0.0f, std::min(_data[i], 1.0f));
196         }

197         return *this;
198     }

199     inline Color4& Color4::operator /=(const GLfloat &rhs)
200     {
201         for (int i = 0; i < 4; i++)
```


© Ágoston Róth, 2018

```
202         {
203             _data[i] /= rhs;
204             _data[i] =  std::max(0.0f, std::min(_data[i], 1.0f));
205         }
206         return *this;
207     }
208     // get constant pointer to constant data
209     inline const GLfloat* Color4::address() const
210     {
211         return _data;
212     }
213     // clone function required by smart pointers based on the deep copy ownership policy
214     inline Color4* Color4::clone() const
215     {
216         return new (std::nothrow) Color4(*this);
217     }
218     // predefined colors
219     namespace colors
220     {
221         const Color4         black(0.000000f, 0.000000f, 0.000000f, 0.500000f);   //
222         const Color4          gray(0.375000f, 0.375000f, 0.375000f, 0.500000f);   //
223         const Color4    light_gray(0.500000f, 0.500000f, 0.500000f, 0.500000f);   //
224         const Color4        silver(0.708232f, 0.708232f, 0.708232f, 0.500000f);   //
225         const Color4         white(1.000000f, 1.000000f, 1.000000f, 0.500000f);   //
226         const Color4      dark_red(0.474937f, 0.080613f, 0.063188f, 0.500000f);   //
227         const Color4           red(0.851563f, 0.144531f, 0.113281f, 0.500000f);   //
228         const Color4     light_red(1.000000f, 0.276371f, 0.244388f, 0.500000f);   //
229         const Color4         green(0.000000f, 0.570313f, 0.246094f, 0.500000f);   //
230         const Color4   light_green(0.659983f, 0.750576f, 0.457252f, 0.500000f);   //
231         const Color4     dark_blue(0.156863f, 0.086275f, 0.435294f, 0.500000f);   //
232         const Color4          blue(0.156863f, 0.086275f, 0.652941f, 0.500000f);   //
233         const Color4     baby_blue(0.400000f, 0.478431f, 0.701961f, 0.500000f);   //
234         const Color4          cyan(0.000000f, 0.576471f, 0.866667f, 0.500000f);   //
235         const Color4    light_blue(0.456214f, 0.728115f, 1.000000f, 0.500000f);   //
236         const Color4      ice_blue(0.458824f, 0.772549f, 0.941176f, 0.500000f);   //
237         const Color4   dark_purple(0.592157f, 0.270588f, 0.470588f, 0.500000f);   //
238         const Color4        purple(0.710582f, 0.324697f, 0.564721f, 0.500000f);   //
239         const Color4 light_purple(0.850000f, 0.700000f, 1.000000f, 0.500000f);   //
240         const Color4   dark_orange(0.902344f, 0.468750f, 0.089844f, 0.500000f);   //
241         const Color4        orange(1.000000f, 0.705730f, 0.448600f, 0.500000f);   //
242     }
243 }
244 #endif // COLORS4_H
```

2.7 Lights

We also provide data structures for directional, point and spotlights that can be used to define light instances which can be sent as uniform variables to shader programs instantiated from our class ShaderProgram that is defined and implemented in Listings 2.35/129 and 2.36/135, respectively. Our shader programs assume that light sources are defined by means of the structure LightSource detailed in Listing 2.15/51.

Listing 2.15. Light source data structures assumed by our shader programs

```
1 struct LightSource
2 {
3     bool     enabled;
4     vec4     position;
5     vec4     half_vector;
6     vec4     ambient;
```


© Ágoston Róth, 2018

```
7       vec4    diffuse;
8       vec4    specular;
9       float   spot_cos_cutoff;
10      float   constant_attenuation;
11      float   linear_attenuation;
12      float   quadratic_attenuation;
13      vec3    spot_direction;
14      float   spot_exponent;
15  };
```

Note that in order to render the geometry and to handle different types of light objects it is not necessary to use the provided class ShaderProgram. As we will see, each of our rendering methods has two overloaded variants. The first one assumes that the rendering is done through a ShaderProgram instance, while the second one accepts locations of user-defined position, color, normal and (possibly projective) texture attributes of types vec3, vec4, vec3 and vec4, respectively, i.e., users may define their custom shader program classes and light source objects in a totally different way. Naturally, in the latter case the handling of attributes and the communication with uniform variables are the obligations of the user. These second variants of our rendering methods will verify whether the given attribute locations can be found in the list of active attribute locations of the currently active shader program and whether they are associated with attributes of correct types.

Classes DirectionalLight, PointLight and Spotlight were introduced in our function library for convenience in order to ease the communication with our ShaderProgram instances. Their class diagrams are illustrated in Figs. 2.10/52, 2.11/53 and 2.12/53, while their definitions and implementations can be found in Listings 2.16/52 and 2.17/55, respectively.

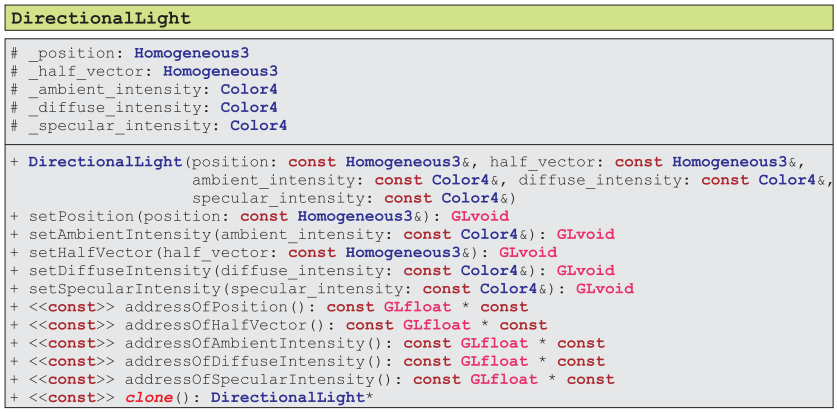


Fig. 2.10: Class diagram of directional lights

Listing 2.16. Directional, point and spot lights (**Core/Geometry/Surfaces/Lights.h**)

```
1  #ifndef LIGHTS_H
2  #define LIGHTS_H
3
3  #include "../Coordinates/Cartesians3.h"
4  #include "../Coordinates/Colors4.h"
5  #include "../Coordinates/Homogeneous3.h"
6
6  namespace cagd
7  {
8      class DirectionalLight
9      {
10     protected:
11         Homogeneous3 _position, _half_vector;
```


© Ágoston Róth, 2018

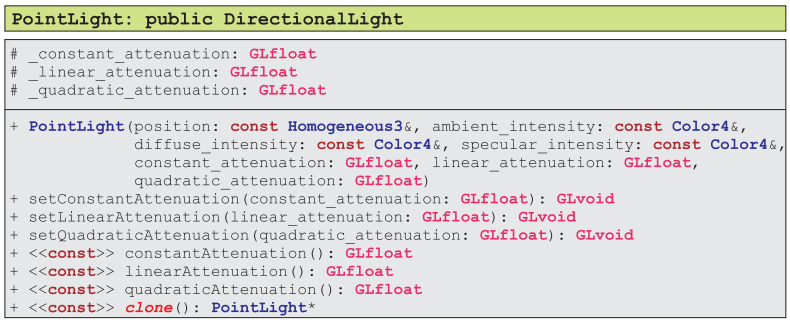


Fig. 2.11: Class diagram of point lights

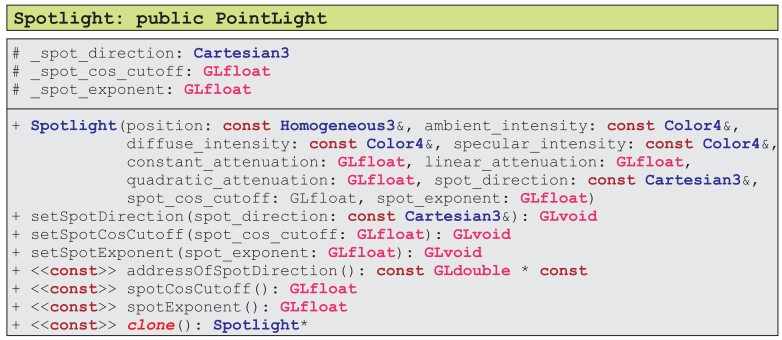


Fig. 2.12: Class diagram of spotlights

```
12          Color4          _ambient_intensity, _diffuse_intensity, _specular_intensity;

13      public:
14          // special constructor
15          DirectionalLight(
16              const Homogeneous3 &position,
17              const Homogeneous3 &half_vector,
18              const Color4       &ambient_intensity,
19              const Color4       &diffuse_intensity,
20              const Color4       &specular_intensity);

21          // setters
22          GLvoid setPosition(const Homogeneous3 &position);
23          GLvoid setHalfVector(const Homogeneous3 &half_vector);
24          GLvoid setAmbientIntensity(const Color4 &ambient_intensity);
25          GLvoid setDiffuseIntensity(const Color4 &diffuse_intensity);
26          GLvoid setSpecularIntensity(const Color4 &specular_intensity);

27          // getters
28          const GLfloat* addressOfPosition() const;
29          const GLfloat* addressOfHalfVector() const;
30          const GLfloat* addressOfAmbientIntensity() const;
31          const GLfloat* addressOfDiffuseIntensity() const;
32          const GLfloat* addressOfSpecularIntensity() const;

33          // clone function required by smart pointers based on the deep copy ownership policy
34          virtual DirectionalLight* clone() const;

35          // virtual destructor
36          virtual ~DirectionalLight();
37      };

38      class PointLight: public DirectionalLight
39      {
```


© Ágoston Róth, 2018

```
40      protected:
41          GLfloat _constant_attenuation,
42                  _linear_attenuation,
43                  _quadratic_attenuation;

44      public:
45          // special constructor
46          PointLight(
47              const Homogeneous3  &position,
48              const Color4        &ambient_intensity,
49              const Color4        &diffuse_intensity,
50              const Color4        &specular_intensity,
51              GLfloat              constant_attenuation,
52              GLfloat              linear_attenuation,
53              GLfloat              quadratic_attenuation);

54          // setters
55          GLvoid setConstantAttenuation(GLfloat constant_attenuation);
56          GLvoid setLinearAttenuation(GLfloat linear_attenuation);
57          GLvoid setQuadraticAttenuation(GLfloat quadratic_attenuation);

58          // getters
59          GLfloat constantAttenuation() const;
60          GLfloat linearAttenuation() const;
61          GLfloat quadraticAttenuation() const;

62          // redeclared clone function required by smart pointers based on the deep copy ownership policy
63          PointLight* clone() const;

64          // virtual destructor
65          virtual ~PointLight();
66      };

67      class Spotlight: public PointLight
68      {
69      private:
70          Cartesian3 _spot_direction;
71          GLfloat    _spot_cos_cutoff, _spot_exponent;

72      public:
73          // special constructor
74          Spotlight(
75              const Homogeneous3& position,
76              const Color4        &ambient_intensity,
77              const Color4        &diffuse_intensity,
78              const Color4        &specular_intensity,
79              GLfloat              constant_attenuation,
80              GLfloat              linear_attenuation,
81              GLfloat              quadratic_attenuation,
82              const Cartesian3    &spot_direction,
83              GLfloat              spot_cos_cutoff,
84              GLfloat              spot_exponent);

85          // setters
86          GLvoid setSpotDirection(const Cartesian3 &spot_direction);
87          GLvoid setSpotCosCutoff(GLfloat spot_cos_cutoff);
88          GLvoid setSpotExponent(GLfloat spot_exponent);

89          // getters
90          const GLdouble* addressOfSpotDirection() const;
91          GLfloat spotCosCutoff() const;
92          GLfloat spotExponent() const;

93          // redeclared clone function required by smart pointers based on the deep copy ownership policy
94          Spotlight* clone() const;

95          // virtual default destructor
96          virtual ~Spotlight();
97      };
98  }

99  #endif // LIGHTS_H
```


© Ágoston Róth, 2018

Listing 2.17. Directional, point and spot lights (**Core/Geometry/Surfaces/Lights.cpp**)

```
#include "Lights.h"
#include "../../Exceptions.h"

#include <new>

using namespace std;

namespace cagd
{
    // special constructor
    DirectionalLight::DirectionalLight(
        const Homogeneous3 &position,
        const Homogeneous3 &half_vector,
        const Color4       &ambient_intensity,
        const Color4       &diffuse_intensity,
        const Color4       &specular_intensity):

        _position(position),
        _half_vector(half_vector),
        _ambient_intensity(ambient_intensity),
        _diffuse_intensity(diffuse_intensity),
        _specular_intensity(specular_intensity)
    {
    }

    // setters
    GLvoid DirectionalLight::setPosition(const Homogeneous3 &position)
    {
        _position = position;
    }

    GLvoid DirectionalLight::setHalfVector(const Homogeneous3 &half_vector)
    {
        _half_vector = half_vector;
    }

    GLvoid DirectionalLight::setAmbientIntensity(const Color4 &ambient_intensity)
    {
        _ambient_intensity = ambient_intensity;
    }

    GLvoid DirectionalLight::setDiffuseIntensity(const Color4 &diffuse_intensity)
    {
        _diffuse_intensity = diffuse_intensity;
    }

    GLvoid DirectionalLight::setSpecularIntensity(const Color4 &specular_intensity)
    {
        _specular_intensity = specular_intensity;
    }

    // getters
    const GLfloat* DirectionalLight::addressOfPosition() const
    {
        return _position.address();
    }

    const GLfloat* DirectionalLight::addressOfHalfVector() const
    {
        return _half_vector.address();
    }

    const GLfloat* DirectionalLight::addressOfAmbientIntensity() const
    {
        return _ambient_intensity.address();
    }

    const GLfloat* DirectionalLight::addressOfDiffuseIntensity() const
    {
        return _diffuse_intensity.address();
    }
```


© Ágoston Róth, 2018

```
59      const GLfloat* DirectionalLight::addressOfSpecularIntensity() const
60      {
61          return _specular_intensity.address();
62      }
63      // clone function required by smart pointers based on the deep copy ownership policy
64      DirectionalLight* DirectionalLight::clone() const
65      {
66          return new (nothrow) DirectionalLight(*this);
67      }
68      // virtual default destructor
69      DirectionalLight::~DirectionalLight() = default;
70      // special constructor
71      PointLight::PointLight(
72          const Homogeneous3  &position,
73          const Color4        &ambient_intensity,
74          const Color4        &diffuse_intensity,
75          const Color4        &specular_intensity,
76          GLfloat              constant_attenuation,
77          GLfloat              linear_attenuation,
78          GLfloat              quadratic_attenuation):
79          DirectionalLight(position, Homogeneous3(),
80                           ambient_intensity, diffuse_intensity, specular_intensity),
81          _constant_attenuation(constant_attenuation),
82          _linear_attenuation(linear_attenuation),
83          _quadratic_attenuation(quadratic_attenuation)
84      {
85          if (position.w() == 0.0f)
86          {
87              throw Exception("PointLight::PointLight - Wrong position.");
88          }
89      }
90      // setters
91      GLvoid PointLight::setConstantAttenuation(GLfloat constant_attenuation)
92      {
93          _constant_attenuation = constant_attenuation;
94      }
95      GLvoid PointLight::setLinearAttenuation(GLfloat linear_attenuation)
96      {
97          _linear_attenuation = linear_attenuation;
98      }
99      GLvoid PointLight::setQuadraticAttenuation(GLfloat quadratic_attenuation)
100     {
101         _quadratic_attenuation = quadratic_attenuation;
102     }
103     // getters
104     GLfloat PointLight::constantAttenuation() const
105     {
106         return _constant_attenuation;
107     }
108     GLfloat PointLight::linearAttenuation() const
109     {
110         return _linear_attenuation;
111     }
112     GLfloat PointLight::quadraticAttenuation() const
113     {
114         return _quadratic_attenuation;
115     }
116     // redefined clone function required by smart pointers based on the deep copy ownership policy
117     PointLight* PointLight::clone() const
118     {
119         return new (nothrow) PointLight(*this);
```


© Ágoston Róth, 2018

```
120     }
121     // virtual default destructor
122     PointLight::~PointLight() = default;
123     // special constructor
124     Spotlight::Spotlight(
125         const Homogeneous3 &position,
126         const Color4       &ambient_intensity,
127         const Color4       &diffuse_intensity,
128         const Color4       &specular_intensity,
129         GLfloat             constant_attenuation,
130         GLfloat             linear_attenuation,
131         GLfloat             quadratic_attenuation,
132         const Cartesian3   &spot_direction,
133         GLfloat             spot_cos_cutoff,
134         GLfloat             spot_exponent):
135         PointLight(position,
136                    ambient_intensity, diffuse_intensity, specular_intensity,
137                    constant_attenuation, linear_attenuation, quadratic_attenuation),
138         _spot_direction(spot_direction),
139         _spot_cos_cutoff(spot_cos_cutoff),
140         _spot_exponent(spot_exponent)
141     {
142         if (position.w() == 0.0f)
143         {
144             throw Exception("Spotlight::Spotlight - Wrong position.");
145         }
146         if (_spot_cos_cutoff > 90.0f)
147         {
148             throw Exception("Spotlight::Spotlight - Wrong spot cosine cutoff.");
149         }
150     }
151     // setters
152     GLvoid Spotlight::setSpotDirection(const Cartesian3 &spot_direction)
153     {
154         _spot_direction = spot_direction;
155     }
156     GLvoid Spotlight::setSpotCosCutoff(GLfloat spot_cos_cutoff)
157     {
158         _spot_cos_cutoff = spot_cos_cutoff;
159     }
160     GLvoid Spotlight::setSpotExponent(GLfloat spot_exponent)
161     {
162         _spot_exponent = spot_exponent;
163     }
164     // getters
165     const GLdouble* Spotlight::addressOfSpotDirection() const
166     {
167         return _spot_direction.address();
168     }
169     GLfloat Spotlight::spotCosCutoff() const
170     {
171         return _spot_cos_cutoff;
172     }
173     GLfloat Spotlight::spotExponent() const
174     {
175         return _spot_exponent;
176     }
177     // redefined clone function required by smart pointers based on the deep copy ownership policy
178     Spotlight* Spotlight::clone() const
179     {
180         return new (nothrow) Spotlight(*this);
181     }
```


© Ágoston Róth, 2018

```
182     // virtual default destructor
183     Spotlight::~Spotlight() = default;
184 }
```

2.8 Materials

We also provide a class for materials whose instances can be sent as uniform variables to shader programs instantiated from our class ShaderProgram that is defined and implemented in Listings 2.35/129 and 2.36/135, respectively. Our shader programs assume that uniform material variables are defined by means of the structure Material detailed in Listing 2.18/58.

Listing 2.18. Type of uniform material variables assumed by our shader programs

```
1 struct Material
2 {
3     vec4    ambient;
4     vec4    diffuse;
5     vec4    specular;
6     vec4    emission;
7     float   shininess;
8 };
```

Note that in order to render the geometry and to handle different types of materials it is not necessary to use the provided class ShaderProgram. As we will see, each of our rendering methods has two overloaded variants. The first one assumes that the rendering is done through a ShaderProgram instance, while the second one accepts locations of user-defined position, color, normal and (possibly projective) texture attributes of types vec3, vec4, vec3 and vec4, respectively, i.e., users may define their custom shader program classes and materials in a totally different way. Naturally, in the latter case the handling of attributes and the communication with uniform variables are the obligations of the user. These second variants of our rendering methods will verify whether the given attribute locations can be found in the list of active attribute locations of the currently active shader program and whether they are associated with attributes of correct types.

The class Material was introduced in our function library for convenience in order to ease the communication with our ShaderProgram instances, its diagram is illustrated in Fig. 2.13/59, while its definition and implementation can be found in Listings 2.19/58 and 2.20/60, respectively. We also provide in lines 105/62–221/65 of Listing 2.20/60 predefined materials like black plastique, black rubber, brass, bronze, chrome, copper, emerald, gold, jade, obsidian, pearl, pewter, polished bronze, polished copper, polished gold, polished silver, ruby, silver and turquoise.

Listing 2.19. Materials (**Core/Geometry/Surfaces/Materials.h**)

```
1 #ifndef MATERIALS_H
2 #define MATERIALS_H
3 #include "../Coordinates/Colors4.h"
4 namespace cagd
5 {
6     class Material
7     {
8     private:
9         // ambient, diffuse, specular reflection coefficients and emissive color components
10        Color4  _ambient, _diffuse, _specular, _emission;
11        // shininess of the material, its value should be in the interval [0, 128]
12        GLfloat _shininess;
```


© Ágoston Róth, 2018

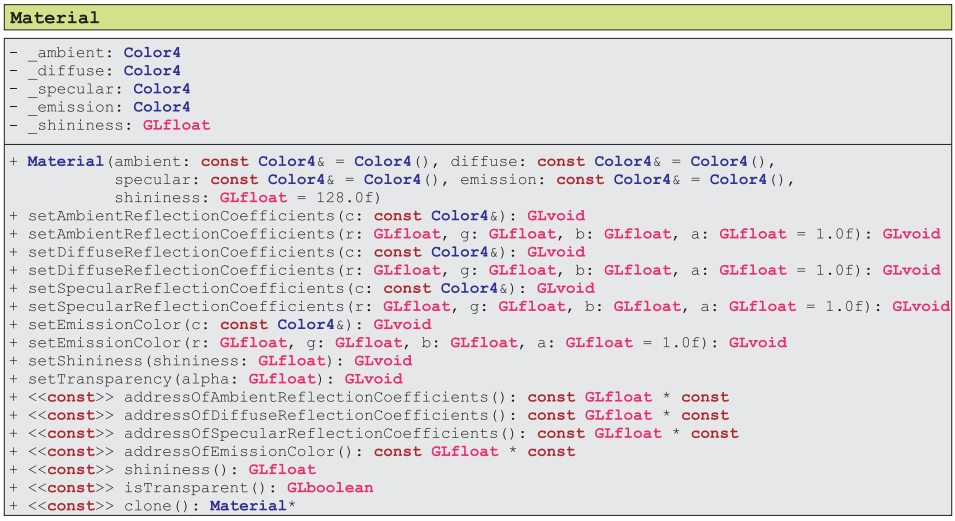


Fig. 2.13: Class diagram of materials

```
13      public:
14          // default/special constructor
15          Material(const Color4 &ambient   = Color4(),
16                   const Color4 &diffuse   = Color4(),
17                   const Color4 &specular  = Color4(),
18                   const Color4 &emission  = Color4(),
19                   GLfloat shininess = 128.0f);

20          // setters
21          GLvoid setAmbientReflectionCoefficients(const Color4 &c);
22          GLvoid setAmbientReflectionCoefficients(
23                       GLfloat r, GLfloat g, GLfloat b, GLfloat a = 1.0f);

24          GLvoid setDiffuseReflectionCoefficients(const Color4 &c);
25          GLvoid setDiffuseReflectionCoefficients(
26                       GLfloat r, GLfloat g, GLfloat b, GLfloat a = 1.0f);

27          GLvoid setSpecularReflectionCoefficients(const Color4 &c);
28          GLvoid setSpecularReflectionCoefficients(
29                       GLfloat r, GLfloat g, GLfloat b, GLfloat a = 1.0f);

30          GLvoid setEmissionColor(const Color4 &c);
31          GLvoid setEmissionColor(
32                       GLfloat r, GLfloat g, GLfloat b, GLfloat a = 1.0f);

33          GLvoid setShininess(GLfloat shininess);
34          GLvoid setTransparency(GLfloat alpha);

35          // getters
36          const GLfloat* addressOfAmbientReflectionCoefficients()  const;
37          const GLfloat* addressOfDiffuseReflectionCoefficients()  const;
38          const GLfloat* addressOfSpecularReflectionCoefficients() const;
39          const GLfloat* addressOfEmissionColor() const;

40          GLfloat         shininess() const;
41          GLboolean       isTransparent() const;

42          // clone function required by smart pointers based on the deep copy ownership policy
43          Material* clone() const;
44      };

45      // predefined materials
46      namespace materials
47      {
48          extern Material black_plastique,
```


© Ágoston Róth, 2018

```
                        black_rubber,
                        brass,
                        bronze,
                        chrome,
                        copper,
                        emerald,
                        gold,
                        jade,
                        obsidian,
                        pearl,
                        pewter,
                        polished_bronze,
                        polished_copper,
                        polished_gold,
                        polished_silver,
                        ruby,
                        silver,
                        turquoise;
    }
}

#endif // MATERIALS_H
```

Listing 2.20. Materials (**Core/Geometry/Surfaces/Materials.cpp**)

```
#include "Materials.h"

#include <new>

using namespace std;

namespace cagd
{
    // default/special constructor
    Material::Material(const Color4 &ambient,  const Color4 &diffuse,
                       const Color4 &specular, const Color4 &emission,
                       GLfloat shininess):
        _ambient(ambient),
        _diffuse(diffuse),
        _specular(specular),
        _emission(emission),
        _shininess(shininess)
    {
    }

    // setters
    GLvoid Material::setAmbientReflectionCoefficients(
                GLfloat r, GLfloat g, GLfloat b, GLfloat a)
    {
        _ambient.r() = r;
        _ambient.g() = g;
        _ambient.b() = b;
        _ambient.a() = a;
    }

    GLvoid Material::setDiffuseReflectionCoefficients(
                GLfloat r, GLfloat g, GLfloat b, GLfloat a)
    {
        _diffuse.r() = r;
        _diffuse.g() = g;
        _diffuse.b() = b;
        _diffuse.a() = a;
    }

    GLvoid Material::setSpecularReflectionCoefficients(
                GLfloat r, GLfloat g, GLfloat b, GLfloat a)
    {
        _specular.r() = r;
        _specular.g() = g;
        _specular.b() = b;
        _specular.a() = a;
    }
```


© Ágoston Róth, 2018

```
42      GLvoid Material::setEmissionColor(
43                  GLfloat r, GLfloat g, GLfloat b, GLfloat a)
44      {
45          _emission.r() = r;
46          _emission.g() = g;
47          _emission.b() = b;
48          _emission.a() = a;
49      }

50      GLvoid Material::setAmbientReflectionCoefficients(const Color4 &c)
51      {
52          _ambient = c;
53      }

54      GLvoid Material::setDiffuseReflectionCoefficients(const Color4 &c)
55      {
56          _diffuse = c;
57      }

58      GLvoid Material::setSpecularReflectionCoefficients(const Color4 &c)
59      {
60          _specular = c;
61      }

62      GLvoid Material::setEmissionColor(const Color4 &c)
63      {
64          _emission = c;
65      }

66      GLvoid Material::setShininess(GLfloat shininess)
67      {
68          _shininess = shininess;
69      }

70      GLvoid Material::setTransparency(GLfloat alpha)
71      {
72          _ambient.a() = _diffuse.a() = _specular.a() = alpha;
73      }

74      // getters
75      const GLfloat* Material::addressOfAmbientReflectionCoefficients() const
76      {
77          return _ambient.address();
78      }

79      const GLfloat* Material::addressOfDiffuseReflectionCoefficients() const
80      {
81          return _diffuse.address();
82      }

83      const GLfloat* Material::addressOfSpecularReflectionCoefficients() const
84      {
85          return _specular.address();
86      }

87      const GLfloat* Material::addressOfEmissionColor() const
88      {
89          return _emission.address();
90      }

91      GLfloat Material::shininess() const
92      {
93          return _shininess;
94      }

95      GLboolean Material::isTransparent() const
96      {
97          return (_diffuse.a() < 1.0f);
98      }

99      // clone function required by smart pointers based on the deep copy ownership policy
100     Material* Material::clone() const
101     {
```


© Ágoston Róth, 2018

```
102         return new (nothrow) Material(*this);
103     }
104     // predefined materials
105     namespace materials
106     {
```



```
107         Material black_plastique = Material(
108                 Color4(0.000000f, 0.000000f, 0.000000f, 0.4f),
109                 Color4(0.020000f, 0.020000f, 0.020000f, 0.6f),
110                 Color4(0.600000f, 0.600000f, 0.600000f, 0.8f),
111                 Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
112                 32.0f),
```



```
113             black_rubber = Material(
114                 Color4(0.000000f, 0.000000f, 0.000000f, 0.4f),
115                 Color4(0.010000f, 0.010000f, 0.010000f, 0.6f),
116                 Color4(0.500000f, 0.500000f, 0.500000f, 0.8f),
117                 Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
118                 32.0f),
```



```
119             brass = Material(
120                 Color4(0.329412f, 0.223529f, 0.027451f, 0.4f),
121                 Color4(0.780392f, 0.568627f, 0.113725f, 0.6f),
122                 Color4(0.992157f, 0.941176f, 0.807843f, 0.8f),
123                 Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
124                 27.8974f),
```



```
125             bronze = Material(
126                 Color4(0.250000f, 0.148000f, 0.064750f, 0.4f),
127                 Color4(0.400000f, 0.236800f, 0.103600f, 0.6f),
128                 Color4(0.774597f, 0.458561f, 0.200621f, 0.8f),
129                 Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
130                 76.8f),
```



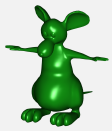
```
131             chrome = Material(
132                 Color4(0.250000f, 0.250000f, 0.250000f, 0.4f),
133                 Color4(0.400000f, 0.400000f, 0.400000f, 0.6f),
134                 Color4(0.774597f, 0.774597f, 0.774597f, 0.8f),
135                 Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
136                 76.8f),
```



```
137             copper = Material(
138                 Color4(0.191250f, 0.073500f, 0.022500f, 0.4f),
```


© Ágoston Róth, 2018

```
139                     Color4(0.703800f, 0.270480f, 0.082800f, 0.6f),
140                     Color4(0.256777f, 0.137622f, 0.086014f, 0.8f),
141                     Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
142                     51.2f),
```



```
143             emerald = Material(
144                     Color4(0.021500f, 0.174500f, 0.021500f, 0.4f),
145                     Color4(0.075680f, 0.614240f, 0.075680f, 0.6f),
146                     Color4(0.633000f, 0.727811f, 0.633000f, 0.8f),
147                     Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
148                     76.8f),
```



```
149             gold = Material(
150                     Color4(0.247250f, 0.224500f, 0.064500f, 0.4f),
151                     Color4(0.346150f, 0.314300f, 0.090300f, 0.6f),
152                     Color4(0.797357f, 0.723991f, 0.208006f, 0.8f),
153                     Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
154                     83.2f),
```



```
155             jade = Material(
156                     Color4(0.135000f, 0.222500f, 0.157500f, 0.4f),
157                     Color4(0.540000f, 0.890000f, 0.630000f, 0.6f),
158                     Color4(0.316228f, 0.316228f, 0.316228f, 0.8f),
159                     Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
160                     12.8f),
```



```
161             obsidian = Material(
162                     Color4(0.053750f, 0.050000f, 0.066250f, 0.4f),
163                     Color4(0.182750f, 0.170000f, 0.225250f, 0.6f),
164                     Color4(0.332741f, 0.328634f, 0.346435f, 0.8f),
165                     Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
166                     38.4f),
```



```
167             pearl = Material(
168                     Color4(0.250000f, 0.207250f, 0.207250f, 0.4f),
169                     Color4(1.000000f, 0.829000f, 0.829000f, 0.6f),
170                     Color4(0.296648f, 0.296648f, 0.296648f, 0.8f),
171                     Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
172                     11.264f),
```



```
173             pewter = Material(
174                     Color4(0.105882f, 0.058824f, 0.113725f, 0.4f),
175                     Color4(0.427451f, 0.470588f, 0.541176f, 0.6f),
```


© Ágoston Róth, 2018

```
176                 Color4(0.333333f, 0.333333f, 0.521569f, 0.8f),
177                 Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
178                 9.84615f),
```



```
179         polished_bronze = Material(
180                 Color4(0.212500f, 0.127500f, 0.054000f, 0.4f),
181                 Color4(0.714000f, 0.428400f, 0.181440f, 0.6f),
182                 Color4(0.393548f, 0.271906f, 0.166721f, 0.8f),
183                 Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
184                 25.6f),
```



```
185         polished_copper = Material(
186                 Color4(0.229500f, 0.088250f, 0.0275000f, 0.4f),
187                 Color4(0.550800f, 0.211800f, 0.0660000f, 0.6f),
188                 Color4(0.580594f, 0.223257f, 0.0695701f, 0.8f),
189                 Color4(0.000000f, 0.000000f, 0.0000000f, 0.0f),
190                 12.8f),
```



```
191         polished_gold = Material(
192                 Color4(0.247250f, 0.199500f, 0.074500f, 0.4f),
193                 Color4(0.751640f, 0.606480f, 0.226480f, 0.6f),
194                 Color4(0.628281f, 0.555802f, 0.366065f, 0.8f),
195                 Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
196                 51.2f),
```



```
197         polished_silver = Material(
198                 Color4(0.192250f, 0.192250f, 0.192250f, 0.4f),
199                 Color4(0.507540f, 0.507540f, 0.507540f, 0.6f),
200                 Color4(0.508273f, 0.508273f, 0.508273f, 0.8f),
201                 Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
202                 51.2f),
```



```
203         ruby = Material(
204                 Color4(0.174500f, 0.011750f, 0.011750f, 0.4f),
205                 Color4(0.614240f, 0.041360f, 0.041360f, 0.6f),
206                 Color4(0.727811f, 0.626959f, 0.626959f, 0.8f),
207                 Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
208                 76.8f),
```



```
209         silver = Material(
210                 Color4(0.231250f, 0.231250f, 0.231250f, 0.4f),
211                 Color4(0.277500f, 0.277500f, 0.277500f, 0.6f),
212                 Color4(0.773911f, 0.773911f, 0.773911f, 0.8f),
```


© Ágoston Róth, 2018

```
213                         Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
214                         89.6f),
215                 turquoise = Material(
216                         Color4(0.100000f, 0.187250f, 0.174500f, 0.4f),
217                         Color4(0.396000f, 0.741510f, 0.691020f, 0.6f),
218                         Color4(0.297254f, 0.308290f, 0.306678f, 0.8f),
219                         Color4(0.000000f, 0.000000f, 0.000000f, 0.0f),
220                         12.8f);
221     }
222 }
```



©The model file mouse.off is the creation of the author. It was rendered by means of the proposed function library.

2.9 Different template and specialized matrices

2.9.1 Generic template matrices

In order to represent traditional matrices, knot and weight vectors, control polygons and nets, Pascal triangles of binomial coefficients, triangle matrices of partial derivatives, or data point sequences and grids that have to be interpolated, we also define rectangular, column, row and triangular template matrices (Matrix<T>, ColumnMatrix<T>, RowMatrix<T> and TriangularMatrix<T>). Fig. 2.14/66 illustrates the diagrams of these classes. Their implementation can be found in Listing 2.21/65.

Listing 2.21. Rectangular, row, column and triangular template matrices (**Core/Math/Matrices.h**)

```
1  #ifndef MATRICES_H
2  #define MATRICES_H
3  #include <algorithm>
4  #include <cassert>
5  #include <iostream>
6  #include <vector>
7  namespace cagd
8  {
9      // forward declaration of the template class Matrix
10     template <typename T>
11     class Matrix;
12     // forward declaration of the template class RowMatrix
13     template <typename T>
14     class RowMatrix;
15     // forward declaration of the template class ColumnMatrix
16     template <typename T>
17     class ColumnMatrix;
18     // forward declaration of the template class TriangularMatrix
19     template <typename T>
20     class TriangularMatrix;
21     // forward declarations of overloaded and templated input/output from/to stream operators
22     template <typename T>
23     std::ostream& operator << (std::ostream& lhs, const Matrix<T>& rhs);
24     template <typename T>
25     std::istream& operator >>(std::istream& lhs, Matrix<T>& rhs);
26     template <typename T>
```


© Ágoston Róth, 2018

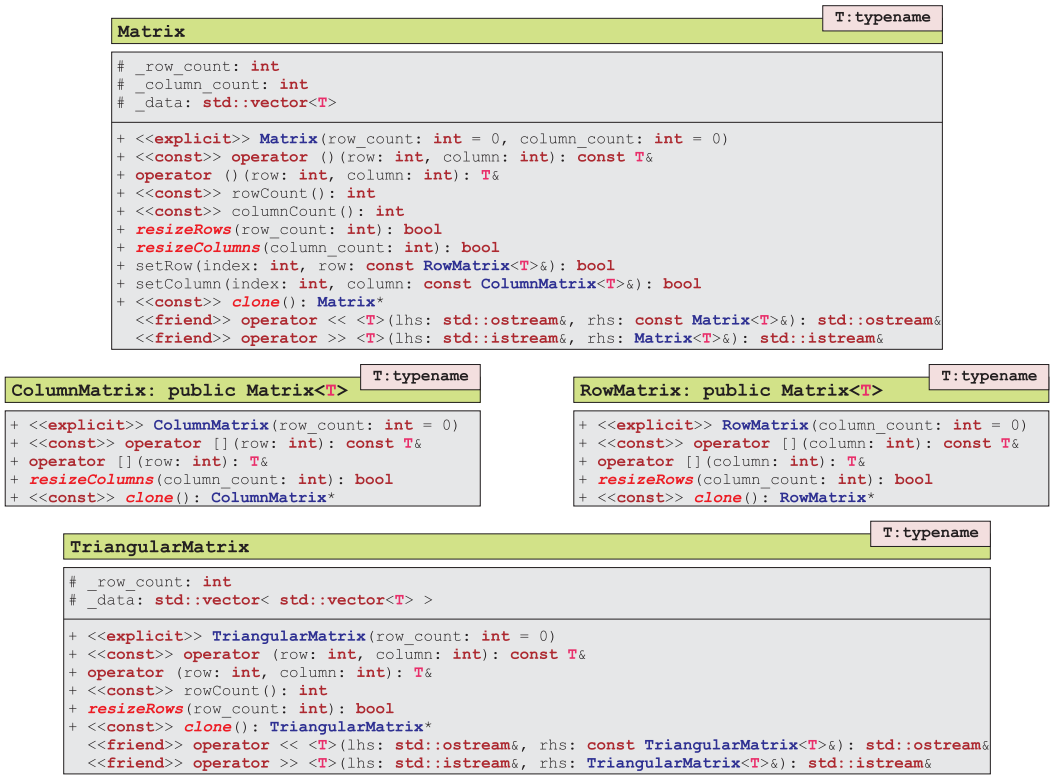


Fig. 2.14: Class diagrams of different types of template matrices

```
27      std::ostream& operator << (std::ostream& lhs, const TriangularMatrix<T>& rhs);

28      template <typename T>
29      std::istream& operator >>(std::istream& lhs, TriangularMatrix<T>& rhs);

30      // Interface of the template class Matrix:
31      template <typename T>
32      class Matrix
33      {
34          friend std::ostream& operator << <T>(std::ostream&, const Matrix<T>& rhs);
35          friend std::istream& operator >> <T>(std::istream&, Matrix<T>& rhs);

36      protected:
37          int             _row_count;
38          int             _column_count;
39          std::vector<T> _data;

40      public:
41          // default/special constructor
42          explicit Matrix(int row_count = 1, int column_count = 1);

43          // get element by non-constant reference
44          T& operator ()(int row, int column);

45          // get element by constant reference
46          const T& operator ()(int row, int column) const;

47          // get dimensions
48          int rowCount() const;
49          int columnCount() const;

50          // set dimensions
51          virtual bool resizeRows(int row_count);
```


© Ágoston Róth, 2018

```
52          virtual bool resizeColumns(int column_count);

53          // update
54          bool setRow(int index, const RowMatrix<T>& row);
55          bool setColumn(int index, const ColumnMatrix<T>& column);

56          // clone function required by smart pointers based on the deep copy ownership policy
57          virtual Matrix* clone() const;

58          // destructor
59          virtual ~Matrix();
60      };

61      // Interface of the template class RowMatrix:
62      template <typename T>
63      class RowMatrix: public Matrix<T>
64      {
65      public:
66          // default/special constructor
67          explicit RowMatrix(int column_count = 1);

68          // get element by non-constant reference
69          T& operator ()(int column);
70          T& operator [](int column);

71          // get element by constant reference
72          const T& operator ()(int column) const;
73          const T& operator [](int column) const;

74          // a row matrix consists of a single row
75          bool resizeRows(int row_count);

76          // redeclared clone function required by smart pointers based on the deep copy ownership policy
77          RowMatrix* clone() const;
78      };

79      // Interface of the template class ColumnMatrix:
80      template <typename T>
81      class ColumnMatrix: public Matrix<T>
82      {
83      public:
84          // default/special constructor
85          explicit ColumnMatrix(int row_count = 1);

86          // get element by non-constant reference
87          T& operator ()(int row);
88          T& operator [](int row);

89          // get element by constant reference
90          const T& operator ()(int row) const;
91          const T& operator [](int row) const;

92          // a column matrix consists of a single column
93          bool resizeColumns(int column_count);

94          // redeclared clone function required by smart pointers based on the deep copy ownership policy
95          ColumnMatrix* clone() const;
96      };

97      // Interface of the template class TriangularMatrix:
98      template <typename T>
99      class TriangularMatrix
100     {
101         friend std::ostream& operator << <T>(std::ostream&,
102                                              const TriangularMatrix<T>& rhs);
103         friend std::istream& operator >> <T>(std::istream&,
104                                              TriangularMatrix<T>& rhs);

105     protected:
106         int                             _row_count;
107         std::vector< std::vector<T> > _data;

108     public:
109         // default/special constructor
```


© Ágoston Róth, 2018

```
110         explicit TriangularMatrix(int row_count = 1);

111         // get element by non-constant reference
112         T& operator ()(int row, int column);

113         // get element by constant reference
114         const T& operator ()(int row, int column) const;

115         // get dimension
116         int rowCount() const;

117         // set dimension
118         virtual bool resizeRows(int row_count);

119         // clone function required by smart pointers based on the deep copy ownership policy
120         virtual TriangularMatrix* clone() const;

121         // destructor
122         virtual ~TriangularMatrix();
123     };

124     // Implementation of the template class Matrix:

125     // default/special constructor
126     template <typename T>
127     Matrix<T>::Matrix(int row_count, int column_count):
128         _row_count(row_count < 0 || column_count <= 0 ? 0 : row_count),
129         _column_count(column_count < 0 || !_row_count ? 0 : column_count),
130         _data(_row_count * _column_count)
131     {
132         assert("The row count of a matrix should be non-negative!" && row_count >= 0);
133         assert("The column count of a matrix should be non-negative!" &&
134                column_count >= 0);
135     }

136     // get element by non-constant reference
137     template <typename T>
138     inline T& Matrix<T>::operator ()(int row, int column)
139     {
140         assert("The given row index is out of bounds!" && (row >= 0 && row < _row_count));
141         assert("The given column index is out of bounds!" &&
142                (column >= 0 && column < _column_count));
143         return _data[row * _column_count + column];
144     }

145     // get element by constant reference
146     template <typename T>
147     inline const T& Matrix<T>::operator ()(int row, int column) const
148     {
149         assert("The given row index is out of bounds!" && (row >= 0 && row < _row_count));
150         assert("The given column index is out of bounds!" &&
151                (column >= 0 && column < _column_count));
152         return _data[row * _column_count + column];
153     }

154     // get dimensions
155     template <typename T>
156     inline int Matrix<T>::rowCount() const
157     {
158         return _row_count;
159     }

160     template <typename T>
161     inline int Matrix<T>::columnCount() const
162     {
163         return _column_count;
164     }

165     // set dimensions
166     template <typename T>
167     bool Matrix<T>::resizeRows(int row_count)
168     {
169         assert("The row count of a matrix should be non-negative!" && row_count >= 0);
```


© Ágoston Róth, 2018

```
170         if (row_count < 0)
171         {
172             return false;
173         }
174         if (_row_count != row_count)
175         {
176             _data.resize(row_count * _column_count);
177             _row_count = row_count;
178             if (_row_count == 0)
179             {
180                 _column_count = 0;
181             }
182         }
183         return false;
184     }
185     template <typename T>
186     bool Matrix<T>::resizeColumns(int column_count)
187     {
188         assert("The column count of a matrix should be non-negative!" &&
189                column_count >= 0);
190         if (column_count < 0)
191         {
192             return false;
193         }
194         if (_column_count != column_count)
195         {
196             std::vector<T> new_data(_row_count * column_count);
197             int preserved_column_count = std::min(column_count, _column_count);
198             #pragma omp parallel for
199             for (int r = 0; r < _row_count; r++)
200             {
201                 for (int c = 0,
202                      old_offset = r * _column_count,
203                      new_offset = r * column_count;
204                      c < preserved_column_count;
205                      c++)
206                 {
207                     int old_index       = old_offset + c;
208                     int new_index       = new_offset + c;
209                     new_data[new_index] = _data[old_index];
210                 }
211             }
212             _data.swap(new_data);
213             _column_count = column_count;
214             if (_column_count == 0)
215             {
216                 _row_count = 0;
217             }
218         }
219         return true;
220     }
221     // update
222     template <class T>
223     bool Matrix<T>::setRow(int row, const RowMatrix<T>& entire_row)
224     {
225         assert("The given column index is out of bounds!" &&
226                (row >= 0 && row < _row_count));
227         assert("The column counts of the underlying row matrices should be equal!" &&
228                _column_count == entire_row.columnCount());
229         if (row < 0 || row >= _row_count || _column_count != entire_row.columnCount())
```


© Ágoston Róth, 2018

```
230         {
231             return false;
232         }
233         int offset = row * _column_count;
234         #pragma omp parallel for
235         for (int c = 0; c < _column_count; c++)
236         {
237             _data[offset + c] = entire_row._data[c];
238         }
239         return true;
240     }
241     template <class T>
242     bool Matrix<T>::setColumn(int column, const ColumnMatrix<T>& entire_column)
243     {
244         assert("The given column index is out of bounds!" &&
245                (column >= 0 && column < _column_count));
246         assert("The row counts of the underlying column matrices should be equal!" &&
247                _row_count == entire_column.rowCount());
248         if (column < 0 || column >= _column_count ||
249             _row_count != entire_column.rowCount())
250         {
251             return false;
252         }
253         #pragma omp parallel for
254         for (int r = 0; r < _row_count; r++)
255         {
256             _data[r * _column_count + column] = entire_column._data[r];
257         }
258         return true;
259     }
260     // clone function required by smart pointers based on the deep copy ownership policy
261     template <typename T>
262     Matrix<T>* Matrix<T>::clone() const
263     {
264         return new (std::nothrow) Matrix<T>(*this);
265     }
266     // destructor
267     template <typename T>
268     Matrix<T>::~Matrix()
269     {
270         _row_count = _column_count = 0;
271         _data.clear();
272     }
273     // Implementation of the template class RowMatrix:
274     // default/special constructor
275     template <typename T>
276     RowMatrix<T>::RowMatrix(int column_count): Matrix<T>(1, column_count)
277     {
278     }
279     // get element by non-constant reference
280     template <typename T>
281     inline T& RowMatrix<T>::operator ()(int column)
282     {
283         assert("The given column index is out of bounds!" &&
284                (column >= 0 && column < this->_column_count));
285         return this->_data[column];
286     }
287     template <typename T>
288     inline T& RowMatrix<T>::operator [](int column)
289     {
290         assert("The given column index is out of bounds!" &&
```


© Ágoston Róth, 2018

```
291                 (column >= 0 && column < this->_column_count));
292         return this->_data[column];
293     }
294
295     // get element by constant reference
296     template <typename T>
297     inline const T& RowMatrix<T>::operator ()(int column) const
298     {
299         assert("The given column index is out of bounds!" &&
300                 (column >= 0 && column < this->_column_count));
301         return this->_data[column];
302     }
303
304     template <typename T>
305     inline const T& RowMatrix<T>::operator [](int column) const
306     {
307         assert("The given column index is out of bounds!" &&
308                 (column >= 0 && column < this->_column_count));
309         return this->_data[column];
310     }
311
312     // a row matrix consists of a single row
313     template <typename T>
314     bool RowMatrix<T>::resizeRows(int row_count)
315     {
316         return (row_count == 1);
317     }
318
319     // redefined clone function required by smart pointers based on the deep copy ownership policy
320     template <typename T>
321     RowMatrix<T>* RowMatrix<T>::clone() const
322     {
323         return new (std::nothrow) RowMatrix<T>(*this);
324     }
325
326     // Implementation of the template class ColumnMatrix:
327
328     // default/special constructor
329     template <typename T>
330     ColumnMatrix<T>::ColumnMatrix(int row_count): Matrix<T>(row_count, 1)
331     {
332     }
333
334     // get element by non-constant reference
335     template <typename T>
336     inline T& ColumnMatrix<T>::operator ()(int row)
337     {
338         assert("The given row index is out of bounds!" &&
339                 (row >= 0 && row < this->_row_count));
340         return this->_data[row];
341     }
342
343     template <typename T>
344     inline T& ColumnMatrix<T>::operator [](int row)
345     {
346         assert("The given row index is out of bounds!" &&
347                 (row >= 0 && row < this->_row_count));
348         return this->_data[row];
349     }
350
351     // get element by constant reference
352     template <typename T>
353     inline const T& ColumnMatrix<T>::operator ()(int row) const
354     {
355         assert("The given row index is out of bounds!" &&
356                 (row >= 0 && row < this->_row_count));
357         return this->_data[row];
358     }
359
360     template <typename T>
361     inline const T& ColumnMatrix<T>::operator [](int row) const
362     {
363         assert("The given row index is out of bounds!" &&
364                 (row >= 0 && row < this->_row_count));
```


© Ágoston Róth, 2018

```
355         return this->_data[row];
356     }
357     // a column matrix consists of a single column
358     template <typename T>
359     bool ColumnMatrix<T>::resizeColumns(int column_count)
360     {
361         return (column_count == 1);
362     }
363     // redefined clone function required by smart pointers based on the deep copy ownership policy
364     template <typename T>
365     ColumnMatrix<T>* ColumnMatrix<T>::clone() const
366     {
367         return new (std::nothrow) ColumnMatrix<T>(*this);
368     }
369     // Implementation of the template class TriangularMatrix:
370     // default/special constructor
371     template <typename T>
372     TriangularMatrix<T>::TriangularMatrix(int row_count):
373         _row_count(row_count < 0 ? 0 : row_count),
374         _data(_row_count)
375     {
376         assert("The row count of a triangular matrix should be a non-negative "
377                "integer!" && row_count >= 0);
378         for (int r = 0; r < _row_count; r++)
379         {
380             _data[r].resize(r + 1);
381         }
382     }
383     // get element by non-constant reference
384     template <typename T>
385     inline T& TriangularMatrix<T>::operator ()(int row, int column)
386     {
387         assert("The given row index is out of bounds!" && (row >= 0 && row < _row_count));
388         assert("The given column index is out of bounds!" &&
389                (column >= 0 && column <= row));
390         return _data[row][column];
391     }
392     // get element by constant reference
393     template <typename T>
394     inline const T& TriangularMatrix<T>::operator ()(int row, int column) const
395     {
396         assert("The given row index is out of bounds!" && (row >= 0 && row < _row_count));
397         assert("The given column index is out of bounds!" &&
398                (column >= 0 && column <= row));
399         return _data[row][column];
400     }
401     // get dimension
402     template <typename T>
403     inline int TriangularMatrix<T>::rowCount() const
404     {
405         return _row_count;
406     }
407     // set dimension
408     template <typename T>
409     bool TriangularMatrix<T>::resizeRows(int row_count)
410     {
411         assert("The row count of a triangular matrix should be a non-negative "
412                "integer!" && row_count >= 0);
413         if (row_count < 0)
414         {
415             return false;
416         }
417         if (_row_count != row_count)
```


© Ágoston Róth, 2018

```
418         {
419             _data.resize(row_count);
420             for (int r = _row_count; r < row_count; r++)
421             {
422                 _data[r].resize(r + 1);
423             }
424             _row_count = row_count;
425         }
426         return true;
427     }
428     // clone function required by smart pointers based on the deep copy ownership policy
429     template <typename T>
430     TriangularMatrix<T>* TriangularMatrix<T>::clone() const
431     {
432         return new (std::nothrow) TriangularMatrix<T>(*this);
433     }
434     // destructor
435     template <typename T>
436     TriangularMatrix<T>::~TriangularMatrix()
437     {
438         _row_count = 0;
439         _data.clear();
440     }
441     // Definitions of overloaded and templated input/output from/to stream operators:
442     // output to stream
443     template <typename T>
444     std::ostream& operator <<(std::ostream& lhs, const Matrix<T>& rhs)
445     {
446         lhs << rhs._row_count << " " << rhs._column_count << std::endl;
447         for (int r = 0; r < rhs._row_count; r++)
448         {
449             for (int c = 0, offset = r * rhs._column_count; c < rhs._column_count; c++)
450             {
451                 lhs << rhs._data[offset + c] << " ";
452             }
453             lhs << std::endl;
454         }
455         return lhs;
456     }
457     // input from stream
458     template <typename T>
459     std::istream& operator >>(std::istream& lhs, Matrix<T>& rhs)
460     {
461         lhs >> rhs._row_count >> rhs._column_count;
462         int size = rhs._row_count * rhs._column_count;
463         rhs._data.resize(size);
464         for (int i = 0; i < size; i++)
465         {
466             lhs >> rhs._data[i];
467         }
468         return lhs;
469     }
470     // output to stream
471     template <typename T>
472     std::ostream& operator <<(std::ostream& lhs, const TriangularMatrix<T>& rhs)
473     {
474         lhs << rhs._row_count << std::endl;
```


© Ágoston Róth, 2018

```
475         for (int r = 0; r < rhs._row_count; r++)
476         {
477             for (int c = 0; c <= r; c++)
478             {
479                 lhs << rhs._data[r][c] << " ";
480             }
481             lhs << std::endl;
482         }
483         return lhs;
484     }
485     // input from stream
486     template <typename T>
487     std::istream& operator >>(std::istream& lhs, TriangularMatrix<T>& rhs)
488     {
489         lhs >> rhs._row_count;
490         rhs._data.resize(rhs._row_count);
491         for (int r = 0; r < rhs._row_count; r++)
492         {
493             rhs._data[r].resize(r + 1);
494             for (int c = 0; c <= r; r++)
495             {
496                 lhs >> rhs._data[r][c];
497             }
498         }
499         return lhs;
500     }
501 }
502 #endif // MATRICES_H
```

Triangular matrices will be used to store either all (mixed) partial derivatives of a smooth surface up to a specified maximum order of differentiation, or Pascal triangles of binomial coefficients that are required for the evaluation of differentiation formulas which are based on the application of the Leibniz rule. The definition and implementation of Pascal triangles can be found in Listings 2.22/74 and 2.23/74, respectively.

Listing 2.22. Pascal triangles of binomial coefficients (**Core/Math/PascalTriangles.h**)

```
1  #ifndef PASCALTRIANGLES_H
2  #define PASCALTRIANGLES_H
3  #include "Matrices.h"
4  namespace cagd
5  {
6      class PascalTriangle: public TriangularMatrix<double>
7      {
8      public:
9          // default/special constructor
10         PascalTriangle(int order = 0);
11         // redeclared virtual resizing method
12         bool resizeRows(int row_count);
13     };
14 }
15 #endif // PASCALTRIANGLES_H
```

Listing 2.23. Pascal triangles of binomial coefficients (**Core/Math/PascalTriangles.cpp**)

```
1  #include "PascalTriangles.h"
```


© Ágoston Róth, 2018

```
2  namespace cagd
3  {
4      // default/special constructor
5      PascalTriangle::PascalTriangle(int order): TriangularMatrix<double>(order + 1)
6      {
7          _data[0][0] = 1.0;
8          for (int k = 1; k <= order; k++)
9          {
10             _data[k][0] = _data[k][k] = 1.0;
11             #pragma omp parallel for
12             for (int l = 1; l <= k / 2; l++)
13             {
14                 _data[k][l]     = _data[k - 1][l - 1] + _data[k - 1][l];
15                 _data[k][k - l] = _data[k][l];
16             }
17         }
18     }
19     // redefined virtual resizing method
20     bool PascalTriangle::resizeRows(int row_count)
21     {
22         int old_row_count = _row_count;
23         if (!TriangularMatrix::resizeRows(row_count))
24         {
25             return false;
26         }
27         for (int k = old_row_count; k < row_count; k++)
28         {
29             _data[k][0] = _data[k][k] = 1.0;
30             #pragma omp parallel for
31             for (int l = 1; l <= k / 2; l++)
32             {
33                 _data[k][l]     = _data[k - 1][l - 1] + _data[k - 1][l];
34                 _data[k][k - l] = _data[k][l];
35             }
36         }
37         return true;
38     }
39 }
```

2.9.2 Real matrices and decompositions

We also provide real matrices (RealMatrix: public Matrix<double>) and data structures (like PLUDecomposition, FactorizedUnpivotedLUDecomposition and SVDecomposition) for pivoted and non-pivoted Doolittle-type LU decompositions and for singular value decompositions as well.

Real matrices provide multi-threaded arithmetical operators for matrix related operations, while using different matrix decompositions we are able:

- to solve by multi-threading linear systems of equations described by template matrices that may appear not only in traditional real cases, but in case of curve and surface interpolation problems as well;
- to study the condition numbers of possibly ill-conditioned systems of linear of equations;
- to construct the normalized B-basis of the underlying EC spaces.

Real matrices

Diagrams of the class RealMatrix and of some auxiliary real row and column matrix related arithmetical binary operators are illustrated in Fig. 2.15/76, while their definitions and implementations can be found in Listings 2.24/75 and 2.25/78, respectively.

© Ágoston Róth, 2018

RealMatrix: public Matrix<double>

```
+ <<explicit>> RealMatrix(row_count: int = 1, column_count: int = 1)
+ <<const>> operator +(rhs: const RealMatrix&): const RealMatrix
+ <<const>> operator -(rhs: const RealMatrix&): const RealMatrix
+ <<const>> operator *(rhs: const RealMatrix&): const RealMatrix
+ <<const>> operator *(rhs: const double&): const RealMatrix
+ <<const>> operator *(rhs: const ColumnMatrix<double>&): const RealMatrix
+ <<const>> operator /(rhs: const double&): const RealMatrix
+ operator +=(rhs: const RealMatrix&): RealMatrix&
+ operator -=(rhs: const RealMatrix&): RealMatrix&
+ operator *=(rhs: const RealMatrix&): RealMatrix&
+ operator *=(rhs: const double&): RealMatrix&
+ operator *=(rhs: const ColumnMatrix<double>&): RealMatrix&
+ operator /=(rhs: const double&): RealMatrix&
+ loadNullMatrix(): void
+ loadIdentityMatrix(): void
+ <<const>> transpose(): RealMatrix
+ <<const>> isSquare(): bool
+ <<const>> isRowMatrix(): bool
+ <<const>> isColumnMatrix(): bool
+ <<const>> clone(): RealMatrix*
  <<friend>> operator * (lhs: const double&, rhs: const RealMatrix&): const RealMatrix
  <<friend>> operator * (lhs: const RowMatrix<double>&, rhs: const RealMatrix&): const RealMatrix
  <<friend>> operator * (lhs: const ColumnMatrix<double>&, rhs: const RowMatrix<double>&): const RealMatrix
  <<friend>> operator *=(lhs: RowMatrix<double>&, rhs: const RealMatrix&): RowMatrix<double>&
```

Auxiliary real row and column matrix related arithmetical binary operators

```
operator * (lhs: const RowMatrix<double>&, rhs: const ColumnMatrix<double>&): double
operator + (lhs: const RowMatrix<double>&, rhs: const RowMatrix<double>&): const RowMatrix<double>
operator - (lhs: const RowMatrix<double>&, rhs: const RowMatrix<double>&): const RowMatrix<double>
operator * (lhs: const RowMatrix<double>&, rhs: const double&): const RowMatrix<double>
operator * (lhs: const double&, rhs: const RowMatrix<double>&): const RowMatrix<double>
operator +=(lhs: RowMatrix<double>&, rhs: const RowMatrix<double>&): RowMatrix<double>&
operator -=(lhs: RowMatrix<double>&, rhs: const RowMatrix<double>&): RowMatrix<double>&
operator *=(lhs: RowMatrix<double>&, rhs: const double&): RowMatrix<double>&
operator /=(lhs: RowMatrix<double>&, rhs: const double&): RowMatrix<double>&
operator + (lhs: const ColumnMatrix<double>&, rhs: const ColumnMatrix<double>&): const ColumnMatrix<double>
operator - (lhs: const ColumnMatrix<double>&, rhs: const ColumnMatrix<double>&): const ColumnMatrix<double>
operator * (lhs: const ColumnMatrix<double>&, rhs: const double&): const ColumnMatrix<double>
operator * (lhs: const double&, rhs: const ColumnMatrix<double>&): const ColumnMatrix<double>
operator +=(lhs: ColumnMatrix<double>&, rhs: const ColumnMatrix<double>&): ColumnMatrix<double>&
operator -=(lhs: ColumnMatrix<double>&, rhs: const ColumnMatrix<double>&): ColumnMatrix<double>&
operator *=(lhs: ColumnMatrix<double>&, rhs: const double&): ColumnMatrix<double>&
operator /=(lhs: ColumnMatrix<double>&, rhs: const double&): ColumnMatrix<double>&
```

Fig. 2.15: Diagrams of real matrices and of some auxiliary real row and column matrix related arithmetical binary operators

Listing 2.24. Real matrices (**Core/Math/RealMatrices.h**)

```
1  #ifndef REALMATRICES_H
2  #define REALMATRICES_H

3  #include "Matrices.h"

4  namespace cagd
5  {
6      // General (rectangular) real matrices.
7      class RealMatrix: public Matrix<double>
8      {
9      public:
10         // default/special constructor
11         explicit RealMatrix(int row_count = 1, int column_count = 1);

12         // matrix addition
13         const RealMatrix operator +(const RealMatrix &rhs) const;

14         // matrix subtraction
15         const RealMatrix operator -(const RealMatrix &rhs) const;

16         // matrix multiplication
17         const RealMatrix operator *(const RealMatrix &rhs) const;

18         // multiplication by scalar from right
```


© Ágoston Róth, 2018

```
19          const RealMatrix operator *(const double &rhs) const;

20          // multiplication by scalar from left
21          friend const RealMatrix operator *(const double &lhs, const RealMatrix &rhs);

22          // multiplication by real column vector from right
23          const RealMatrix operator *(const ColumnMatrix<double> &rhs) const;

24          // multiplication by real row vector from left
25          friend const RealMatrix operator *(const RowMatrix<double> &lhs,
26                                             const RealMatrix &rhs);

27          // multiplication of a real column matrix by a real row matrix from right
28          friend const RealMatrix operator *(const ColumnMatrix<double> &lhs,
29                                             const RowMatrix<double> &rhs);

30          // multiplication of a real row matrix by a real matrix from right
31          friend RowMatrix<double>& operator *=(RowMatrix<double> &lhs,
32                                                const RealMatrix &rhs);

33          // division by scalar from right
34          const RealMatrix operator /(const double &rhs) const;

35          // add to *this
36          RealMatrix& operator +=(const RealMatrix &rhs);

37          // subtract from *this
38          RealMatrix& operator -= (const RealMatrix &T);

39          // multiplicate *this by a real matrix
40          RealMatrix& operator *= (const RealMatrix &rhs);

41          // multiplicate *this by a constant
42          RealMatrix& operator *= (const double &rhs);

43          // multiplicate *this by a real column vector from right
44          RealMatrix& operator *=(const ColumnMatrix<double>& rhs);

45          // divide *this by a constant
46          RealMatrix& operator /= (const double &rhs);

47          // sets each matrix element to zero
48          void loadNullMatrix();

49          // loads a matrix with ones on the first main diagonal and zeros elsewhere
50          void loadIdentityMatrix();

51          // returns the transpose of the stored matrix
52          RealMatrix transpose() const;

53          // decides whether the matrix is a square one
54          bool isSquare() const;

55          // decides whether the matrix consists of a single row
56          bool isRowMatrix() const;

57          // decides whether the matrix consists of a single column
58          bool isColumnMatrix() const;

59          // redeclared clone function required by smart pointers based on the deep copy ownership policy
60          RealMatrix* clone() const;
61      };

62      // Auxiliary real row and column matrix related arithmetical binary operators.

63      double operator *(const RowMatrix<double> &lhs, const ColumnMatrix<double> &rhs);

64      const RowMatrix<double> operator +(
65          const RowMatrix<double> &lhs, const RowMatrix<double> &rhs);
66      const RowMatrix<double> operator -(
67          const RowMatrix<double> &lhs, const RowMatrix<double> &rhs);
68      const RowMatrix<double> operator *(const RowMatrix<double> &lhs, const double &rhs);
69      const RowMatrix<double> operator *(const double &lhs, const RowMatrix<double> &rhs);
```


© Ágoston Róth, 2018

```
70      RowMatrix<double>& operator +=(RowMatrix<double> &lhs, const RowMatrix<double> &rhs);
71      RowMatrix<double>& operator -=(RowMatrix<double> &lhs, const RowMatrix<double> &rhs);

72      RowMatrix<double>& operator *=(RowMatrix<double> &lhs, const double &rhs);
73      RowMatrix<double>& operator /=(RowMatrix<double> &lhs, const double &rhs);

74      const ColumnMatrix<double> operator +(
75          const ColumnMatrix<double> &lhs, const ColumnMatrix<double> &rhs);
76      const ColumnMatrix<double> operator -(
77          const ColumnMatrix<double> &lhs, const ColumnMatrix<double> &rhs);
78      const ColumnMatrix<double> operator *(
79          const ColumnMatrix<double> &lhs, const double &rhs);
80      const ColumnMatrix<double> operator *(
81          const double &lhs, const ColumnMatrix<double> &rhs);

82      ColumnMatrix<double>& operator +=(
83          ColumnMatrix<double> &lhs, const ColumnMatrix<double> &rhs);
84      ColumnMatrix<double>& operator -=(
85          ColumnMatrix<double> &lhs, const ColumnMatrix<double> &rhs);

86      ColumnMatrix<double>& operator *=(ColumnMatrix<double> &lhs, const double &rhs);
87      ColumnMatrix<double>& operator /=(ColumnMatrix<double> &lhs, const double &rhs);
88  }

89  #endif // REALMATRICES_H
```

Listing 2.25. Real matrices (**Core/Math/RealMatrices.cpp**)

```
1   #include "RealMatrices.h"

2   #include "../Exceptions.h"

3   #include <new>
4   using namespace std;

5   namespace cagd
6   {
7       // Implementation of general (rectangular) real matrices.

8       // default/special constructor
9       RealMatrix::RealMatrix(int row_count, int column_count):
10          Matrix<double>(row_count, column_count)
11      {
12      }

13      // matrix addition
14      const RealMatrix RealMatrix::operator +(const RealMatrix &rhs) const
15      {
16          if (_row_count != rhs._row_count || _column_count != rhs._column_count)
17          {
18              throw Exception("RealMatrix::operator +(const RealMatrix &rhs) : "
19                              "incompatible matrix dimensions!");
20          }

21          RealMatrix result(*this);

22          #pragma omp parallel for
23          for (int i = 0; i < _row_count * _column_count; i++)
24          {
25              result._data[i] += rhs._data[i];
26          }

27          return result;
28      }

29      // matrix subtraction
30      const RealMatrix RealMatrix::operator -(const RealMatrix &rhs) const
31      {
32          if (_row_count != rhs._row_count || _column_count != rhs._column_count)
33          {
34              throw Exception("RealMatrix::operator -(const RealMatrix &rhs) : "
35                              "incompatible matrix dimensions!");
```


© Ágoston Róth, 2018

```
36          }
37          RealMatrix result(*this);
38          #pragma omp parallel for
39          for (int i = 0; i < _row_count * _column_count; i++)
40          {
41              result._data[i] -= rhs._data[i];
42          }
43          return result;
44      }
45      // matrix multiplication
46      const RealMatrix RealMatrix::operator *(const RealMatrix &rhs) const
47      {
48          if (_column_count != rhs._row_count)
49          {
50              throw Exception("RealMatrix::operator *(const RealMatrix &rhs) : "
51                              "incompatible inner matrix dimensions!");
52          }
53          RealMatrix result(_row_count, rhs._column_count);
54          #pragma omp parallel for
55          for (int r_c = 0; r_c < _row_count * rhs._column_count; r_c++)
56          {
57              int r             = r_c / rhs._column_count;
58              int c             = r_c % rhs._column_count;
59              int offset        = r * _column_count;
60              int result_offset = r * rhs._column_count + c;
61              for (int i = 0; i < _column_count; i++)
62              {
63                  result._data[result_offset] += _data[offset + i] *
64                                                 rhs._data[i * rhs._column_count + c];
65              }
66          }
67          return result;
68      }
69      // multiplication by scalar from right
70      const RealMatrix RealMatrix::operator *(const double &rhs) const
71      {
72          RealMatrix result(*this);
73          #pragma omp parallel for
74          for (int i = 0; i < _row_count * _column_count; i++)
75          {
76              result._data[i] *= rhs;
77          }
78          return result;
79      }
80      // multiplication by scalar from left
81      const RealMatrix operator *(const double &lhs, const RealMatrix &rhs)
82      {
83          RealMatrix result(rhs);
84          #pragma omp parallel for
85          for (int i = 0; i < result._row_count * result._column_count; i++)
86          {
87              result._data[i] *= lhs;
88          }
89          return result;
90      }
91      // multiplication by real column vector from right
92      const RealMatrix RealMatrix::operator *(const ColumnMatrix<double> &rhs) const
93      {
94          if (_column_count != rhs.rowCount())
```


© Ágoston Róth, 2018

```
95          {
96              throw Exception("RealMatrix::operator *(const ColumnMatrix<double> &rhs): "
97                              "incompatible inner matrix dimensions!");
98          }
99          RealMatrix result(_row_count, 1);
100         #pragma omp parallel for
101         for (int r = 0; r < _row_count; r++)
102         {
103             for (int c = 0, offset = r * _column_count; c < _column_count; c++)
104             {
105                 result._data[r] += _data[offset + c] * rhs[c];
106             }
107         }
108         return result;
109     }
110     // multiplication by real row vector from left
111     const RealMatrix operator *(const RowMatrix<double> &lhs, const RealMatrix &rhs)
112     {
113         if (lhs.columnCount() != rhs._row_count)
114         {
115             throw Exception("operator *(const RowMatrix<double> &lhs, "
116                             "const RealMatrix &rhs): "
117                             "incompatible inner matrix dimensions!");
118         }
119         RealMatrix result(1, rhs._column_count);
120         #pragma omp parallel for
121         for (int c = 0; c < rhs._column_count; c++)
122         {
123             for (int r = 0; r < rhs._row_count; r++)
124             {
125                 result._data[c] += lhs[r] * rhs._data[r * rhs._column_count + c];
126             }
127         }
128         return result;
129     }
130     // multiplication of a real column matrix by a real row matrix from right
131     const RealMatrix operator *(
132         const ColumnMatrix<double> &lhs, const RowMatrix<double> &rhs)
133     {
134         RealMatrix result(lhs.rowCount(), rhs.columnCount());
135         #pragma omp parallel for
136         for (int r = 0; r < result._row_count; r++)
137         {
138             result.setRow(r, rhs);
139             double multiplier = lhs[r];
140             for (int c = 0, offset = r * result._column_count; c < result._column_count;
141                  c++)
142             {
143                 result._data[offset + c] *= multiplier;
144             }
145         }
146         return result;
147     }
148     // multiplication of a real row matrix by a real matrix from right
149     RowMatrix<double>& operator *=(RowMatrix<double> &lhs, const RealMatrix &rhs)
150     {
151         if (lhs.columnCount() != rhs._row_count)
152         {
153             throw Exception("operator *=(RowMatrix<double> &lhs, "
154                             "const RealMatrix &rhs): "
155                             "incompatible inner matrix dimensions!");
```


© Ágoston Róth, 2018

```
156         }
157         std::vector<double> result(rhs._column_count);
158         #pragma omp parallel for
159         for (int i = 0; i < rhs._column_count; i++)
160         {
161             for (int c = 0; c < lhs.columnCount(); c++)
162             {
163                 result[i] += lhs[c] * rhs._data[c * rhs._column_count + i];
164             }
165         }
166         lhs.resizeColumns(rhs._column_count);
167         #pragma omp parallel for
168         for (int i = 0; i < lhs.columnCount(); i++)
169         {
170             lhs[i] = result[i];
171         }
172         return lhs;
173     }
174     // division by scalar from right
175     const RealMatrix RealMatrix::operator /(const double &rhs) const
176     {
177         if (!rhs)
178         {
179             throw Exception("RealMatrix::operator /(const double &rhs): "
180                             "division by zero!");
181         }
182         RealMatrix result(*this);
183         #pragma omp parallel for
184         for (int i = 0; i < _row_count * _column_count; i++)
185         {
186             result._data[i] /= rhs;
187         }
188         return result;
189     }
190     // add to *this
191     RealMatrix& RealMatrix::operator +=(const RealMatrix &rhs)
192     {
193         if (_row_count != rhs._row_count || _column_count != rhs._column_count)
194         {
195             throw Exception("RealMatrix::operator +=(const RealMatrix &rhs) : "
196                             "incompatible matrix dimensions!");
197         }
198         #pragma omp parallel for
199         for (int i = 0; i < _row_count * _column_count; i++)
200         {
201             _data[i] += rhs._data[i];
202         }
203         return *this;
204     }
205     // subtract from *this
206     RealMatrix& RealMatrix::operator -=(const RealMatrix &rhs)
207     {
208         if (_row_count != rhs._row_count || _column_count != rhs._column_count)
209         {
210             throw Exception("RealMatrix::operator -=(const RealMatrix &rhs) : "
211                             "incompatible matrix dimensions!");
212         }
213         #pragma omp parallel for
214         for (int i = 0; i < _row_count * _column_count; i++)
215         {
```


© Ágoston Róth, 2018

```
216             _data[i] -= rhs._data[i];
217         }
218         return *this;
219     }
220     // multiplicate *this by a real matrix
221     RealMatrix& RealMatrix::operator *=(const RealMatrix &rhs)
222     {
223         if (_column_count != rhs._row_count)
224         {
225             throw Exception("RealMatrix::operator *=(const RealMatrix &rhs) : "
226                             "incompatible inner matrix dimensions!");
227         }
228         RealMatrix result(_row_count, rhs._column_count);
229         #pragma omp parallel for
230         for (int r_c = 0; r_c < _row_count * rhs._column_count; r_c++)
231         {
232             int r             = r_c / rhs._column_count;
233             int c             = r_c % rhs._column_count;
234             int offset        = r * _column_count;
235             int result_offset = r * rhs._column_count + c;
236             for (int i = 0; i < _column_count; i++)
237             {
238                 result._data[result_offset] += _data[offset + i] *
239                                                rhs._data[i * rhs._column_count + c];
240             }
241         }
242         _data.swap(result._data);
243         _column_count = result._column_count;
244         return *this;
245     }
246     // multiplicate *this by a constant
247     RealMatrix& RealMatrix::operator *=(const double &rhs)
248     {
249         #pragma omp parallel for
250         for (int i = 0; i < _row_count * _column_count; i++)
251         {
252             _data[i] *= rhs;
253         }
254         return *this;
255     }
256     // multiplicate *this by a real column vector from right
257     RealMatrix& RealMatrix::operator *=(const ColumnMatrix<double>& rhs)
258     {
259         if (_column_count != rhs.rowCount())
260         {
261             throw Exception("RealMatrix::operator *=(const ColumnMatrix<double> &rhs): "
262                             "incompatible inner matrix dimensions!");
263         }
264         vector<double> result(_row_count);
265         #pragma omp parallel for
266         for (int r = 0; r < _row_count; r++)
267         {
268             for (int c = 0, offset = r * _column_count; c < _column_count; c++)
269             {
270                 result[r] += _data[offset + c] * rhs[c];
271             }
272         }
273         _column_count = 1;
274         _data.swap(result);
```


© Ágoston Róth, 2018

```
275         return *this;
276     }
277     // divide *this by a constant
278     RealMatrix& RealMatrix::operator /=(const double &rhs)
279     {
280         if (!rhs)
281         {
282             throw Exception("RealMatrix::operator /=(const double &rhs) : "
283                             "division by zero!");
284         }
285         #pragma omp parallel for
286         for (int i = 0; i < _row_count * _column_count; i++)
287         {
288             _data[i] /= rhs;
289         }
290         return *this;
291     }
292     // sets each matrix element to zero
293     void RealMatrix::loadNullMatrix()
294     {
295         #pragma omp parallel for
296         for (int i = 0; i < _row_count * _column_count; i++)
297         {
298             _data[i] = 0.0;
299         }
300     }
301     // loads a matrix with ones on the first main diagonal and zeros elsewhere
302     void RealMatrix::loadIdentityMatrix()
303     {
304         #pragma omp parallel for
305         for (int i = 0; i < _row_count * _column_count; i++)
306         {
307             _data[i] = 0.0;
308         }
309         int step = _column_count + 1;
310         #pragma omp parallel for
311         for (int offset = 0; offset < min(_row_count, _column_count) * _column_count;
312              offset += step)
313         {
314             _data[offset] = 1.0;
315         }
316     }
317     // returns the transpose of the stored matrix
318     RealMatrix RealMatrix::transpose() const
319     {
320         RealMatrix result(_column_count, _row_count);
321         #pragma omp parallel for
322         for (int r_c = 0; r_c < _row_count * _column_count; r_c++)
323         {
324             int r = r_c / _column_count;
325             int c = r_c % _column_count;
326             result._data[c * _row_count + r] = _data[r_c];
327         }
328         return result;
329     }
330     // decides whether the matrix is a square one
331     bool RealMatrix::isSquare() const
332     {
333         return (_row_count == _column_count);
334     }
335     // decides whether the matrix consists of a single row
```


© Ágoston Róth, 2018

```
336     bool RealMatrix::isRowMatrix() const
337     {
338         return (_row_count == 1);
339     }

340     // decides whether the matrix consists of a single column
341     bool RealMatrix::isColumnMatrix() const
342     {
343         return (_column_count == 1);
344     }

345     // redefined clone function required by smart pointers based on the deep copy ownership policy
346     RealMatrix* RealMatrix::clone() const
347     {
348         return new (nothrow) RealMatrix(*this);
349     }

350     // Implementation of auxiliary real row and column matrix related arithmetical binary operators.

351     double operator *(const RowMatrix<double> &lhs, const ColumnMatrix<double> &rhs)
352     {
353         if (lhs.columnCount() != rhs.rowCount())
354         {
355             throw Exception("operator *(const RowMatrix<double> &lhs, "
356                             "const ColumnMatrix<double> &rhs) : "
357                             "incompatible inner matrix dimensions!");
358         }

359         double result = 0.0;

360         #pragma omp parallel for reduction(+:result)
361         for (int i = 0; i < lhs.columnCount(); i++)
362         {
363             result += lhs[i] * rhs[i];
364         }

365         return result;
366     }

367     const RowMatrix<double> operator +(
368         const RowMatrix<double> &lhs, const RowMatrix<double> &rhs)
369     {
370         if (lhs.columnCount() != rhs.columnCount())
371         {
372             throw Exception("operator +(const RowMatrix<double> &lhs, "
373                             "const RowMatrix<double> &rhs) : "
374                             "incompatible matrix dimensions!");
375         }

376         RowMatrix<double> result(lhs);

377         #pragma omp parallel for
378         for (int i = 0; i < result.columnCount(); i++)
379         {
380             result[i] += rhs[i];
381         }

382         return result;
383     }

384     const RowMatrix<double> operator -(
385         const RowMatrix<double> &lhs, const RowMatrix<double> &rhs)
386     {
387         if (lhs.columnCount() != rhs.columnCount())
388         {
389             throw Exception("operator -(const RowMatrix<double> &lhs, "
390                             "const RowMatrix<double> &rhs) : "
391                             "incompatible matrix dimensions!");
392         }

393         RowMatrix<double> result(lhs);

394         #pragma omp parallel for
395         for (int i = 0; i < result.columnCount(); i++)
```


© Ágoston Róth, 2018

```
396         {
397             result[i] -= rhs[i];
398         }
399 
399         return result;
400     }
401     const RowMatrix<double> operator *(const RowMatrix<double> &lhs, const double &rhs)
402     {
403         RowMatrix<double> result(lhs);
404 
404         #pragma omp parallel for
405         for (int i = 0; i < result.columnCount(); i++)
406         {
407             result[i] *= rhs;
408         }
409 
409         return result;
410     }
411     const RowMatrix<double> operator *(const double &lhs, const RowMatrix<double> &rhs)
412     {
413         RowMatrix<double> result(rhs);
414 
414         #pragma omp parallel for
415         for (int i = 0; i < result.columnCount(); i++)
416         {
417             result[i] *= lhs;
418         }
419 
419         return result;
420     }
421     RowMatrix<double>& operator +=(RowMatrix<double> &lhs, const RowMatrix<double> &rhs)
422     {
423         if (lhs.columnCount() != rhs.columnCount())
424         {
425             throw Exception("operator +=(RowMatrix<double> &lhs, "
426                             "const RowMatrix<double> &rhs) : "
427                             "incompatible matrix dimensions!");
428         }
429 
429         #pragma omp parallel for
430         for (int i = 0; i < lhs.columnCount(); i++)
431         {
432             lhs[i] += rhs[i];
433         }
434 
434         return lhs;
435     }
436     RowMatrix<double>& operator -=(RowMatrix<double> &lhs, const RowMatrix<double> &rhs)
437     {
438         if (lhs.columnCount() != rhs.columnCount())
439         {
440             throw Exception("operator -=(RowMatrix<double> &lhs, "
441                             "const RowMatrix<double> &rhs) : "
442                             "incompatible matrix dimensions!");
443         }
444 
444         #pragma omp parallel for
445         for (int i = 0; i < lhs.columnCount(); i++)
446         {
447             lhs[i] -= rhs[i];
448         }
449 
449         return lhs;
450     }
451     RowMatrix<double>& operator *=(RowMatrix<double> &lhs, const double &rhs)
452     {
453         #pragma omp parallel for
454         for (int i = 0; i < lhs.columnCount(); i++)
455         {
```


© Ágoston Róth, 2018

```
456             lhs[i] *= rhs;
457         }
458         return lhs;
459     }
460     RowMatrix<double>& operator /=(RowMatrix<double> &lhs, const double &rhs)
461     {
462         #pragma omp parallel for
463         for (int i = 0; i < lhs.columnCount(); i++)
464         {
465             lhs[i] /= rhs;
466         }
467         return lhs;
468     }
469     const ColumnMatrix<double> operator +(
470         const ColumnMatrix<double> &lhs, const ColumnMatrix<double> &rhs)
471     {
472         if (lhs.rowCount() != rhs.rowCount())
473         {
474             throw Exception("operator +(const ColumnMatrix<double> &lhs, "
475                             "const ColumnMatrix<double> &rhs) : "
476                             "incompatible matrix dimensions!");
477         }
478         ColumnMatrix<double> result(lhs);
479         #pragma omp parallel for
480         for (int i = 0; i < result.rowCount(); i++)
481         {
482             result[i] += rhs[i];
483         }
484         return result;
485     }
486     const ColumnMatrix<double> operator -(
487         const ColumnMatrix<double> &lhs, const ColumnMatrix<double> &rhs)
488     {
489         if (lhs.rowCount() != rhs.rowCount())
490         {
491             throw Exception("operator -(const ColumnMatrix<double> &lhs, "
492                             "const ColumnMatrix<double> &rhs) : "
493                             "incompatible matrix dimensions!");
494         }
495         ColumnMatrix<double> result(lhs);
496         #pragma omp parallel for
497         for (int i = 0; i < result.rowCount(); i++)
498         {
499             result[i] -= rhs[i];
500         }
501         return result;
502     }
503     const ColumnMatrix<double> operator *(
504         const ColumnMatrix<double> &lhs, const double &rhs)
505     {
506         ColumnMatrix<double> result(lhs);
507         #pragma omp parallel for
508         for (int i = 0; i < result.rowCount(); i++)
509         {
510             result[i] *= rhs;
511         }
512         return result;
513     }
514     const ColumnMatrix<double> operator *(
```


© Ágoston Róth, 2018

```
515         const double &lhs, const ColumnMatrix<double> &rhs)
516     {
517         ColumnMatrix<double> result(rhs);
518         #pragma omp parallel for
519         for (int i = 0; i < result.rowCount(); i++)
520         {
521             result[i] *= lhs;
522         }
523         return result;
524     }
525     ColumnMatrix<double>& operator +=(
526         ColumnMatrix<double> &lhs, const ColumnMatrix<double> &rhs)
527     {
528         if (lhs.rowCount() != rhs.rowCount())
529         {
530             throw Exception("operator +=(ColumnMatrix<double> &lhs, "
531                             "const ColumnMatrix<double> &rhs) : "
532                             "incompatible matrix dimensions!");
533         }
534         #pragma omp parallel for
535         for (int i = 0; i < lhs.rowCount(); i++)
536         {
537             lhs[i] += rhs[i];
538         }
539         return lhs;
540     }
541     ColumnMatrix<double>& operator -=(
542         ColumnMatrix<double> &lhs, const ColumnMatrix<double> &rhs)
543     {
544         if (lhs.rowCount() != rhs.rowCount())
545         {
546             throw Exception("operator -=(ColumnMatrix<double> &lhs, "
547                             "const ColumnMatrix<double> &rhs) : "
548                             "incompatible matrix dimensions!");
549         }
550         #pragma omp parallel for
551         for (int i = 0; i < lhs.rowCount(); i++)
552         {
553             lhs[i] -= rhs[i];
554         }
555         return lhs;
556     }
557     ColumnMatrix<double>& operator *=(ColumnMatrix<double> &lhs, const double &rhs)
558     {
559         #pragma omp parallel for
560         for (int i = 0; i < lhs.rowCount(); i++)
561         {
562             lhs[i] *= rhs;
563         }
564         return lhs;
565     }
566     ColumnMatrix<double>& operator /=(ColumnMatrix<double> &lhs, const double &rhs)
567     {
568         #pragma omp parallel for
569         for (int i = 0; i < lhs.rowCount(); i++)
570         {
571             lhs[i] /= rhs;
572         }
573         return lhs;
574     }
575 }
```


© Ágoston Róth, 2018

LU and singular value decompositions

Diagrams of classes PLUDecomposition, FactorizedUnpivotedLUDecomposition and SVDecomposition) are illustrated in Fig. 2.16/88, while their declarations and implementations can be found in Listings 2.26/88 and 2.27/97, respectively. These parts of the proposed function library are based on [Press et al., 2007].



Fig. 2.16: Diagrams of used real matrix decompositions

Listing 2.26. Real matrix decompositions (**Core/Math/RealMatrixDecompositions.h**)

```
#ifndef REALMATRIXDECOMPOSITIONS_H
#define REALMATRIXDECOMPOSITIONS_H

#include "../Exceptions.h"
#include "Matrices.h"
#include "RealMatrices.h"
#include <cmath>
#include <limits>

namespace cagd
{
    // Generates and stores the pivoted Doolittle-type PLU decomposition of a given regular square matrix.
    class PLUDecomposition
    {
```


© Ágoston Róth, 2018

```
13      private:
14          bool            _decomposition_is_done;
15          double          _determinant;
16          RowMatrix<int> _P;
17          RealMatrix      _LU;

18      public:
19          // Tries to determine the pivoted Doolittle-type LU decomposition of the given real
20          // square matrix M. If M is singular then its LU factorization cannot be determined.
21          // If the decomposition is unsuccessful, the boolean variable _decomposition_is_done
22          // will be set to false, and if the input variable generate_exceptions is set to true,
23          // then the method will also generate an exception with a meaningful reason.
24          PLUDecomposition(const RealMatrix &M, bool generate_exceptions = false);

25          // Returns either the successful or the unsuccessful state of the decomposition.
26          bool    isCorrect() const;

27          // If the decomposition was successful the method will return the product of the diagonal entries of
28          // the upper triangular matrix U (i.e., the determinant of M), otherwise it will return zero.
29          double determinant() const;

30          // clone function required by smart pointers based on the deep copy ownership policy
31          PLUDecomposition* clone() const;

32          // Using multi-threading, tries to solve systems of linear equations of type M · X = B, where M is a regular
33          // square matrix, while B can be either a column or rectangular matrix of multiple columns with the same
34          // row dimension as M. In case of success, the solution(s) will be stored in X.
35          // If the input boolean parameter represent_solutions_as_columns is false, then the column dimension of
36          // the transpose of B should be equal to the row dimension of M.
37          // Note that, T can be either double or Cartesian3, or any other custom type which has similar overloaded
38          // mathematical and boolean operators.
39          template <typename T>
40          bool solveLinearSystem(const Matrix<T> &B, Matrix<T> &X,
41                                 bool represent_solutions_as_columns = true);
42      };

43      // Generates and stores the factors L and U of the unpivoted Doolittle-type LU decomposition of a
44      // given regular matrix.
45      class FactorizedUnpivotedLUDecomposition
46      {
47      private:
48          bool        _decomposition_is_done;
49          double      _determinant;
50          RealMatrix _L, _U;

51      public:
52          // The constructor tries to determine the unpivoted Doolittle-type LU decomposition of the given real
53          // square matrix M. If M is singular then its LU factorization cannot be determined.
54          // If the decomposition is unsuccessful, the boolean variable _decomposition_is_done will be
55          // set to false, and if the input variable generate_exceptions is set to true, then the method will
56          // also generate an exception with a meaningful reason.
57          FactorizedUnpivotedLUDecomposition(const RealMatrix &M,
58                                             bool generate_exceptions = false);

59          // Returns either the successful or the unsuccessful state of the decomposition.
60          bool isCorrect() const;

61          // If the decomposition was successful the method will return the product of the diagonal entries of
62          // the upper triangular matrix U (i.e., the determinant of M), otherwise it will return zero.
63          double determinant() const;

64          // Returns a constant reference to the lower triangular matrix L.
65          const RealMatrix& L() const;

66          // Returns a constant reference to the upper triangular matrix U.
67          const RealMatrix& U() const;

68          // clone function required by smart pointers based on the deep copy ownership policy
69          FactorizedUnpivotedLUDecomposition* clone() const;


70          // If M is a regular real square matrix (i.e., the decomposition M = L · U can be constructed),
71          // then the next two methods can be used to solve the system M · X = B of linear equations in
72          // two steps: at first one has to solve the system L · Y = B, then one should solve the system
```


© Ágoston Róth, 2018

```
73          // U · X = Y.
74          // Using multi-threading, tries to solve systems of linear equations of type L · Y = B, where B can
75          // be either a column or rectangular matrix of multiple columns with the same row dimension as L.
76          // In case of success, the solution(s) will be stored in Y.
77          // If the input boolean parameter represent_solutions_as_columns is false, then the column dimension of
78          // the transpose of B should be equal to the row dimension of L.
79          // Note that, T can be either double or Cartesian3, or any other custom type which has similar overloaded
80          // mathematical and boolean operators.
81          template <typename T>
82          bool solveLLinearSystem(const Matrix<T> &B, Matrix<T> &Y,
83                                  bool represent_solutions_as_columns = true);
84          // Using multi-threading, tries to solve systems of linear equations of type U · X = Y, where Y can
85          // be either a column or rectangular matrix of multiple columns with the same row dimension as U.
86          // In case of success, the solution(s) will be stored in X.
87          // If the input boolean parameter represent_solutions_as_columns is false, then the column dimension of
88          // the transpose of Y should be equal to the row dimension of U.
89          // Note that, T can be either double or Cartesian3, or any other custom type which has similar overloaded
90          // mathematical and boolean operators.
91          template <typename T>
92          bool solveULinearSystem(const Matrix<T> &Y, Matrix<T> &X,
93                                  bool represent_solutions_as_columns = true);
94      };
95      // Generates and stores the singular value decomposition of a given real (not necessarily square) matrix M.
96      class SVDecomposition
97      {
98      private:
99          bool                _decomposition_is_done;
100         double              _product_of_singular_values;
101         RealMatrix          _U;
102         RowMatrix<double>   _S;
103         RealMatrix          _V;
104     public:
105         // Tries to determine the singular value decomposition of the given real matrix M.
106         // If the decomposition is unsuccessful, the boolean variable _decomposition_is_done
107         // will be set to false, and if the input variable generate_exceptions is set to true,
108         // then the method will also generate an exception with a meaningful reason.
109         SVDecomposition(const RealMatrix &M, bool generate_exceptions = false);
110         // Returns either the successful or the unsuccessful state of the decomposition.
111         bool    isCorrect() const;
112         // Returns the ratio of the largest and smalles singular values.
113         double conditionNumber() const;
114         // Returns the ratio of the smallest and largest singular values.
115         double reciprocalConditionNumber() const;
116         // Returns the product of the obtained singular values. If M is a regular real square matrix,
117         // then this product coincides with the determinant of M.
118         double productOfSingularValues() const;
119         // clone function required by smart pointers based on the deep copy ownership policy
120         SVDecomposition* clone() const;
121         // Using multi-threading, tries to solve systems of linear equations of type M · X = B, where M is not
122         // necessarily a real square matrix, while B can be either a column or rectangular matrix of multiple
123         // columns with the same row dimension as M. In case of success, the solution(s) will be stored in X.
124         // If the input boolean parameter represent_solutions_as_columns is false, then the column dimension of
125         // the transpose of B should be equal to the row dimension of M.
126         // Note that, T can be either double or Cartesian3, or any other custom type which has similar overloaded
127         // mathematical and boolean operators.
128         template <typename T>
129         bool solveLinearSystem(const Matrix<T> &B, Matrix<T> &X,
130                                bool represent_solutions_as_columns = true);
131     };
132     // Using multi-threading, tries to solve systems of linear equations of type M · X = B, where M is a regular
133     // square matrix, while B can be either a column or rectangular matrix of multiple columns with the same row
134     // dimension as M. In case of success, the solution(s) will be stored in X.
135     // If the input boolean parameter represent_solutions_as_columns is false, then the column dimension of
```


© Ágoston Róth, 2018

```
136     // the transpose of B should be equal to the row dimension of M.
137     // Note that, T can be either double or Cartesian3, or any other custom type which has similar overloaded
138     // mathematical and boolean operators.
139     template <typename T>
140     bool PLUDecomposition::solveLinearSystem(const Matrix<T> &B, Matrix<T> &X,
141                                              bool represent_solutions_as_columns)
142     {
143         if (!_decomposition_is_done)
144         {
145             return false;
146         }
147         if (represent_solutions_as_columns)
148         {
149             int dimension = _LU.columnCount();
150             if (B.rowCount() != dimension)
151             {
152                 return false;
153             }
154             X = B;
155             #pragma omp parallel for
156             for (int k = 0; k < B.columnCount(); k++)
157             {
158                 int ii = 0;
159                 for (int i = 0; i < dimension; i++)
160                 {
161                     int ip = _P[i];
162                     T sum  = X(ip, k);
163                     X(ip, k) = X(i, k);
164                     if (ii != 0)
165                     {
166                         for (int j = ii - 1; j < i; j++)
167                         {
168                             sum -= _LU(i, j) * X(j, k);
169                         }
170                     }
171                     else
172                     {
173                         if (sum != 0.0)
174                         {
175                             ii = i + 1;
176                         }
177                     }
178                     X(i, k) = sum;
179                 }
180                 for (int i = dimension - 1; i >= 0; i--)
181                 {
182                     T sum = X(i, k);
183                     for (int j = i + 1; j < dimension; j++)
184                     {
185                         sum -= _LU(i, j) * X(j, k);
186                     }
187                     X(i, k) = sum /= _LU(i, i);
188                 }
189             }
190         }
191         else
192         {
193             int dimension = _LU.rowCount();
194             if (B.columnCount() != dimension)
195             {
196                 return false;
197             }
198             X = B;
```


© Ágoston Róth, 2018

```
199             #pragma omp parallel for
200             for (int k = 0; k < B.rowCount(); k++)
201             {
202                 int ii = 0;

203                 for (int i = 0; i < dimension; i++)
204                 {
205                     int ip = _P[i];
206                     T sum  = X(k, ip);
207                     X(k, ip) = X(k, i);

208                     if (ii != 0)
209                     {
210                         for (int j = ii - 1; j < i; j++)
211                         {
212                             sum -= _LU(i, j) * X(k, j);
213                         }
214                     }
215                     else
216                     {
217                         if (sum != 0.0)
218                         {
219                             ii = i + 1;
220                         }
221                     }

222                     X(k, i) = sum;
223                 }

224                 for (int i = dimension - 1; i >= 0; i--)
225                 {
226                     T sum = X(k, i);

227                     for (int j = i + 1; j < dimension; j++)
228                     {
229                         sum -= _LU(i, j) * X(k, j);
230                     }

231                     X(k, i) = sum /= _LU(i, i);
232                 }
233             }
234         }

235         return true;
236     }

237     // If M is a regular real square matrix (i.e., the decomposition M = L · U can be constructed),
238     // then the next two methods can be used to solve the system M · X = B of linear equations in
239     // two steps: at first one has to solve the system L · Y = B, then one should solve the system
240     // U · X = Y.

241     // Using multi-threading, tries to solve systems of linear equations of type L · Y = B, where B can
242     // be either a column or rectangular matrix of multiple columns with the same row dimension as L.
243     // In case of success, the solution(s) will be stored in Y.
244     // If the input boolean parameter represent_solutions_as_columns is false, then the column dimension of
245     // the transpose of B should be equal to the row dimension of L.
246     // Note that, T can be either double or Cartesian3, or any other custom type which has similar overloaded
247     // mathematical and boolean operators.
248     template <typename T>
249     bool FactorizedUnpivotedLUDecomposition::solveLLinearSystem(
250             const Matrix<T> &B, Matrix<T> &Y, bool represent_solutions_as_columns)
251     {
252         if (!_decomposition_is_done)
253         {
254             return false;
255         }

256         int dimension = _L.columnCount();

257         RowMatrix<int> P(dimension);

258         for (int i = 0; i < dimension; i++)
259         {
```


© Ágoston Róth, 2018

```
260             P[i] = i;
261         }
262         if (represent_solutions_as_columns)
263         {
264             if (B.rowCount() != dimension)
265             {
266                 return false;
267             }
268             Y = B;
269             #pragma omp parallel for
270             for (int k = 0; k < B.columnCount(); k++)
271             {
272                 int ii = 0;
273                 for (int i = 0; i < dimension; i++)
274                 {
275                     int ip = P[i];
276                     T sum  = Y(ip, k);
277                     Y(ip, k) = Y(i, k);
278                     if (ii != 0)
279                     {
280                         for (int j = ii - 1; j < i; j++)
281                         {
282                             sum -= _L(i, j) * Y(j, k);
283                         }
284                     }
285                     else
286                     {
287                         if (sum != 0.0)
288                         {
289                             ii = i + 1;
290                         }
291                     }
292                     Y(i, k) = sum;
293                 }
294                 for (int i = dimension - 1; i >= 0; i--)
295                 {
296                     T sum = Y(i, k);
297                     for (int j = i + 1; j < dimension; j++)
298                     {
299                         sum -= _L(i, j) * Y(j, k);
300                     }
301                     Y(i, k) = sum /= _L(i, i);
302                 }
303             }
304         }
305         else
306         {
307             if (B.columnCount() != dimension)
308             {
309                 return false;
310             }
311             Y = B;
312             #pragma omp parallel for
313             for (int k = 0; k < B.rowCount(); k++)
314             {
315                 int ii = 0;
316                 for (int i = 0; i < dimension; i++)
317                 {
318                     int ip = P[i];
319                     T sum  = Y(k, ip);
320                     Y(k, ip) = Y(k, i);
```


© Ágoston Róth, 2018

```
321                     if (ii != 0)
322                     {
323                         for (int j = ii - 1; j < i; j++)
324                         {
325                             sum -= _L(i, j) * Y(k, j);
326                         }
327                     }
328                     else
329                     {
330                         if (sum != 0.0)
331                         {
332                             ii = i + 1;
333                         }
334                     }
335                     Y(k, i) = sum;
336                 }
337                 for (int i = dimension - 1; i >= 0; i--)
338                 {
339                     T sum = Y(k, i);
340                     for (int j = i + 1; j < dimension; j++)
341                     {
342                         sum -= _L(i, j) * Y(k, j);
343                     }
344                     Y(k, i) = sum /= _L(i, i);
345                 }
346             }
347         }
348         return true;
349     }
350     // Using multi-threading, tries to solve systems of linear equations of type U · X = Y, where Y can
351     // be either a column or rectangular matrix of multiple columns with the same row dimension as U.
352     // In case of success, the solution(s) will be stored in X.
353     // If the input boolean parameter represent_solutions_as_columns is false, then the column dimension of
354     // the transpose of Y should be equal to the row dimension of U.
355     // Note that, T can be either double or Cartesian3, or any other custom type which has similar overloaded
356     // mathematical and boolean operators.
357     template <typename T>
358     bool FactorizedUnpivotedLUDecomposition::solveULinearSystem(
359             const Matrix<T> &Y, Matrix<T> &X, bool represent_solutions_as_columns)
360     {
361         if (!_decomposition_is_done)
362         {
363             return false;
364         }
365         int dimension = _U.columnCount();
366         RowMatrix<int> P(dimension);
367         for (int i = 0; i < dimension; i++)
368         {
369             P[i] = i;
370         }
371         if (represent_solutions_as_columns)
372         {
373             if (Y.rowCount() != dimension)
374             {
375                 return false;
376             }
377             X = Y;
378             #pragma omp parallel for
379             for (int k = 0; k < Y.columnCount(); k++)
380             {
381                 int ii = 0;
```


© Ágoston Róth, 2018

```
382                 for (int i = 0; i < dimension; i++)
383                 {
384                     int ip = P[i];
385                     T sum    = X(ip, k);
386                     X(ip, k) = X(i, k);
387                     if (ii != 0)
388                     {
389                         for (int j = ii - 1; j < i; j++)
390                         {
391                             sum -= _U(i, j) * X(j, k);
392                         }
393                     }
394                     else
395                     {
396                         if (sum != 0.0)
397                         {
398                             ii = i + 1;
399                         }
400                     }
401                     X(i, k) = sum;
402                 }
403                 for (int i = dimension - 1; i >= 0; i--)
404                 {
405                     T sum = X(i, k);
406                     for (int j = i + 1; j < dimension; j++)
407                     {
408                         sum -= _U(i, j) * X(j, k);
409                     }
410                     X(i, k) = sum /= _U(i, i);
411                 }
412             }
413         }
414         else
415         {
416             if (Y.columnCount() != dimension)
417             {
418                 return false;
419             }
420             X = Y;
421             #pragma omp parallel for
422             for (int k = 0; k < Y.rowCount(); k++)
423             {
424                 int ii = 0;
425                 for (int i = 0; i < dimension; i++)
426                 {
427                     int ip = P[i];
428                     T sum    = X(k, ip);
429                     X(k, ip) = X(k, i);
430                     if (ii != 0)
431                     {
432                         for (int j = ii - 1; j < i; j++)
433                         {
434                             sum -= _U(i, j) * X(k, j);
435                         }
436                     }
437                     else
438                     {
439                         if (sum != 0.0)
440                         {
441                             ii = i + 1;
442                         }
443                     }
444                     X(k, i) = sum;
445                 }
```


© Ágoston Róth, 2018

```
446                 for (int i = dimension - 1; i >= 0; i--)
447                 {
448                     T sum = X(k, i);
449                     for (int j = i + 1; j < dimension; j++)
450                     {
451                         sum -= _U(i, j) * X(k, j);
452                     }
453                     X(k, i) = sum /= _U(i, i);
454                 }
455             }
456         }
457         return true;
458     }
459     // Using multi-threading, tries to solve systems of linear equations of type M · X = B, where M is not necessarily
460     // a real square matrix, while B can be either a column or rectangular matrix of multiple columns with the same
461     // row dimension as M. In case of success, the solution(s) will be stored in X.
462     // If the input boolean parameter represent_solutions_as_columns is false, then the column dimension of
463     // the transpose of B should be equal to the row dimension of M.
464     // Note that, T can be either double or Cartesian3, or any other custom type which has similar overloaded
465     // mathematical and boolean operators.
466     template <typename T>
467     bool SVDecomposition::solveLinearSystem(
468             const Matrix<T> &B, Matrix<T> &X, bool represent_solutions_as_columns)
469     {
470         if (!_decomposition_is_done)
471         {
472             X.resizeRows(0);
473             return false;
474         }
475         int m = _U.rowCount();
476         int n = _U.columnCount();
477         double epsilon = std::numeric_limits<double>::epsilon();
478         double threshold = 0.5 * sqrt(m + n + 1.0) * _S[0] * epsilon;
479         if (represent_solutions_as_columns)
480         {
481             if (B.rowCount() != m)
482             {
483                 throw Exception("SVDecomposition::solveLinearSystem : bad sizes!");
484             }
485             X.resizeRows(n);
486             X.resizeColumns(B.columnCount());
487             RowMatrix<double> tmp(n);
488             #pragma omp parallel for
489             for (int c = 0; c < B.columnCount(); c++)
490             {
491                 for (int j = 0; j < n; j++)
492                 {
493                     double s = 0.0;
494                     if (_S[j] > threshold)
495                     {
496                         for (int i = 0; i < m; i++)
497                         {
498                             s += _U(i, j) * B(i, c);
499                         }
500                         s /= _S[j];
501                     }
502                     tmp[j] = s;
503                 }
504                 for (int j = 0; j < n; j++)
```


© Ágoston Róth, 2018

```
505                 {
506                     double s = 0.0;
507                     for (int jj = 0; jj < n; jj++)
508                     {
509                         s += _V(j, jj) * tmp[jj];
510                     }
511                     X(j, c) = s;
512                 }
513             }
514         }
515         else
516         {
517             if (B.columnCount() != m)
518             {
519                 throw Exception("SVDecomposition::solveLinearSystem : bad sizes!");
520             }
521             X.resizeRows(B.rowCount());
522             X.resizeColumns(n);
523             RowMatrix<double> tmp(n);
524             #pragma omp parallel for
525             for (int r = 0; r < B.rowCount(); r++)
526             {
527                 for (int j = 0; j < n; j++)
528                 {
529                     double s = 0.0;
530                     if (_S[j] > threshold)
531                     {
532                         for (int i = 0; i < m; i++)
533                         {
534                             s += _U(i, j) * B(r, i);
535                         }
536                         s /= _S[j];
537                     }
538                     tmp[j] = s;
539                 }
540                 for (int j = 0; j < n; j++)
541                 {
542                     double s = 0.0;
543                     for (int jj = 0; jj < n; jj++)
544                     {
545                         s += _V(j, jj) * tmp[jj];
546                     }
547                     X(r, j) = s;
548                 }
549             }
550         }
551         return true;
552     }
553 }
554 #endif // REALMATRIXDECOMPOSITIONS_H
```

Listing 2.27. Real matrix decompositions (**Core/Math/RealMatrixDecompositions.cpp**)

```
1 #include "Constants.h"
2 #include "RealMatrixDecompositions.h"
3 #include "../Exceptions.h"
4 #include <algorithm>
```


© Ágoston Róth, 2018

```
5  #include <cmath>

6  using namespace std;

7  namespace cagd
8  {
9      // Tries to determine the pivoted Doolittle-type LU decomposition of the given real square matrix M.
10     // If M is singular then its LU factorization cannot be determined.
11     // If the decomposition is unsuccessful, the boolean variable _decomposition_is_done will be set to false, and if
12     // the input variable generate_exceptions is set to true, then the method will also generate an exception with a
13     // meaningful reason.
14     PLUDecomposition::PLUDecomposition(const RealMatrix &M,
15                                        bool generate_exceptions):
16         _decomposition_is_done(false),
17         _determinant(0.0),
18         _P(M.rowCount()),
19         _LU(M)
20     {
21         if (!_LU.isSquare() && generate_exceptions)
22         {
23             throw Exception("Only square real matrices can have PLU decompositions!");
24         }

25         int dimension = _LU.rowCount();

26         if (dimension <= 1 && generate_exceptions)
27         {
28             throw Exception("PLUDecomposition : the dimension of the given real "
29                             "square matrix should be at least 2 x 2!");

30             return;
31         }

32         for (int i = 0; i < dimension; i++)
33         {
34             _P[i] = i;
35         }

36         vector<double> implicit_scaling_of_each_row(dimension);

37         double row_interchanges = 1.0;

38         // loop over the rows to get the implicit scaling information
39         for (int r = 0; r < dimension; r++)
40         {
41             double big = 0.0;
42             for (int c = 0; c < dimension; c++)
43             {
44                 double temp = abs(_LU(r, c));
45                 if (temp > big)
46                 {
47                     big = temp;
48                 }
49             }

50             if (big == 0.0)
51             {
52                 _decomposition_is_done = false;
53                 _determinant           = 0.0;

54                 if (generate_exceptions)
55                 {
56                     throw Exception("PLUDecomposition : the given square matrix is "
57                                     "singular!");
58                 }

59                 return;
60             }

61             implicit_scaling_of_each_row[r] = 1.0 / big;
62         }

63         // search for the largest pivot element
64         for (int k = 0; k < dimension; k++)
```


© Ágoston Róth, 2018

```
65          {
66              int i_max = k;
67              double big = 0.0;
68              for (int i = k; i < dimension; i++)
69              {
70                  double temp = implicit_scaling_of_each_row[i] * abs(_LU(i, k));
71                  if (temp > big)
72                  {
73                      big   = temp;
74                      i_max = i;
75                  }
76              }
77              // do we need to interchange rows?
78              if (k != i_max)
79              {
80                  for (int j = 0; j < dimension; j++)
81                  {
82                      double temp   = _LU(i_max, j);
83                      _LU(i_max, j) = _LU(k, j);
84                      _LU(k, j)     = temp;
85                  }
86                  // change the parity of row_interchanges
87                  row_interchanges = -row_interchanges;
88                  // also interchange the scale factor
89                  implicit_scaling_of_each_row[i_max] = implicit_scaling_of_each_row[k];
90              }
91              _P[k] = i_max;
92              if (_LU(k, k) == 0.0)
93              {
94                  _LU(k, k) = TINY;
95              }
96              for (int i = k + 1; i < dimension; i++)
97              {
98                  // divide by pivot element
99                  double temp = _LU(i, k) /= _LU(k, k);
100                 // reduce remaining submatrix
101                 for (int j = k + 1; j < dimension; j++)
102                 {
103                     _LU(i, j) -= temp * _LU(k, j);
104                 }
105             }
106         }
107         _determinant = row_interchanges;
108         for (int i = 0; i < dimension; i++)
109         {
110             _determinant *= _LU(i, i);
111         }
112         _decomposition_is_done = true;
113     }
114     // Returns either the successful or the unsuccessful state of the decomposition.
115     bool PLUDecomposition::isCorrect() const
116     {
117         return _decomposition_is_done;
118     }
119     // If the decomposition was successful the method will return the product of the diagonal entries of
120     // the upper triangular matrix U (i.e., the determinant of M), otherwise it will return zero.
121     double PLUDecomposition::determinant() const
122     {
123         if (_decomposition_is_done)
124         {
125             return _determinant;
126         }
```


© Ágoston Róth, 2018

```
127         return 0.0;
128     }
129     // clone function required by smart pointers based on the deep copy ownership policy
130     PLUDecomposition* PLUDecomposition::clone() const
131     {
132         return new (nothrow) PLUDecomposition(*this);
133     }
134     // The next constructor tries to determine the unpivoted Doolittle-type LU decomposition of the given real
135     // square matrix M. If M is singular then its LU factorization cannot be determined.
136     // If the decomposition is unsuccessful, the boolean variable _decomposition_is_done will be set to false,
137     // and if the input variable generate_exceptions is set to true, then the method will
138     // also generate an exception with a meaningful reason.
139     FactorizedUnpivotedLUDecomposition::FactorizedUnpivotedLUDecomposition(
140             const RealMatrix &M, bool generate_exceptions):
141         _decomposition_is_done(false),
142         _determinant(0.0),
143         _L(M.rowCount(), M.columnCount()), _U(M.rowCount(), M.columnCount())
144     {
145         if (!M.isSquare())
146         {
147             throw Exception("Only square real matrices can have unpivoted "
148                             "LU decompositions!");
149         }
150         int dimension = M.rowCount();
151         _L.loadIdentityMatrix();
152         _U.loadNullMatrix();
153         for (int j = 0; j < dimension; j++)
154         {
155             _U(0, j) = M(0, j);
156         }
157         for (int i = 1; i < dimension; i++)
158         {
159             for (int j = 0; j < dimension; j++)
160             {
161                 for (int k = 0; k <= i - 1; k++)
162                 {
163                     double s1 = 0.0;
164                     if (k == 0)
165                     {
166                         s1 = 0.0;
167                     }
168                     else
169                     {
170                         for (int p = 0; p <= k - 1; p++)
171                         {
172                             s1 += _L(i, p) * _U(p, k);
173                         }
174                     }
175                     if (!_U(k, k))
176                     {
177                         _L.resizeRows(0);
178                         _U.resizeRows(0);
179                         _decomposition_is_done = false;
180                         _determinant = 0.0;
181                         if (generate_exceptions)
182                         {
183                             throw Exception("FactorizedUnpivotedLUDecomposition : the "
184                                             "given square matrix is singular!");
185                         }
186                         return;
187                     }
188                     _L(i, k) = (M(i, k) - s1) / _U(k, k);
```


© Ágoston Róth, 2018

```
189                 }
190                 for (int k=i; k < dimension; k++)
191                 {
192                     double s2 = 0.0;
193                     for (int p = 0; p <= i - 1; p++)
194                     {
195                         s2 += _L(i, p) * _U(p, k);
196                     }
197                     _U(i, k) = M(i, k) - s2;
198                 }
199             }
200         }
201         _determinant = 1.0;
202         for (int i = 0; i < dimension; i++)
203         {
204             _determinant *= _U(i, i);
205         }
206         _decomposition_is_done = true;
207     }
208     // Returns either the successful or the unsuccessful state of the decomposition.
209     bool FactorizedUnpivotedLUDecomposition::isCorrect() const
210     {
211         return _decomposition_is_done;
212     }
213     // If the decomposition was successful the method will return the product of the diagonal entries of
214     // the upper triangular matrix U (i.e., the determinant of M), otherwise it will return zero.
215     double FactorizedUnpivotedLUDecomposition::determinant() const
216     {
217         if (_decomposition_is_done)
218         {
219             return _determinant;
220         }
221         return 0.0;
222     }
223     // Returns a constant reference to the lower triangular matrix L.
224     const RealMatrix& FactorizedUnpivotedLUDecomposition::L() const
225     {
226         return _L;
227     }
228     // Returns a constant reference to the upper triangular matrix U.
229     const RealMatrix& FactorizedUnpivotedLUDecomposition::U() const
230     {
231         return _U;
232     }
233     // clone function required by smart pointers based on the deep copy ownership policy
234     FactorizedUnpivotedLUDecomposition* FactorizedUnpivotedLUDecomposition::clone() const
235     {
236         return new (nothrow) FactorizedUnpivotedLUDecomposition(*this);
237     }
238     // Some auxiliar functions for singular value decomposition.
239     template <typename T>
240     inline T sign(const T &a, const T &b)
241     {
242         return b >= 0 ? (a >= 0 ? a : -a) : (a >= 0 ? -a : a);
243     }
244     double EuclideanNorm(const double &a, const double &b)
245     {
246         double abs_a  = abs(a), abs_b = abs(b),
247                factor = abs_b / abs_a, rfactor = abs_a / abs_b;
248         return (abs_a > abs_b ? abs_a * sqrt(1.0 + factor * factor) :
```


© Ágoston Róth, 2018

```
249                                 (abs_b == 0.0 ?
250                                  0.0 : abs_b * sqrt(1.0 + rfactor * rfactor)));
251     }
252     // Tries to determine the singular value decomposition of the given real matrix M.
253     // If the decomposition is unsuccessful, the boolean variable _decomposition_is_done will be set to false, and if
254     // the input variable generate_exceptions is set to true,
255     // then the method will also generate an exception with a meaningful reason.
256     SVDecomposition::SVDecomposition(const RealMatrix &M, bool generate_exceptions):
257         _decomposition_is_done(false),
258         _product_of_singular_values(0.0),
259         _U(M),
260         _S(M.columnCount()),
261         _V(M.columnCount(), M.columnCount())
262     {
263         int m = _U.rowCount();
264         int n = _U.columnCount();
265         // decompose
266         {
267             bool                flag;
268             int                 i, iterations, j, jj, k, l, nm;
269             double              M_norm = 0.0, c, f, g = 0.0, h, s, scale = 0.0, x, y, z;
270             RowMatrix<double> rv1(n);
271             for (i = 0; i < n; i++)
272             {
273                 l = i + 2;
274                 rv1[i] = scale * g;
275                 g = s = scale = 0.0;
276                 if (i < m)
277                 {
278                     for (k = i; k < m; k++)
279                     {
280                         scale += abs(_U(k, i));
281                     }
282                     if (scale != 0.0)
283                     {
284                         for (k = i; k < m; k++)
285                         {
286                             _U(k, i) /= scale;
287                             s += _U(k, i) * _U(k, i);
288                         }
289                         f = _U(i, i);
290                         g = -sign(sqrt(s), f);
291                         h = f * g - s;
292                         _U(i, i) = f - g;
293                         for (j = l - 1; j < n; j++)
294                         {
295                             s = 0.0;
296                             for (k = i; k < m; k++)
297                             {
298                                 s += _U(k, i) *_U(k, j);
299                             }
300                             f = s / h;
301                             for (k = i; k < m; k++)
302                             {
303                                 _U(k, j) += f * _U(k, i);
304                             }
305                         }
306                         for (k = i; k < m; k++)
307                         {
308                             _U(k, i) *= scale;
309                         }
310                     }
311                 }
```


© Ágoston Róth, 2018

```
312                 _S[i] = scale * g;
313                 g = s = scale = 0.0;
314                 if (i+1 <= m && i+1 != n)
315                 {
316                     for (k = l - 1; k < n; k++)
317                     {
318                         scale += abs(_U(i, k));
319                     }
320                     if (scale != 0.0)
321                     {
322                         for (k = l - 1; k < n; k++)
323                         {
324                             _U(i, k) /= scale;
325                             s += _U(i, k) * _U(i, k);
326                         }
327                         f = _U(i, l - 1);
328                         g = -sign(sqrt(s),f);
329                         h = f * g - s;
330                         _U(i, l - 1) = f - g;
331                         for (k = l - 1; k < n; k++)
332                         {
333                             rv1[k] = _U(i, k) / h;
334                         }
335                         for (j = l - 1; j < m; j++)
336                         {
337                             s = 0.0;
338                             for (k = l - 1; k < n; k++)
339                             {
340                                 s += _U(j, k) * _U(i, k);
341                             }
342                             for (k = l - 1; k < n; k++)
343                             {
344                                 _U(j, k) += s * rv1[k];
345                             }
346                         }
347                         for (k = l - 1; k < n; k++)
348                         {
349                             _U(i, k) *= scale;
350                         }
351                     }
352                 }
353                 M_norm = max(M_norm, (abs(_S[i]) + abs(rv1[i])));
354             }
355             for (i = n - 1; i >= 0; i--)
356             {
357                 if (i < n - 1)
358                 {
359                     if (g != 0.0)
360                     {
361                         for (j = l; j < n; j++)
362                         {
363                             _V(j, i) = (_U(i, j) / _U(i, l)) / g;
364                         }
365                         for (j = l; j < n; j++)
366                         {
367                             s = 0.0;
368                             for (k = l; k < n; k++)
369                             {
370                                 s += _U(i, k) * _V(k, j);
371                             }
372                             for (k = l; k < n; k++)
373                             {
```


© Ágoston Róth, 2018

```
374                                 _V(k, j) += s * _V(k, i);
375                             }
376                         }
377                     }

378                     for (j = l; j < n; j++)
379                     {
380                         _V(i, j) = _V(j, i) = 0.0;
381                     }
382                 }

383                 _V(i, i) = 1.0;
384                 g = rv1[i];
385                 l = i;
386             }

387             for (i = min(m, n) - 1; i >= 0; i--)
388             {
389                 l = i+1;
390                 g = _S[i];
391                 for (j = l; j < n; j++)
392                 {
393                     _U(i, j) = 0.0;
394                 }

395                 if (g != 0.0)
396                 {
397                     g = 1.0 / g;
398                     for (j = l; j < n; j++)
399                     {
400                         s = 0.0;
401                         for (k = l; k < m; k++)
402                         {
403                             s += _U(k, i) * _U(k, j);
404                         }

405                         f = (s / _U(i, i)) * g;
406                         for (k = i; k < m; k++)
407                         {
408                             _U(k, j) += f * _U(k, i);
409                         }
410                     }

411                     for (j = i; j < m; j++)
412                     {
413                         _U(j, i) *= g;
414                     }
415                 }
416                 else
417                 {
418                     for (j = i; j < m; j++)
419                     {
420                         _U(j, i) = 0.0;
421                     }
422                 }

423                 ++_U(i, i);
424             }

425             for (k = n - 1; k >= 0; k--)
426             {
427                 for (iterations = 0; iterations < 30; iterations++)
428                 {
429                     flag = true;
430                     for (l = k; l >= 0; l--)
431                     {
432                         nm = l - 1;

433                         if (l == 0 || abs(rv1[l]) <= MACHINE_EPS * M_norm)
434                         {
435                             flag = false;
436                             break;
437                         }
```


© Ágoston Róth, 2018

```
438                         if (abs(_S[nm]) <= MACHINE_EPS * M_norm)
439                         {
440                             break;
441                         }
442                     }

443                     if (flag)
444                     {
445                         c = 0.0;
446                         s = 1.0;
447                         for (i = l; i < k + 1; i++)
448                         {
449                             f = s * rv1[i];
450                             rv1[i] = c * rv1[i];

451                             if (abs(f) <= MACHINE_EPS * M_norm)
452                             {
453                                 break;
454                             }

455                             g     = _S[i];
456                             h     = EuclideanNorm(f, g);
457                             _S[i] = h;
458                             h     = 1.0 / h;
459                             c     = g * h;
460                             s     = -f * h;

461                             for (j = 0; j < m; j++)
462                             {
463                                 y          = _U(j, nm);
464                                 z          = _U(j, i);
465                                 _U(j, nm) = y * c + z * s;
466                                 _U(j, i)  = z * c - y * s;
467                             }
468                         }
469                     }

470                     z = _S[k];

471                     if (l == k)
472                     {
473                         if (z < 0.0)
474                         {
475                             _S[k] = -z;
476                             for (j = 0; j < n; j++)
477                             {
478                                 _V(j, k) = -_V(j, k);
479                             }
480                         }
481                         break;
482                     }

483                     if (iterations == 29)
484                     {
485                         _decomposition_is_done = false;

486                         if (generate_exceptions)
487                         {
488                             throw Exception("No convergence in 30 singular value "
489                                             "decomposition iterations!");
490                         }

491                         return;
492                     }

493                     x  = _S[l];
494                     nm = k - 1;
495                     y  = _S[nm];
496                     g  = rv1[nm];
497                     h  = rv1[k];
498                     f  = ((y - z) * (y + z) + (g - h) * (g + h)) / (2.0 * h * y);
499                     g  = EuclideanNorm(f, 1.0);
500                     f  = ((x - z) * (x + z) + h * ((y / (f + sign(g, f))) - h)) / x;
501                     c  = s = 1.0;
```


© Ágoston Róth, 2018

```
502                     for (j = l; j <= nm; j++)
503                     {
504                         i       =  j + 1;
505                         g       =  rv1[i];
506                         y       =  _S[i];
507                         h       =  s * g;
508                         g       =  c * g;
509                         z       =  EuclideanNorm(f, h);
510                         rv1[j] =  z;
511                         c       =  f / z;
512                         s       =  h / z;
513                         f       =  x * c + g * s;
514                         g       =  g * c - x * s;
515                         h       =  y * s;
516                         y      *= c;
517                         for (jj = 0; jj < n; jj++)
518                         {
519                             x         = _V(jj, j);
520                             z         = _V(jj, i);
521                             _V(jj, j) = x * c + z * s;
522                             _V(jj, i) = z * c - x * s;
523                         }
524                         z      = EuclideanNorm(f, h);
525                         _S[j] = z;
526                         if (z)
527                         {
528                             z = 1.0 / z;
529                             c = f * z;
530                             s = h * z;
531                         }
532                         f = c * g + s * y;
533                         x = c * y - s * g;
534                         for (jj = 0; jj < m; jj++)
535                         {
536                             y          = _U(jj, j);
537                             z          = _U(jj, i);
538                             _U(jj, j) = y * c + z * s;
539                             _U(jj, i) = z * c - y * s;
540                         }
541                     }
542                     rv1[l] = 0.0;
543                     rv1[k] = f;
544                     _S[k]  = x;
545                 }
546             }
547         }
548         // reorder
549         {
550             int                 i, j, k, s, inc = 1;
551             double              sw;
552             RowMatrix<double> su(m), sv(n);
553             do
554             {
555                 inc *= 3;
556                 inc++;
557             }
558             while (inc <= n);
559             do
560             {
561                 inc /= 3;
562                 for (i = inc; i < n; i++)
563                 {
564                     sw = _S[i];
565                     for (k = 0; k < m; k++)
```


© Ágoston Róth, 2018

```
566                     {
567                         su[k] = _U(k, i);
568                     }
569                     for (k = 0; k < n; k++)
570                     {
571                         sv[k] = _V(k, i);
572                     }
573                     j = i;
574                     while (_S[j - inc] < sw)
575                     {
576                         _S[j] = _S[j - inc];
577                         for (k=0;k<m;k++)
578                         {
579                             _U(k, j) = _U(k, j - inc);
580                         }
581                         for (k = 0; k < n; k++)
582                         {
583                             _V(k, j) = _V(k, j - inc);
584                         }
585                         j -= inc;
586                         if (j < inc)
587                         {
588                             break;
589                         }
590                     }
591                     _S[j] = sw;
592                     for (k = 0; k < m; k++)
593                     {
594                         _U(k, j) = su[k];
595                     }
596                     for (k = 0; k < n; k++)
597                     {
598                         _V(k, j) = sv[k];
599                     }
600                 }
601             }
602             while (inc > 1);
603             for (k = 0; k < n; k++)
604             {
605                 s=0;
606                 for (i = 0; i < m; i++)
607                 {
608                     if (_U(i, k) < 0.0)
609                     {
610                         s++;
611                     }
612                 }
613                 for (j = 0; j < n; j++)
614                 {
615                     if (_V(j, k) < 0.0)
616                     {
617                         s++;
618                     }
619                 }
620                 if (s > (m + n) / 2)
621                 {
622                     for (i = 0; i < m; i++)
623                     {
624                         _U(i, k) = -_U(i, k);
625                     }
```


© Ágoston Róth, 2018

```
626                     for (j = 0; j < n; j++)
627                     {
628                         _V(j, k) = -_V(j, k);
629                     }
630                 }
631             }
632         }
633         // product of singular values (i.e., the determinant of M provided that it is a square matrix)
634         _product_of_singular_values = 1.0;
635         for (int i = 0; i < _S.columnCount(); i++)
636         {
637             _product_of_singular_values *= _S[i];
638         }
639         _decomposition_is_done = true;
640     }
641     // Returns either the successful or the unsuccessful state of the decomposition.
642     bool SVDecomposition::isCorrect() const
643     {
644         return _decomposition_is_done;
645     }
646     // Returns the ratio of the largest and smalles singular values.
647     double SVDecomposition::conditionNumber() const
648     {
649         int last = _S.columnCount() - 1;
650         return (_S[0] <= 0.0 || _S[last] <= 0.0) ? 0.0 : _S[0] / _S[last];
651     }
652     // Returns the ratio of the smallest and largest singular values.
653     double SVDecomposition::reciprocalConditionNumber() const
654     {
655         int last = _S.columnCount() - 1;
656         return (_S[0] <= 0.0 || _S[last] <= 0.0) ? 0.0 : _S[last] / _S[0];
657     }
658     // Returns the product of the obtained singular values. If M is a regular real square matrix,
659     // then this product coincides with the determinant of M.
660     double SVDecomposition::productOfSingularValues() const
661     {
662         if (_decomposition_is_done)
663         {
664             return _product_of_singular_values;
665         }
666         return 0.0;
667     }
668     // clone function required by smart pointers based on the deep copy ownership policy
669     SVDecomposition* SVDecomposition::clone() const
670     {
671         return new (nothrow) SVDecomposition(*this);
672     }
673 }
```

2.9.3 Generic and special OpenGL transformation matrices

Since the classic OpenGL transformation commands (like glMultMatrix{f|d}, glRotate{f|d}, glTranslate{f|d}, glScale{f|d}, glOrtho{f|d}, gluPerspective, gluLookAt{f|d}) and the traditional matrix stacks that are associated with the projection and model-view matrix modes are considered deprecated, we also provide classes for generic and special OpenGL transformation matrices by means of which one can define general, rotation, translation, scaling, orthogonal or perspective projection and world coordinate transformation matrices. By design, each special OpenGL transformation is derived from the generic one GLTransformation that stores a 4×4 column-major ordered matrix in a memory-continuous float array whose constant address can be sent either to our or to user-defined custom shader program objects in order to initialize/communicate with matrices

© Ágoston Róth, 2018

defined as uniform variables of type mat4. The class GLTransformation also defines several multi-threaded mathematical operators and auxiliary methods by means of which one can either compose more complex transformations or to transform Homogeneous3 and Cartesian3 coordinates as well. (Since matrix classes Matrix<T>, RealMatrix: public Matrix<double> and RealSquareMatrix: public RealMatrix store their data in row-major order and OpenGL expects the transformation matrices by default in column-major order, the class GLTransformation is not inherited from the matrix classes presented so far.)

The structure of the base class GLTransformation is illustrated in Fig. 2.17/109, while its definition and implementation can be found in Listings 2.28/109 and 2.29/111, respectively.

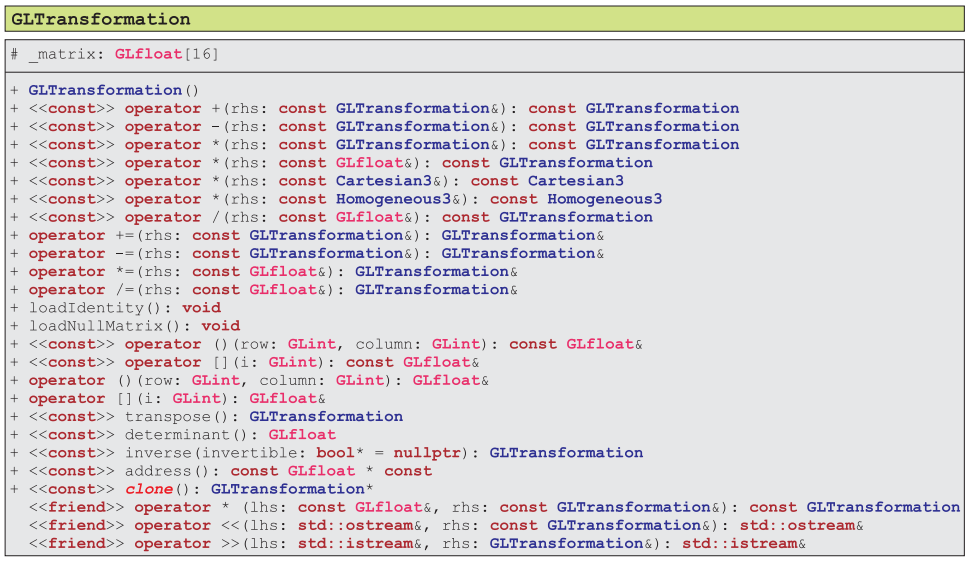


Fig. 2.17: Diagram of generic OpenGL transformations

Listing 2.28. Generic OpenGL transformations (**Core/Math/GenericGLTransformations.h**)

```
1  #ifndef GENERICGLTRANSFORMATIONS_H
2  #define GENERICGLTRANSFORMATIONS_H

3  #include "../Geometry/Coordinates/Cartesians3.h"
4  #include "../Geometry/Coordinates/Homogeneous3.h"

5  #include <GL/glew.h>
6  #include <cmath>
7  #include <iostream>
8  #include <new>

9  namespace cagd
10 {
11     class GLTransformation
12     {
13         // multiplicate the transformation matrix rhs by a constant from left
14         friend const GLTransformation operator *(GLfloat lhs,
15                                                  const GLTransformation &rhs);

16         // output to stream
17         friend std::ostream& operator <<(std::ostream &lhs, const GLTransformation &rhs);

18         // input from stream
19         friend std::istream& operator >>(std::istream &lhs, GLTransformation &rhs);
```


© Ágoston Róth, 2018

```
20      protected:
21          // stores a 4 × 4 column-major ordered transformation matrix, i.e.,
22          //
23          // T = (_matrix[0] _matrix[4] _matrix[ 8] _matrix[12]
24          //      _matrix[1] _matrix[5] _matrix[ 9] _matrix[13]
25          //      _matrix[2] _matrix[6] _matrix[10] _matrix[14]
26          //      _matrix[3] _matrix[7] _matrix[11] _matrix[15])
27          GLfloat _matrix[16];
28      public:
29          // default constructor, loads the identity matrix
30          GLTransformation();
31          // matrix addition
32          const GLTransformation operator +(const GLTransformation &rhs) const;
33          // matrix subtraction
34          const GLTransformation operator -(const GLTransformation &rhs) const;
35          // matrix multiplication
36          const GLTransformation operator *(const GLTransformation &rhs) const;
37          // multiplication by scalar from right
38          const GLTransformation operator *(const GLfloat &rhs) const;
39          // using homogeneous coordinates, transforms/maps a Cartesian coordinate into another one
40          const Cartesian3 operator *(const Cartesian3 &rhs) const;
41          // transforms the given homogeneous coordinate to another one
42          const Homogeneous3 operator *(const Homogeneous3 &rhs) const;
43          // division by scalar from right
44          const GLTransformation operator /(const GLfloat &rhs) const;
45          // add to *this
46          GLTransformation& operator +=(const GLTransformation &rhs);
47          // subtract from *this
48          GLTransformation& operator -=(const GLTransformation &rhs);
49          // multiplicate *this by a constant
50          GLTransformation& operator *=(const GLfloat &rhs);
51          // divide *this by a constant
52          GLTransformation& operator /=(const GLfloat &rhs);
53          // loads the 4 × 4 identity matrix
54          GLvoid loadIdentity();
55          // loads the 4 × 4 null matrix
56          GLvoid loadNullMatrix();
57          // get element by constant reference
58          const GLfloat& operator [](GLint i) const
59          {
60              assert("The given index is out of bounds!" && (i >= 0 && i < 16));
61              return _matrix[i];
62          }
63          const GLfloat& operator ()(GLint row, GLint column) const
64          {
65              assert("The given row index is out of bounds!" && (row >= 0 && row < 4));
66              assert("The given column index is out of bounds!" &&
67                     (column >= 0 && column < 4));
68              return _matrix[column * 4 + row];
69          }
70          // get element by non-constant reference
71          GLfloat& operator [](GLint i)
72          {
73              assert("The given index is out of bounds!" && (i >= 0 && i < 16));
74              return _matrix[i];
75          }
```


© Ágoston Róth, 2018

```
76          GLfloat& operator ()(GLint row, GLint column)
77          {
78              assert("The given row index is out of bounds!" && (row >= 0 && row < 4));
79              assert("The given column index is out of bounds!" &&
80                     (column >= 0 && column < 4));
81              return _matrix[column * 4 + row];
82          }
83          // returns the transpose of the stored matrix
84          GLTransformation transpose() const;
85          // calculates the determinant of the stored matrix
86          GLfloat determinant() const;
87          // calculates the inverse of the stored matrix
88          // if the stored matrix is singular, a 4 x 4 identity matrix will be returned
89          GLTransformation inverse(bool *invertible = nullptr) const;
90          // returns the constant memory address of the stored matrix
91          const GLfloat* address() const;
92          // clone function required by smart pointers based on the deep copy ownership policy
93          virtual GLTransformation* clone() const;
94          // virtual default destructor
95          virtual ~GLTransformation();
96      };
97  }
98  #endif // GENERICGLTRANSFORMATIONS_H
```

Listing 2.29. Generic OpenGL transformations (**Core/Math/GenericGLTransformations.cpp**)

```
1   #include "GenericGLTransformations.h"
2   #include <new>
3   using namespace std;
4   namespace cagd
5   {
6       // default constructor, loads the identity matrix
7       GLTransformation::GLTransformation()
8       {
9           #pragma omp parallel for
10          for (GLint i = 0; i < 16; i++)
11          {
12              _matrix[i] = ((i % 5 == 0) ? 1.0f : 0.0f);
13          }
14      }
15      // matrix addition
16      const GLTransformation GLTransformation::operator +(const GLTransformation &rhs) const
17      {
18          GLTransformation result(*this);
19          #pragma omp parallel for
20          for (GLint i = 0; i < 16; i++)
21          {
22              result._matrix[i] += rhs._matrix[i];
23          }
24          return result;
25      }
26      // matrix subtraction
27      const GLTransformation GLTransformation::operator -(const GLTransformation &rhs) const
28      {
29          GLTransformation result(*this);
30          #pragma omp parallel for
31          for (GLint i = 0; i < 16; i++)
```


© Ágoston Róth, 2018

```
32          {
33              result._matrix[i] -= rhs._matrix[i];
34          }
35          return result;
36      }
37      // matrix multiplication
38      const GLTransformation GLTransformation::operator *(const GLTransformation &rhs) const
39      {
40          GLTransformation result;
41          result.loadNullMatrix();
42          #pragma omp parallel for
43          for (GLint i_j = 0; i_j < 16; i_j++)
44          {
45              GLint i     = i_j % 4;
46              GLint j     = i_j / 4;
47              GLint index = 4 * j + i;
48              for (GLint k = 0; k < 4; k++)
49              {
50                  result._matrix[index] += _matrix[4 * k + i] * rhs._matrix[4 * j + k];
51              }
52          }
53          return result;
54      }
55      // multiplication by scalar from right
56      const GLTransformation GLTransformation::operator *(const GLfloat &rhs) const
57      {
58          GLTransformation result(*this);
59          #pragma omp parallel for
60          for (GLint i = 0; i < 16; i++)
61          {
62              result._matrix[i] *= rhs;
63          }
64          return result;
65      }
66      // using homogeneous coordinates, transforms/maps a Cartesian coordinate into another one
67      const Cartesian3 GLTransformation::operator *(const Cartesian3 &rhs) const
68      {
69          GLfloat hIn[] = {(GLfloat)rhs[0], (GLfloat)rhs[1], (GLfloat)rhs[2], 1.0f};
70          GLfloat hOut[] = {0.0f, 0.0f, 0.0f, 0.0f};
71          #pragma omp parallel for
72          for (GLint j_i = 0; j_i < 16; j_i++)
73          {
74              GLint j = j_i / 4;
75              GLint i = j_i % 4;
76              GLfloat member = _matrix[4 * i + j] * hIn[i];
77              #pragma omp atomic
78              hOut[j] += member;
79          }
80          return Cartesian3(hOut[0] / hOut[3], hOut[1] / hOut[3], hOut[2] / hOut[3]);
81      }
82      // transforms the given homogeneous coordinate to another one
83      const Homogeneous3 GLTransformation::operator *(const Homogeneous3 &rhs) const
84      {
85          Homogeneous3 result(0.0f, 0.0f, 0.0f, 0.0f);
86          #pragma omp parallel for
87          for (GLint j_i = 0; j_i < 16; j_i++)
88          {
89              GLint j = j_i / 4;
90              GLint i = j_i % 4;
91              GLfloat member = _matrix[4 * i + j] * rhs[i];
92              #pragma omp atomic
```


© Ágoston Róth, 2018

```
93              result[j] += member;
94          }
95          return result;
96      }
97      // division by scalar from right
98      const GLTransformation GLTransformation::operator /(const GLfloat &rhs) const
99      {
100         assert("GLTransformation::operator /(const GLfloat &rhs) : division by zero!" &&
101                 rhs != 0.0f);
102         GLTransformation result(*this);
103         #pragma omp parallel for
104         for (GLint i = 0; i < 16; i++)
105         {
106             result._matrix[i] /= rhs;
107         }
108         return result;
109     }
110     // add to *this
111     GLTransformation& GLTransformation::operator +=(const GLTransformation &rhs)
112     {
113         #pragma omp parallel for
114         for (GLint i = 0; i < 16; i++)
115         {
116             _matrix[i] += rhs._matrix[i];
117         }
118         return *this;
119     }
120     // subtract from *this
121     GLTransformation& GLTransformation::operator -= (const GLTransformation &rhs)
122     {
123         #pragma omp parallel for
124         for (GLint i = 0; i < 16; i++)
125         {
126             _matrix[i] -= rhs._matrix[i];
127         }
128         return *this;
129     }
130     // multiplicate *this by a constant
131     GLTransformation& GLTransformation::operator *= (const GLfloat &rhs)
132     {
133         #pragma omp parallel for
134         for (GLint i = 0; i < 16; i++)
135         {
136             _matrix[i] *= rhs;
137         }
138         return *this;
139     }
140     // divide *this by a constant
141     GLTransformation& GLTransformation::operator /= (const GLfloat &rhs)
142     {
143         assert("GLTransformation::operator /=(const GLfloat &constant) : division "
144                 "by zero!" && rhs != 0.0f);
145         #pragma omp parallel for
146         for (GLint i = 0; i < 16; i++)
147         {
148             _matrix[i] /= rhs;
149         }
150         return *this;
151     }
```


© Ágoston Róth, 2018

```
152     // loads the 4 × 4 identity matrix
153     GLvoid GLTransformation::loadIdentity()
154     {
155         #pragma omp parallel for
156         for (GLint i = 0; i < 16; i++)
157         {
158             _matrix[i] = (i % 5) ? 0.0f : 1.0f;
159         }
160     }

161     // loads the 4 × 4 null matrix
162     GLvoid GLTransformation::loadNullMatrix()
163     {
164         #pragma omp parallel for
165         for (GLint i = 0; i < 16; i++)
166         {
167             _matrix[i] = 0.0f;
168         }
169     }

170     // returns the transpose of the stored matrix
171     GLTransformation GLTransformation::transpose() const
172     {
173         GLTransformation result;

174         #pragma omp parallel for
175         for (GLint r_c = 0; r_c < 16; r_c++)
176         {
177             GLint r = r_c / 4;
178             GLint c = r_c % 4;

179             result._matrix[c * 4 + r] = _matrix[r * 4 + c];
180         }

181         return result;
182     }

183     // auxiliar function that evaluates the determinant of a 3 × 3 matrix
184     GLfloat determinat3x3(
185             GLfloat a00, GLfloat a01, GLfloat a02,
186             GLfloat a10, GLfloat a11, GLfloat a12,
187             GLfloat a20, GLfloat a21, GLfloat a22)
188     {
189         return (a00 * a11 * a22 + a10 * a21 * a02 + a20 * a01 * a12) -
190                (a02 * a11 * a20 + a12 * a21 * a00 + a22 * a01 * a10);
191     }

192     // calculates the determinant of the stored matrix
193     GLfloat GLTransformation::determinant() const
194     {
195         GLfloat
196         result = + _matrix[0] * determinat3x3(
197                                 _matrix[5], _matrix[ 9], _matrix[13],
198                                 _matrix[6], _matrix[10], _matrix[14],
199                                 _matrix[7], _matrix[11], _matrix[15])

200                  - _matrix[4] * determinat3x3(
201                                 _matrix[1], _matrix[ 9], _matrix[13],
202                                 _matrix[2], _matrix[10], _matrix[14],
203                                 _matrix[3], _matrix[11], _matrix[15])

204                  + _matrix[8] * determinat3x3(
205                                 _matrix[1], _matrix[5], _matrix[13],
206                                 _matrix[2], _matrix[6], _matrix[14],
207                                 _matrix[3], _matrix[7], _matrix[15])

208                  - _matrix[12] * determinat3x3(
209                                 _matrix[1], _matrix[5], _matrix[ 9],
210                                 _matrix[2], _matrix[6], _matrix[10],
211                                 _matrix[3], _matrix[7], _matrix[11]);

212         return result;
213     }
```


© Ágoston Róth, 2018

```
214     // calculates the inverse of the stored matrix
215     // if the stored matrix is singular, a 4 x 4 identity matrix will be returned
216     GLTransformation GLTransformation::inverse(bool *invertible) const
217     {
218         #define SWAP_ROWS(a, b) { GLfloat *_tmp = a; (a)=(b); (b)=_tmp; }
219         #define MATRIX(m,r,c) (m)[(c)*4+(r)]
220 
221             GLTransformation result;
222 
223             GLfloat wtmp[4][8];
224             GLfloat m0, m1, m2, m3, s;
225             GLfloat *r0, *r1, *r2, *r3;
226 
227             r0 = wtmp[0], r1 = wtmp[1], r2 = wtmp[2], r3 = wtmp[3];
228 
229             r0[0] = MATRIX(_matrix, 0, 0);
230             r0[1] = MATRIX(_matrix, 0, 1);
231             r0[2] = MATRIX(_matrix, 0, 2);
232             r0[3] = MATRIX(_matrix, 0, 3);
233             r0[4] = 1.0, r0[5] = r0[6] = r0[7] = 0.0f;
234 
235             r1[0] = MATRIX(_matrix, 1, 0);
236             r1[1] = MATRIX(_matrix, 1, 1);
237             r1[2] = MATRIX(_matrix, 1, 2);
238             r1[3] = MATRIX(_matrix, 1, 3);
239             r1[5] = 1.0, r1[4] = r1[6] = r1[7] = 0.0f;
240 
241             r2[0] = MATRIX(_matrix, 2, 0);
242             r2[1] = MATRIX(_matrix, 2, 1);
243             r2[2] = MATRIX(_matrix, 2, 2);
244             r2[3] = MATRIX(_matrix, 2, 3);
245             r2[6] = 1.0, r2[4] = r2[5] = r2[7] = 0.0f;
246 
247             r3[0] = MATRIX(_matrix, 3, 0);
248             r3[1] = MATRIX(_matrix, 3, 1);
249             r3[2] = MATRIX(_matrix, 3, 2);
250             r3[3] = MATRIX(_matrix, 3, 3);
251             r3[7] = 1.0, r3[4] = r3[5] = r3[6] = 0.0f;
252 
253             // choose pivot - or die
254             if (abs(r3[0]) > abs(r2[0]))
255             {
256                 SWAP_ROWS(r3, r2);
257             }
258 
259             if (abs(r2[0]) > abs(r1[0]))
260             {
261                 SWAP_ROWS(r2, r1);
262             }
263 
264             if (abs(r1[0]) > abs(r0[0]))
265             {
266                 SWAP_ROWS(r1, r0);
267             }
268 
269             if (0.0f == r0[0])
270             {
271                 if (invertible)
272                 {
273                     *invertible = false;
274                 }
275                 return result;
276             }
277 
278             // eliminate first variable
279             m1 = r1[0] / r0[0];
280             m2 = r2[0] / r0[0];
281             m3 = r3[0] / r0[0];
282 
283             s  = r0[1];
284             r1[1] -= m1 * s;
285             r2[1] -= m2 * s;
286             r3[1] -= m3 * s;
```


© Ágoston Róth, 2018

```
274             s       = r0[2];
275             r1[2] -= m1 * s;
276             r2[2] -= m2 * s;
277             r3[2] -= m3 * s;

278             s       = r0[3];
279             r1[3] -= m1 * s;
280             r2[3] -= m2 * s;
281             r3[3] -= m3 * s;

282             s = r0[4];
283             if (s != 0.0f)
284             {
285                 r1[4] -= m1 * s;
286                 r2[4] -= m2 * s;
287                 r3[4] -= m3 * s;
288             }

289             s = r0[5];
290             if (s != 0.0f)
291             {
292                 r1[5] -= m1 * s;
293                 r2[5] -= m2 * s;
294                 r3[5] -= m3 * s;
295             }

296             s = r0[6];
297             if (s != 0.0f)
298             {
299                 r1[6] -= m1 * s;
300                 r2[6] -= m2 * s;
301                 r3[6] -= m3 * s;
302             }
303             s = r0[7];
304             if (s != 0.0f)
305             {
306                 r1[7] -= m1 * s;
307                 r2[7] -= m2 * s;
308                 r3[7] -= m3 * s;
309             }

310             // choose pivot - or die
311             if (abs(r3[1]) > abs(r2[1]))
312             {
313                 SWAP_ROWS(r3, r2);
314             }

315             if (abs(r2[1]) > abs(r1[1]))
316             {
317                 SWAP_ROWS(r2, r1);
318             }

319             if (0.0f == r1[1])
320             {
321                 if (invertible)
322                 {
323                     *invertible = false;
324                 }
325                 return result;
326             }

327             // eliminate second variable
328             m2 = r2[1] / r1[1];
329             m3 = r3[1] / r1[1];

330             r2[2] -= m2 * r1[2];
331             r3[2] -= m3 * r1[2];
332             r2[3] -= m2 * r1[3];
333             r3[3] -= m3 * r1[3];

334             s = r1[4];
335             if (0.0f != s)
336             {
337                 r2[4] -= m2 * s;
```


© Ágoston Róth, 2018

```
338             r3[4] -= m3 * s;
339         }
340         s = r1[5];
341         if (0.0f != s)
342         {
343             r2[5] -= m2 * s;
344             r3[5] -= m3 * s;
345         }
346         s = r1[6];
347         if (0.0f != s)
348         {
349             r2[6] -= m2 * s;
350             r3[6] -= m3 * s;
351         }
352         s = r1[7];
353         if (0.0f != s)
354         {
355             r2[7] -= m2 * s;
356             r3[7] -= m3 * s;
357         }
358         // choose pivot - or die
359         if (abs(r3[2]) > abs(r2[2]))
360         {
361             SWAP_ROWS(r3, r2);
362         }
363         if (0.0f == r2[2])
364         {
365             if (invertible)
366             {
367                 *invertible = false;
368             }
369             return result;
370         }
371         // eliminate third variable
372         m3 = r3[2] / r2[2];
373         r3[3] -= m3 * r2[3], r3[4] -= m3 * r2[4],
374         r3[5] -= m3 * r2[5], r3[6] -= m3 * r2[6], r3[7] -= m3 * r2[7];
375         // last check
376         if (0.0f == r3[3])
377         {
378             if (invertible)
379             {
380                 *invertible = false;
381             }
382             return result;
383         }
384         s       = 1.0f / r3[3];                 // now back substitute row 3
385         r3[4] *= s;
386         r3[5] *= s;
387         r3[6] *= s;
388         r3[7] *= s;
389         m2     = r2[3];                         // now back substitute row 2
390         s      = 1.0f / r2[2];
391         r2[4] = s * (r2[4] - r3[4] * m2), r2[5] = s * (r2[5] - r3[5] * m2),
392         r2[6] = s * (r2[6] - r3[6] * m2), r2[7] = s * (r2[7] - r3[7] * m2);
393         m1     = r1[3];
394         r1[4] -= r3[4] * m1, r1[5] -= r3[5] * m1,
395         r1[6] -= r3[6] * m1, r1[7] -= r3[7] * m1;
396         m0     = r0[3];
397         r0[4] -= r3[4] * m0, r0[5] -= r3[5] * m0,
398         r0[6] -= r3[6] * m0, r0[7] -= r3[7] * m0;
399         m1     = r1[2];                         // now back substitute row 1
```


© Ágoston Róth, 2018

```
400             s      = 1.0f / r1[1];
401             r1[4] = s * (r1[4] - r2[4] * m1), r1[5] = s * (r1[5] - r2[5] * m1),
402             r1[6] = s * (r1[6] - r2[6] * m1), r1[7] = s * (r1[7] - r2[7] * m1);

403             m0     = r0[2];
404             r0[4] -= r2[4] * m0, r0[5] -= r2[5] * m0,
405             r0[6] -= r2[6] * m0, r0[7] -= r2[7] * m0;

406             m0     = r0[1];                          // now back substitute row 0
407             s      = 1.0f / r0[0];
408             r0[4] = s * (r0[4] - r1[4] * m0), r0[5] = s * (r0[5] - r1[5] * m0),
409             r0[6] = s * (r0[6] - r1[6] * m0), r0[7] = s * (r0[7] - r1[7] * m0);

410             MATRIX(result._matrix, 0, 0) = r0[4];
411             MATRIX(result._matrix, 0, 1) = r0[5];
412             MATRIX(result._matrix, 0, 2) = r0[6];
413             MATRIX(result._matrix, 0, 3) = r0[7];

414             MATRIX(result._matrix, 1, 0) = r1[4];
415             MATRIX(result._matrix, 1, 1) = r1[5];
416             MATRIX(result._matrix, 1, 2) = r1[6];
417             MATRIX(result._matrix, 1, 3) = r1[7];

418             MATRIX(result._matrix, 2, 0) = r2[4];
419             MATRIX(result._matrix, 2, 1) = r2[5];
420             MATRIX(result._matrix, 2, 2) = r2[6];
421             MATRIX(result._matrix, 2, 3) = r2[7];

422             MATRIX(result._matrix, 3, 0) = r3[4];
423             MATRIX(result._matrix, 3, 1) = r3[5];
424             MATRIX(result._matrix, 3, 2) = r3[6];
425             MATRIX(result._matrix, 3, 3) = r3[7];

426             if (invertible)
427             {
428                 *invertible = true;
429             }
430             return result;

431         #undef MATRIX
432         #undef SWAP_ROWS
433     }

434     // returns the constant memory address of the stored matrix
435     const GLfloat* GLTransformation::address() const
436     {
437         return _matrix;
438     }

439     // clone function required by smart pointers based on the deep copy ownership policy
440     GLTransformation* GLTransformation::clone() const
441     {
442         return new (nothrow) GLTransformation(*this);
443     }

444     // virtual default destructor
445     GLTransformation::~GLTransformation() = default;

446     // multiplicate the transformation matrix rhs by a constant from left
447     const GLTransformation operator *(GLfloat lhs, const GLTransformation &rhs)
448     {
449         GLTransformation result(rhs);

450         #pragma omp parallel for
451         for (GLint i = 0; i < 16; i++)
452         {
453             result._matrix[i] *= lhs;
454         }

455         return result;
456     }

457     // output to stream
458     std::ostream& operator <<(std::ostream &lhs, const GLTransformation &rhs)
```


© Ágoston Róth, 2018

```
459     {
460         for (GLint i = 0; i < 16; i++)
461         {
462             lhs << rhs[i] << " ";
463         }
464         return lhs;
465     }
466     // input from stream
467     std::istream& operator >>(std::istream &lhs, GLTransformation &rhs)
468     {
469         for (GLint i = 0; i < 16; i++)
470         {
471             lhs >> rhs[i];
472         }
473         return lhs;
474     }
475 }
```

Structures of the derived transformation matrices Translate, Scale, Rotate, PerspectiveProjection, OrthogonalProjection and LookAt are illustrated in Fig. 2.18/119, while their definitions and implementations are presented in Listings 2.30/119 and 2.31/122, respectively.



Fig. 2.18: Diagrams of special OpenGL transformations

© Ágoston Róth, 2018

Listing 2.30. Special OpenGL transformations (**Core/Math/SpecialGLTransformations.h**)

```
#ifndef SPECIALGLTRANSFORMATIONS_H
#define SPECIALGLTRANSFORMATIONS_H

#include "GenericGLTransformations.h"
#include "Constants.h"

namespace cagd
{
    // Can be used to define translation matrices.
    class Translate: public GLTransformation
    {
    public:
        // default/special constructor
        Translate(GLfloat ux = 0.0f, GLfloat uy = 0.0f, GLfloat uz = 0.0f);

        // special constructor, the given direction vector will be normalized
        Translate(Cartesian3 &direction, GLfloat distance);

        // setters
        GLvoid setXDirectionalUnits(GLfloat ux);
        GLvoid setYDirectionalUnits(GLfloat uy);
        GLvoid setZDirectionalUnits(GLfloat uz);

        // redeclared clone function required by smart pointers based on the deep copy ownership policy
        Translate* clone() const;
    };

    // Can be used to define scaling transformations.
    class Scale: public GLTransformation
    {
    public:
        // default/special constructor
        Scale(GLfloat sx = 1.0f, GLfloat sy = 1.0f, GLfloat sz = 1.0f);

        // sets the scaling factors along axes x, y and z
        GLvoid setScalingFactors(GLfloat sx, GLfloat sy, GLfloat sz);

        // redeclared clone function required by smart pointers based on the deep copy ownership policy
        Scale* clone() const;
    };

    // Can be used to define rotation matrices.
    class Rotate: public GLTransformation
    {
    protected:
        Cartesian3 _direction;
        GLfloat    _angle_in_radians;

    public:
        // default/special constructor
        Rotate(const Cartesian3 &direction = Cartesian3(1.0, 0.0, 0.0),
               GLfloat angle_in_radians = 0.0f);

        // sets the unit direction vector of the rotation axis
        GLvoid setDirection(const Cartesian3 &direction);

        // sets the rotation angle in radians
        GLvoid setAngle(GLfloat angle_in_radians);

        // redeclared clone function required by smart pointers based on the deep copy ownership policy
        Rotate* clone() const;
    };

    // Can be used to define perspective projection matrices.
    class PerspectiveProjection: public GLTransformation
    {
    protected:
        GLfloat _aspect;
        GLfloat _fov;
        GLfloat _f;
        GLfloat _near;
```


© Ágoston Róth, 2018

```
58          GLfloat _far;

59      public:
60          // special constructor
61          PerspectiveProjection(
62                  GLfloat aspect,
63                  GLfloat fov = 45.0f * DEG_TO_RADIAN,
64                  GLfloat near = 1.0f, GLfloat far = 1000.0f);

65          // setters
66          GLvoid setAspectRatio(GLfloat aspect);
67          GLvoid setFieldOfView(GLfloat fov);
68          GLvoid setNearClippingPlaneDistance(GLfloat near);
69          GLvoid setFarClippingPlaneDistance(GLfloat far);

70          // redeclared clone function required by smart pointers based on the deep copy ownership policy
71          PerspectiveProjection* clone() const;
72      };

73      // Can be used to define orthogonal projection matrices.
74      class OrthogonalProjection: public GLTransformation
75      {
76      protected:
77          GLfloat _aspect;
78          GLfloat _x_min, _x_max;
79          GLfloat _y_min, _y_max;
80          GLfloat _z_min, _z_max;

81      public:
82          // special constructor
83          OrthogonalProjection(
84              GLfloat aspect,
85              GLfloat x_min, GLfloat x_max,
86              GLfloat y_min, GLfloat y_max,
87              GLfloat z_min, GLfloat z_max);

88          // setters
89          GLvoid setAspectRatio(GLfloat aspect);

90          GLvoid setXMin(GLfloat x_min);
91          GLvoid setXMax(GLfloat x_max);

92          GLvoid setYMin(GLfloat y_min);
93          GLvoid setYMax(GLfloat y_max);

94          GLvoid setZMin(GLfloat z_min);
95          GLvoid setZMax(GLfloat z_max);

96          // redeclared clone function required by smart pointers based on the deep copy ownership policy
97          OrthogonalProjection* clone() const;
98      };

99      // Can be used to define world coordinate transformations.
100     class LookAt: public GLTransformation
101     {
102     protected:
103         Cartesian3 _eye;
104         Cartesian3 _center;
105         Cartesian3 _up;

106     public:
107         // special constructor
108         LookAt(const Cartesian3 &eye, const Cartesian3 &center, const Cartesian3 &up);

109         // redeclared clone function required by smart pointers based on the deep copy ownership policy
110         LookAt* clone() const;
111     };
112 }

113 #endif // SPECIALGLTRANSFORMATIONS_H
```


© Ágoston Róth, 2018

Listing 2.31. Special OpenGL transformations (**Core/Math/SpecialGLTransformations.cpp**)

```
#include "SpecialGLTransformations.h"

namespace cagd
{
    // Implementation of translation matrices.

    // default/special constructor
    Translate::Translate(GLfloat ux, GLfloat uy, GLfloat uz): GLTransformation()
    {
        _matrix[12] = ux;
        _matrix[13] = uy;
        _matrix[14] = uz;
    }

    // special constructor, the given direction vector will be normalized
    Translate::Translate(Cartesian3 &direction, GLfloat distance): GLTransformation()
    {
        direction.normalize();
        _matrix[12] = (GLfloat)direction[0] * distance;
        _matrix[13] = (GLfloat)direction[1] * distance;
        _matrix[14] = (GLfloat)direction[2] * distance;
    }

    // setters
    GLvoid Translate::setXDirectionalUnits(GLfloat ux)
    {
        _matrix[12] = ux;
    }

    GLvoid Translate::setYDirectionalUnits(GLfloat uy)
    {
        _matrix[13] = uy;
    }

    GLvoid Translate::setZDirectionalUnits(GLfloat uz)
    {
        _matrix[14] = uz;
    }

    // redefined clone function required by smart pointers based on the deep copy ownership policy
    Translate* Translate::clone() const
    {
        return new Translate(*this);
    }

    // Implementation of scaling transformation.

    // default/special constructor
    Scale::Scale(GLfloat sx, GLfloat sy, GLfloat sz): GLTransformation()
    {
        _matrix[ 0] = sx;
        _matrix[ 5] = sy;
        _matrix[10] = sz;
    }

    // sets the scaling factors along axes x, y and z
    GLvoid Scale::setScalingFactors(GLfloat sx, GLfloat sy, GLfloat sz)
    {
        _matrix[ 0] = sx;
        _matrix[ 5] = sy;
        _matrix[10] = sz;
    }

    // redefined clone function required by smart pointers based on the deep copy ownership policy
    Scale* Scale::clone() const
    {
        return new Scale(*this);
    }

    // Implementation of rotation matrices.
```


© Ágoston Róth, 2018

```
59      // default/special constructor
60      Rotate::Rotate(const Cartesian3 &direction, GLfloat angle_in_radians):
61          GLTransformation(),
62          _direction(direction), _angle_in_radians(angle_in_radians)
63      {
64          // rotation R will correspond to the given axis of unit direction and the rotation angle given in radians
65          //
66          // we will use the formula R = I3 - sin(angle_in_radians) * S + (1.0 - cos(angle_in_radians)) * S^2,
67          // where I3 is the 3 x 3 identity matrix, while S = [ 0 direction[2] -direction[1] ; -direction[2] 0 direction[0] ; direction[1] -direction[0] 0 ]
68          // is a skew-symmetric matrix
69          _direction.normalize();
70          GLTransformation S;
71          S.loadNullMatrix();
72          S[4] =  (GLfloat)_direction[2];
73          S[8] = -(GLfloat)_direction[1];
74          S[1] = -(GLfloat)_direction[2];
75          S[9] =  (GLfloat)_direction[0];
76          S[2] =  (GLfloat)_direction[1];
77          S[6] = -(GLfloat)_direction[0];
78          GLTransformation S2 = S * S;
79          S  *= -sin(_angle_in_radians);
80          S2 *= 1.0f - cos(_angle_in_radians);
81          *this += S;
82          *this += S2;
83      }
84      // sets the unit direction vector of the rotation axis
85      GLvoid Rotate::setDirection(const Cartesian3 &direction)
86      {
87          _direction = direction;
88          _direction.normalize();
89          GLTransformation S;
90          S.loadNullMatrix();
91          S[4] =  (GLfloat)_direction[2];
92          S[8] = -(GLfloat)_direction[1];
93          S[1] = -(GLfloat)_direction[2];
94          S[9] =  (GLfloat)_direction[0];
95          S[2] =  (GLfloat)_direction[1];
96          S[6] = -(GLfloat)_direction[0];
97          GLTransformation S2 = S * S;
98          S  *= -sin(_angle_in_radians);
99          S2 *= 1.0f - cos(_angle_in_radians);
100         loadIdentity();
101         *this += S;
102         *this += S2;
103     }
104     // sets the rotation angle in radians
105     GLvoid Rotate::setAngle(GLfloat angle_in_radians)
106     {
107         _angle_in_radians = angle_in_radians;
108         GLTransformation S;
109         S.loadNullMatrix();
110         S[4] =  (GLfloat)_direction[2];
111         S[8] = -(GLfloat)_direction[1];
112         S[1] = -(GLfloat)_direction[2];
113         S[9] =  (GLfloat)_direction[0];
114         S[2] =  (GLfloat)_direction[1];
115         S[6] = -(GLfloat)_direction[0];
```


© Ágoston Róth, 2018

```
116         GLTransformation S2 = S * S;

117         S  *= -sin(_angle_in_radians);
118         S2 *= 1.0f - cos(_angle_in_radians);

119         loadIdentity();

120         *this += S;
121         *this += S2;
122     }

123     // redefined clone function required by smart pointers based on the deep copy ownership policy
124     Rotate* Rotate::clone() const
125     {
126         return new Rotate(*this);
127     }

128     // Implementation of perspective projection matrices.

129     // special constructor
130     PerspectiveProjection::PerspectiveProjection(
131             GLfloat aspect, GLfloat fov, GLfloat near, GLfloat far):
132         _aspect(aspect), _fov(fov), _f(1.0f / tan(_fov / 2.0f)), _near(near), _far(far)
133     {
134         _matrix[ 0] =  _f / _aspect;
135         _matrix[ 5] =  _f;
136         _matrix[10] =  (_near + _far) / (_near - _far);
137         _matrix[11] = -1.0f;
138         _matrix[14] =  2.0f * _near * _far / (_near - _far);
139         _matrix[15] =  0.0f;
140     }

141     // setters
142     GLvoid PerspectiveProjection::setAspectRatio(GLfloat aspect)
143     {
144         _aspect    = aspect;
145         _matrix[0] = _f / _aspect;
146     }

147     GLvoid PerspectiveProjection::setFieldOfView(GLfloat fov)
148     {
149         _fov       = fov;
150         _f         = 1.0f / tan(_fov / 2.0f);
151         _matrix[0] = _f / _aspect;
152         _matrix[5] = _f;
153     }

154     GLvoid PerspectiveProjection::setNearClippingPlaneDistance(GLfloat near)
155     {
156         _near      = near;
157         _matrix[10] = (_near + _far) / (_near - _far);
158         _matrix[14] = 2.0f * _near * _far / (_near - _far);
159     }

160     GLvoid PerspectiveProjection::setFarClippingPlaneDistance(GLfloat far)
161     {
162         _far       = far;
163         _matrix[10] = (_near + _far) / (_near - _far);
164         _matrix[14] = 2.0f * _near * _far / (_near - _far);
165     }

166     // redefined clone function required by smart pointers based on the deep copy ownership policy
167     PerspectiveProjection* PerspectiveProjection::clone() const
168     {
169         return new PerspectiveProjection(*this);
170     }

171     // Implementation of orthogonal projection matrices.

172     // special constructor
173     OrthogonalProjection::OrthogonalProjection(
174             GLfloat aspect,
175             GLfloat x_min, GLfloat x_max,
```


© Ágoston Róth, 2018

```
176             GLfloat y_min, GLfloat y_max,
177             GLfloat z_min, GLfloat z_max):
178         _aspect(aspect),
179         _x_min(x_min), _x_max(x_max),
180         _y_min(y_min), _y_max(y_max),
181         _z_min(z_min), _z_max(z_max)
182     {
183         _matrix[ 0] =  2.0f / (_x_max - _x_min) / _aspect;
184         _matrix[ 5] =  2.0f / (_y_max - _y_min);
185         _matrix[10] = -2.0f / (_z_max - _z_min);
186         _matrix[12] = (_x_min + _x_max) / (_x_min - _x_max);
187         _matrix[13] = (_y_min + _y_max) / (_y_min - _y_max);
188         _matrix[14] = (_z_min + _z_max) / (_z_min - _z_max);
189     }

190     // setters
191     GLvoid OrthogonalProjection::setAspectRatio(GLfloat aspect)
192     {
193         _aspect    = aspect;
194         _matrix[0] = 2.0f / (_x_max - _x_min) / _aspect;
195     }

196     GLvoid OrthogonalProjection::setXMin(GLfloat x_min)
197     {
198         _x_min      = x_min;
199         _matrix[ 0] = 2.0f / (_x_max - _x_min) / _aspect;
200         _matrix[12] = (_x_min + _x_max) / (_x_min - _x_max);
201     }

202     GLvoid OrthogonalProjection::setXMax(GLfloat x_max)
203     {
204         _x_max      = x_max;
205         _matrix[ 0] = 2.0f / (_x_max - _x_min) / _aspect;
206         _matrix[12] = (_x_min + _x_max) / (_x_min - _x_max);
207     }

208     GLvoid OrthogonalProjection::setYMin(GLfloat y_min)
209     {
210         _y_min      = y_min;
211         _matrix[ 5] = 2.0f / (_y_max - _y_min);
212         _matrix[13] = (_y_min + _y_max) / (_y_min - _y_max);
213     }

214     GLvoid OrthogonalProjection::setYMax(GLfloat y_max)
215     {
216         _y_max      = y_max;
217         _matrix[ 5] = 2.0f / (_y_max - _y_min);
218         _matrix[13] = (_y_min + _y_max) / (_y_min - _y_max);
219     }

220     GLvoid OrthogonalProjection::setZMin(GLfloat z_min)
221     {
222         _z_min      = z_min;
223         _matrix[10] = 2.0f / (_z_max - _z_min);
224         _matrix[14] = (_z_min + _z_max) / (_z_min - _z_max);
225     }

226     GLvoid OrthogonalProjection::setZMax(GLfloat z_max)
227     {
228         _z_max      = z_max;
229         _matrix[10] = 2.0f / (_z_max - _z_min);
230         _matrix[14] = (_z_min + _z_max) / (_z_min - _z_max);
231     }

232     // redefined clone function required by smart pointers based on the deep copy ownership policy
233     OrthogonalProjection* OrthogonalProjection::clone() const
234     {
235         return new OrthogonalProjection(*this);
236     }

237     // Implementation of world coordinate transformations.

238     // special constructor
239     LookAt::LookAt(const Cartesian3 &eye, const Cartesian3 &center, const Cartesian3 &up):
```


© Ágoston Róth, 2018

```
240         _eye(eye), _center(center), _up(up)
241     {
242         Cartesian3 look(_eye);
243         look -= _center;
244         look.normalize();
245         Cartesian3 right(_up);
246         right ^= look;
247         right.normalize();
248         _up = look;
249         _up ^= right;
250         // first column
251         _matrix[0] = (GLfloat)right[0];
252         _matrix[1] = (GLfloat)  _up[0];
253         _matrix[2] = (GLfloat) look[0];
254         _matrix[3] = 0.0f;
255         // second column
256         _matrix[4] = (GLfloat)right[1];
257         _matrix[5] = (GLfloat)  _up[1];
258         _matrix[6] = (GLfloat) look[1];
259         _matrix[7] = 0.0f;
260         // third column
261         _matrix[ 8] = (GLfloat)right[2];
262         _matrix[ 9] = (GLfloat)  _up[2];
263         _matrix[10] = (GLfloat) look[2];
264         _matrix[11] = 0.0f;
265         // fourth column
266         _matrix[12] = -(GLfloat)(right * _eye);
267         _matrix[13] = -(GLfloat)(  _up * _eye);
268         _matrix[14] = -(GLfloat)( look * _eye);
269         _matrix[15] = 1.0f;
270     }
271     // redefined clone function required by smart pointers based on the deep copy ownership policy
272     LookAt* LookAt::clone() const
273     {
274         return new LookAt(*this);
275     }
276 }
```

2.10 Constants and utility functions

Using some of the classes detailed so far, we also provide some frequently used constants and utility functions in Listings 2.32/126 and 2.33/127/2.34/127, respectively.

Listing 2.32. Constants (**Core/Math/Constants.h**)

```
1  #ifndef CONSTANTS_H
2  #define CONSTANTS_H
3  #include "PascalTriangles.h"
4  #include <limits>
5  namespace cagd
6  {
7      const double          PI              = 3.14159265358979323846264338327950;
8      const double          TWO_PI          = 2.0 * PI;
9      const double          HALF_PI         = PI / 2.0;
10     const double          DEG_TO_RADIAN   = PI / 180.0;
11     const double          EPS             = 1.0e-9;
12     const double          MACHINE_EPS     = std::numeric_limits<double>::epsilon();
13     const double          TINY            = std::numeric_limits<double>::min();
```


© Ágoston Róth, 2018

```
14      const int             MAX_DIFF_ORDER = 500; // maximal differentiation order
15      const PascalTriangle BC(MAX_DIFF_ORDER);    // Pascal triangle of binomial coefficients

16      namespace variable
17      {
18          enum Type{U = 0, V = 1};
19      }
20  }

21  #endif // CONSTANTS_H
```

Listing 2.33. Utility functions (**Core/Utilities.h**)

```
1   #ifndef UTILITIES_H
2   #define UTILITIES_H

3   #include "Geometry/Coordinates/Colors4.h"

4   #include <sstream>

5   namespace cagd
6   {
7       bool         platformIsSupported();
8       Color4       coldToHotColormap(GLfloat value, GLfloat min_value, GLfloat max_value);
9       const char* openGLTypeToString(GLenum type);

10      template <typename T>
11      std::string toString(const T &value)
12      {
13          std::ostringstream stream;
14          stream << value;
15          return stream.str();
16      }
17  }

18  #endif // UTILITIES_H
```

Listing 2.34. Utility functions (**Core/Utilities.cpp**)

```
1   #include "Utilities.h"

2   #include <omp.h>
3   #include <GL/glew.h>

4   namespace cagd
5   {
6       bool platformIsSupported()
7       {
8           return (omp_get_max_threads() > 1 && glewIsSupported("GL_VERSION_3_0"));
9       }

10      // 
11      Color4 coldToHotColormap(GLfloat value, GLfloat min_value, GLfloat max_value)
12      {
13          Color4 color(1.0, 1.0, 1.0);

14          if (value < min_value)
15          {
16              value = min_value;
17          }

18          if (value > max_value)
19          {
20              value = max_value;
21          }

22          float dv = max_value - min_value;
```


© Ágoston Róth, 2018

```
23          if (value < (min_value + 0.25f * dv))
24          {
25              color.r() = 0.0;
26              color.g() = 4.0f * (value - min_value) / dv;
27          }
28          else
29          {
30              if (value < (min_value + 0.5f * dv))
31              {
32                  color.r() = 0.0f;
33                  color.b() = 1.0f + 4.0f * (min_value + 0.25f * dv - value) / dv;
34              }
35              else
36              {
37                  if (value < (min_value + 0.75f * dv))
38                  {
39                      color.r() = 4.0f * (value - min_value - 0.5f * dv) / dv;
40                      color.b() = 0.0f;
41                  }
42                  else
43                  {
44                      color.g() = 1.0f + 4.0f * (min_value + 0.75f * dv - value) / dv;
45                      color.b() = 0.0f;
46                  }
47              }
48          }

49          return color;
50      }

51      const char* openGLTypeToString(GLenum type)
52      {
53          switch (type)
54          {
55              case GL_FLOAT:                                  return "float";
56              case GL_FLOAT_VEC2:                             return "vec2";
57              case GL_FLOAT_VEC3:                             return "vec3";
58              case GL_FLOAT_VEC4:                             return "vec4";
59              case GL_INT:                                    return "int";
60              case GL_INT_VEC2:                               return "ivec2";
61              case GL_INT_VEC3:                               return "ivec3";
62              case GL_INT_VEC4:                               return "ivec4";
63              case GL_UNSIGNED_INT:                           return "unsigned int";
64              case GL_UNSIGNED_INT_VEC2:                      return "uvec2";
65              case GL_UNSIGNED_INT_VEC3:                      return "uvec3";
66              case GL_UNSIGNED_INT_VEC4:                      return "uvec4";
67              case GL_BOOL:                                   return "bool";
68              case GL_BOOL_VEC2:                              return "bvec2";
69              case GL_BOOL_VEC3:                              return "bvec3";
70              case GL_BOOL_VEC4:                              return "bvec4";
71              case GL_FLOAT_MAT2:                             return "mat2";
72              case GL_FLOAT_MAT3:                             return "mat3";
73              case GL_FLOAT_MAT4:                             return "mat4";
74              case GL_FLOAT_MAT2x3:                           return "mat2x3";
75              case GL_FLOAT_MAT2x4:                           return "mat2x4";
76              case GL_FLOAT_MAT3x2:                           return "mat3x2";
77              case GL_FLOAT_MAT3x4:                           return "mat3x4";
78              case GL_FLOAT_MAT4x2:                           return "mat4x2";
79              case GL_FLOAT_MAT4x3:                           return "mat4x3";
80              case GL_SAMPLER_1D:                             return "sampler1D";
81              case GL_SAMPLER_2D:                             return "sampler2D";
82              case GL_SAMPLER_3D:                             return "sampler3D";
83              case GL_SAMPLER_CUBE:                           return "samplerCube";
84              case GL_SAMPLER_1D_SHADOW:                      return "sampler1DShadow";
85              case GL_SAMPLER_2D_SHADOW:                      return "sampler2DShadow";
86              case GL_SAMPLER_1D_ARRAY:                       return "sampler1DArray";
87              case GL_SAMPLER_2D_ARRAY:                       return "sampler2DArray";
88              case GL_SAMPLER_1D_ARRAY_SHADOW:                return "sampler1DArrayShadow";
89              case GL_SAMPLER_2D_ARRAY_SHADOW:                return "sampler2DArrayShadow";
90              case GL_SAMPLER_2D_MULTISAMPLE:                 return "sampler2DMS";
91              case GL_SAMPLER_2D_MULTISAMPLE_ARRAY:           return "sampler2DMSArray";
92              case GL_SAMPLER_CUBE_SHADOW:                    return "samplerCubeShadow";
93              case GL_SAMPLER_BUFFER:                         return "samplerBuffer";
94              case GL_SAMPLER_2D_RECT:                        return "sampler2DRect";
```


© Ágoston Róth, 2018

```
95          case GL_SAMPLER_2D_RECT_SHADOW:                 return "sampler2DRectShadow";
96          case GL_INT_SAMPLER_1D:                         return "isampler1D";
97          case GL_INT_SAMPLER_2D:                         return "isampler2D";
98          case GL_INT_SAMPLER_3D:                         return "isampler3D";
99          case GL_INT_SAMPLER_CUBE:                       return "isamplerCube";
100         case GL_INT_SAMPLER_1D_ARRAY:                   return "isampler1DArray";
101         case GL_INT_SAMPLER_2D_ARRAY:                   return "isampler2DArray";
102         case GL_INT_SAMPLER_2D_MULTISAMPLE:             return "isampler2DMS";
103         case GL_INT_SAMPLER_2D_MULTISAMPLE_ARRAY:       return "isampler2DMSArray";
104         case GL_INT_SAMPLER_BUFFER:                     return "isamplerBuffer";
105         case GL_INT_SAMPLER_2D_RECT:                    return "isampler2DRect";
106         case GL_UNSIGNED_INT_SAMPLER_1D:                return "usampler1D";
107         case GL_UNSIGNED_INT_SAMPLER_2D:                return "usampler2D";
108         case GL_UNSIGNED_INT_SAMPLER_3D:                return "usampler3D";
109         case GL_UNSIGNED_INT_SAMPLER_CUBE:              return "usamplerCube";
110         case GL_UNSIGNED_INT_SAMPLER_1D_ARRAY:          return "usampler2DArray";
111         case GL_UNSIGNED_INT_SAMPLER_2D_ARRAY:          return "usampler2DArray";
112         case GL_UNSIGNED_INT_SAMPLER_2D_MULTISAMPLE:    return "usampler2DMS";
113         case GL_UNSIGNED_INT_SAMPLER_2D_MULTISAMPLE_ARRAY: return "usampler2DMSArray";
114         case GL_UNSIGNED_INT_SAMPLER_BUFFER:            return "usamplerBuffer";
115         case GL_UNSIGNED_INT_SAMPLER_2D_RECT:           return "usampler2DRect";
116         default:                                        return "other";
117         }
118     }
119 }
```

2.11 Shader programs

In order to render geometries (like control polygons and nets, or generic curves and triangle meshes obtained, e.g., as the images of linear combinations and tensor product surfaces, respectively), we rely on shader programs written in the OpenGL Shading Language. As we will present later on, classes GenericCurve3, TriangleMesh3, BCurve3: public LinearCombination3 and BSurface3: public TensorProductSurface3 provide several rendering methods. Each of them have two variants, the first of these overloaded member functions relies on instances of our class ShaderProgram detailed in Listings 2.35/129 and 2.36/135 below, while the second one assumes that users handle their shader programs in a different way by providing at the same time attribute (like position, color, normal and texture coordinate) locations of proper types as input parameters for this second variant of our rendering methods. The latter variants will return a false value whenever one of the followings happens: the users did not activate their custom shader programs; the given attribute locations cannot be found in the list of active attribute locations, or they exist but are associated with attributes that have incorrect types (positions, colors, normals and texture coordinate attribute variables have to be of types vec3, vec4, vec3 and vec4, respectively); the rendering mode or other input parameters are invalid. For more details on the input and output parameters of our rendering methods consider later the lines 34/158–59/159, 49/172–71/172, 56/187–75/187 and 145/197–168/197 of Listings 2.37/157, 2.40/170, 2.42/185 and 2.44/194, respectively.

Section 3.2/278 provides examples for the usage of our class ShaderProgram and for convenience also describes shader programs for simple (flat) color shading, for two-sided per pixel lighting (that is able to handle user-defined directional, point and spotlights with uniform front and back materials) and another one for reflection lines that are combined with two-sided per pixel lighting. All figures of this user manual were rendered by using these shader programs.

Listing 2.35. Shader programs (**Core/Shaders/ShaderPrograms.h**)

```
1 #ifndef SHADERPROGRAMS_H
2 #define SHADERPROGRAMS_H

3 #include <GL/glew.h>

4 #include <iostream>
```


© Ágoston Róth, 2018

```
5  #include <map>
6  #include <string>
7  #include <vector>

8  #include "../Geometry/Surfaces/Lights.h"
9  #include "../Geometry/Surfaces/Materials.h"

10 namespace cagd
11 {
12     class ShaderProgram
13     {
14         // friend classes that will be discussed in Listings 2.37/157–2.38/159, 2.42/185–2.43/188, 2.40/170–2.41/173
15         // and 2.44/194–2.45/197, respectively
16         friend class GenericCurve3;
17         friend class LinearCombination3;
18         friend class TriangleMesh3;
19         friend class TensorProductSurface3;

20     public:
21         // nested class that represent different types of shaders
22         class Shader
23         {
24             friend class ShaderProgram;

25         public:
26             // possible shader types
27             enum Type{VERTEX = 0,
28                       FRAGMENT,
29                       COMPUTE,
30                       GEOMETRY,
31                       TESS_CONTROL,
32                       TESS_EVALUATION};

33         private:
34             GLuint      _id;
35             GLint       _compile_status;
36             Type        _type;
37             std::string _source;

38         public:
39             // default constructor
40             Shader();

41             // copy constructor
42             Shader(const Shader &shader);

43             // assignment operator
44             Shader& operator =(const Shader &rhs);

45             // creates, loads and compiles from source file
46             GLboolean createLoadAndCompileFromSourceFile(
47                     Type type, const std::string &file_name,
48                     GLboolean logging_is_enabled = GL_FALSE,
49                     std::ostream &output = std::cout);

50             // either deletes, or flags for deletion the shader
51             GLvoid flagForDeletion() const;

52             // destructor
53             ~Shader();
54         };

55     public:
56         // nested class that represents active uniform and attribute variables
57         class ActiveVariable
58         {
59         private:
60             GLenum  _type;
61             GLint   _array_size;
62             GLint   _location;

63         public:
64             // default/special constructor
65             ActiveVariable(GLenum type = GL_NONE, GLint arraySize = 0,
```


© Ágoston Róth, 2018

```
66                              GLint location = -1);

67              // getters
68              GLenum         type() const;
69              GLint          arraySize() const;
70              GLint          location()  const;
71              const GLchar* typeDescription() const;
72          };

73          typedef ActiveVariable Attribute;
74          typedef ActiveVariable Uniform;

75      private:
76          GLuint                              _id;
77          GLint                               _link_status;
78          std::vector<Shader>                 _shader;
79          std::map<std::string, Attribute>    _attribute;
80          std::map<std::string, Uniform>      _uniform;

81          std::string                         _position_attribute_name;
82          std::string                         _normal_attribute_name;
83          std::string                         _color_attribute_name;
84          std::string                         _texture_attribute_name;

85      public:
86          // default constructor
87          //
88          // Creates an empty unlinked shader program and sets the names of position, normal,
89          // color and texture attributes to "position", "normal", "color" and "texture",
90          // respectively.
91          //
92          // If required, these attribute names can be changed by using one of the methods
93          // setPositionAttributeName, setNormalAttributeName, setColorAttributeName or
94          // setTextureAttributeName.
95          //
96          // By default, it is assumed that, in the vertex shader, these attributes are defined as
97          // the input variables:
98          //
99          // in vec3 position;
100         // in vec3 normal;
101         // in vec4 color;
102         // in vec4 texture;
103         //
104         // The reason for this is that our rendering methods are based on vertex buffer objects
105         // that describe:
106         //   - positions and unit normals as packages of 3 floats;
107         //   - (potentially projective) texture elements and colors as packages of 4 floats.
108         //
109         // The rendering methods GenericCurve3::renderDerivatives, LinearCombination3::renderData
110         // and TensorProductSurface3::renderData require only the attribute "position", while the
111         // constant color of the rendered derivatives, control polygons and control nets can be
112         // specified as a uniform variable as it is illustrated in the next simple vertex and
113         // fragment shaders:
114         //
115         //     ----------
116         //     color.vert
117         //     ----------
118         //     uniform mat4 PVM;       // product of projection, view and model matrices
119         //     in      vec3 position; // input attribute
120         //
121         //     void main()
122         //     {
123         //         gl_Position = PVM * vec4(position, 1.0);
124         //     }
125         //
126         //     ----------
127         //     color.frag
128         //     ----------
129         //     uniform vec4 color;            // constant/flat color
130         //     out     vec4 fragment_color;   // output of the fragment shader
131         //
132         //     void main()
133         //     {
134         //         fragment_color = color;
```


© Ágoston Róth, 2018

```
135         //     }
136         //
137         // However, the rendering method TriangleMesh3::render assumes that a successfully
138         // compiled and linked shader program provides active attributes for positions,
139         // unit normals and texture elements, i.e., in this case, one should use vertex and
140         // fragment shaders similar to the next examples:
141         //
142         //     ------------
143         //     shader.vert
144         //     ------------
145         //     uniform mat4 VM;   // product of view and model matrices
146         //     uniform mat4 PVM;  // product of projection, view and model matrices
147         //     uniform mat4 N;    // normal matrix, i.e., transposed inverse of VM
148         //
149         //     // input attributes
150         //     in       vec3 position;
151         //     in       vec3 normal;
152         //     in       vec4 color;
153         //     in       vec4 texture;
154         //
155         //     // outputs of the vertex shader
156         //     out      vec3 interpolated_position;
157         //     out      vec3 interpolated_normal;
158         //     out      vec4 interpolated_color;
159         //     out      vec4 interpolated_texture;
160         //
161         //     void main()
162         //     {
163         //         // transform the normal vector to the eye space, then normalize it
164         //         interpolated_normal = normalize(vec3(N * vec4(normal, 0.0)));
165         //
166         //         // transform the vertex position to the eye space
167         //         // (it may be useful in case of point and spot lights)
168         //         interpolated_position = vec3(VM * vec4(position, 1.0));
169         //
170         //         // interpolate texture coordinates and color components
171         //         interpolated_color    = color;
172         //         interpolated_texture = texture;
173         //
174         //         // convert the given vertex to clip coordinates and pass along
175         //         gl_Position = PVM * vec4(position, 1.0);
176         //     }
177         //
178         //     ------------
179         //     shader.frag
180         //     ------------
181         //     in   vec3 interpolated_position;
182         //     in   vec3 interpolated_normal;
183         //     in   vec4 interpolated_color;
184         //     in   vec4 interpolated_texture;
185         //
186         //     ...
187         //
188         //     out vec4 fragment_color; // output of the fragment shader
189         //
190         //     void main()
191         //     {
192         //         // one should use all interpolated values computed by the vertex shader,
193         //         // otherwise the compiler will optimize out any inactive attributes and
194         //         // the rendering method TriangleMesh3::render will return GL_FALSE,
195         //         // since it will be not able to find the locations of all required
196         //         // active attributes that have to be passed to function calls like
197         //         // glVertexAttribPointer
198         //
199         //         ...
200         //
201         //         fragment_color = ...;
202         //     }
203         ShaderProgram();

204         // copy constructor
205         ShaderProgram(const ShaderProgram &program);

206         // assignment operator
```


© Ágoston Róth, 2018

```
207         ShaderProgram& operator =(const ShaderProgram &rhs);

208         // Tries to create, load and compile a shader from the specified source file.
209         // In case of success, a new Shader object will be inserted at the end of the
210         // std::vector<Shader> _shader.
211         GLboolean attachNewShaderFromSourceFile(
212                 Shader::Type type, const std::string &file_name,
213                 GLboolean logging_is_enabled = GL_FALSE,
214                 std::ostream &output = std::cout);

215         // Attaches an existing pre-compiled shader object to the shader program. The index
216         // corresponds to the insertion order of the underlying shader. If the provided
217         // index is greater than or equal to the size of the std::vector<Shader> _shader, the
218         // method returns GL_FALSE. If the specified shader exists and it is already attached
219         // to the shader program, the method will also return GL_FALSE.
220         GLboolean attachExistingShader(GLuint index);

221         // Detaches an existing shader object from the shader program. The index corresponds
222         // to the insertion order of the underlying shader. In case of success, the shader
223         // will be not deleted, it will simply be not considered in a later linking process
224         // and, if required, it can be reattached later on. If the provided index is greater than
225         // or equal to the size of the std::vector<Shader> _shader, the method returns GL_FALSE.
226         GLboolean detachExistingShader(GLuint index);

227         // Links the attached pre-compiled shader objects. In case of success, the method
228         // also loads the names, types, array sizes and locations of active uniforms and
229         // attributes. The names and associated properties of these variables will
230         // be stored in the private maps std::map<std::string, Uniform> _uniform and
231         // std::map<std::string, Attribute> _attribute.
232         GLboolean linkAttachedShaders(GLboolean logging_is_enabled = GL_FALSE,
233                                       std::ostream &output = std::cout);

234         // Assuming that the shader program was already compiled and successfully linked,
235         // the method enables and validates its usage.
236         GLboolean enable(GLboolean logging_is_enabled = GL_FALSE,
237                          std::ostream &output = std::cout) const;

238         // The method disables the underlying shader program.
239         GLvoid    disable() const;

240         // getters
241         GLint     positionAttributeLocation() const;
242         GLint     normalAttributeLocation() const;
243         GLint     colorAttributeLocation() const;
244         GLint     textureAttributeLocation() const;

245         // Returns the properties of an active attribute that corresponds to the given name.
246         // If there is no such active attribute, the default attribute object
247         // will be returned - the type, array size and location of which is set
248         // to GL_NONE/"other", 0 and -1, respectively.
249         Attribute activeAttribute(const std::string &name) const;

250         // Returns the properties of an active uniform that corresponds to the given name.
251         // If there is no such active uniform, the default uniform object
252         // will be returned - the type, array size and location of which is set
253         // to GL_NONE/"other", 0 and -1, respectively.
254         Uniform   activeUniform(const std::string &name) const;

255         // setters
256         GLvoid    setPositionAttributeName(const std::string &name);
257         GLvoid    setNormalAttributeName(const std::string &name);
258         GLvoid    setColorAttributeName(const std::string &name);
259         GLvoid    setTextureAttributeName(const std::string &name);

260         GLboolean setUniformValue1i(const std::string &name, GLint value) const;
261         GLboolean setUniformValue2i(const std::string &name,
262                                     GLint value_0, GLint value_1) const;
263         GLboolean setUniformValue3i(const std::string &name,
264                                     GLint value_0, GLint value_1, GLint value_2) const;
265         GLboolean setUniformValue4i(const std::string &name,
266                                     GLint value_0, GLint value_1,
267                                     GLint value_2, GLint value_3) const;

268         GLboolean setUniformValue1ui(const std::string &name, GLuint value) const;
```


© Ágoston Róth, 2018

```
269         GLboolean setUniformValue2ui(const std::string &name,
270                                      GLuint value_0, GLuint value_1) const;
271         GLboolean setUniformValue3ui(const std::string &name,
272                                      GLuint value_0, GLuint value_1,
273                                      GLuint value_2) const;
274         GLboolean setUniformValue4ui(const std::string &name,
275                                      GLuint value_0, GLuint value_1,
276                                      GLuint value_2, GLuint value_3) const;

277         GLboolean setUniformValue1f(const std::string &name, GLfloat value) const;
278         GLboolean setUniformValue2f(const std::string &name,
279                                     GLfloat value_0, GLfloat value_1) const;
280         GLboolean setUniformValue3f(const std::string &name,
281                                     GLfloat value_0, GLfloat value_1,
282                                     GLfloat value_2) const;
283         GLboolean setUniformValue4f(const std::string &name,
284                                     GLfloat value_0, GLfloat value_1,
285                                     GLfloat value_2, GLfloat value_3) const;

286         GLboolean setUniformValue1iv(const std::string &name,
287                                      GLsizei count, const GLint *value) const;
288         GLboolean setUniformValue2iv(const std::string &name,
289                                      GLsizei count, const GLint *value) const;
290         GLboolean setUniformValue3iv(const std::string &name,
291                                      GLsizei count, const GLint *value) const;
292         GLboolean setUniformValue4iv(const std::string &name,
293                                      GLsizei count, const GLint *value) const;

294         GLboolean setUniformValue1uiv(const std::string &name,
295                                       GLsizei count, const GLuint *value) const;
296         GLboolean setUniformValue2uiv(const std::string &name,
297                                       GLsizei count, const GLuint *value) const;
298         GLboolean setUniformValue3uiv(const std::string &name,
299                                       GLsizei count, const GLuint *value) const;
300         GLboolean setUniformValue4uiv(const std::string &name,
301                                       GLsizei count, const GLuint *value) const;

302         GLboolean setUniformValue1fv(const std::string &name,
303                                      GLsizei count, const GLfloat *value) const;
304         GLboolean setUniformValue2fv(const std::string &name,
305                                      GLsizei count, const GLfloat *value) const;
306         GLboolean setUniformValue3fv(const std::string &name,
307                                      GLsizei count, const GLfloat *value) const;
308         GLboolean setUniformValue4fv(const std::string &name,
309                                      GLsizei count, const GLfloat *value) const;

310         GLboolean setUniformMatrix2fv(const std::string &name,
311                                       GLsizei count, GLboolean transpose,
312                                       const GLfloat *values) const;
313         GLboolean setUniformMatrix3fv(const std::string &name,
314                                       GLsizei count, GLboolean transpose,
315                                       const GLfloat *values) const;
316         GLboolean setUniformMatrix4fv(const std::string &name,
317                                       GLsizei count, GLboolean transpose,
318                                       const GLfloat *values) const;

319         GLboolean setUniformMatrix2x3fv(const std::string &name,
320                                         GLsizei count, GLboolean transpose,
321                                         const GLfloat *values) const;
322         GLboolean setUniformMatrix3x2fv(const std::string &name,
323                                         GLsizei count, GLboolean transpose,
324                                         const GLfloat *values) const;
325         GLboolean setUniformMatrix2x4fv(const std::string &name,
326                                         GLsizei count, GLboolean transpose,
327                                         const GLfloat *values) const;
328         GLboolean setUniformMatrix4x2fv(const std::string &name,
329                                         GLsizei count, GLboolean transpose,
330                                         const GLfloat *values) const;
331         GLboolean setUniformMatrix3x4fv(const std::string &name,
332                                         GLsizei count, GLboolean transpose,
333                                         const GLfloat *values) const;
334         GLboolean setUniformMatrix4x3fv(const std::string &name,
335                                         GLsizei count, GLboolean transpose,
336                                         const GLfloat *values) const;
```


© Ágoston Róth, 2018

```
337         GLboolean setUniformColor(const std::string &name, const Color4 &color) const;

338         // The shader program assumes that uniform light source variables are of the type
339         //
340         //     struct LightSource
341         //     {
342         //         bool     enabled;
343         //         vec4     position;
344         //         vec4     half_vector;
345         //         vec4     ambient;
346         //         vec4     diffuse;
347         //         vec4     specular;
348         //         float    spot_cos_cutoff;
349         //         float    constant_attenuation;
350         //         float    linear_attenuation;
351         //         float    quadratic_attenuation;
352         //         vec3     spot_direction;
353         //         float    spot_exponent;
354         //     };
355         GLboolean setUniformDirectionalLight(const std::string &name,
356                                              const DirectionalLight &light) const;
357         GLboolean setUniformPointLight(const std::string &name,
358                                        const PointLight &light) const;
359         GLboolean setUniformSpotlight(const std::string &name,
360                                       const Spotlight &light) const;

361         // The shader program assumes that uniform material variables are of the type
362         //
363         //      struct Material
364         //      {
365         //          vec4     ambient;
366         //          vec4     diffuse;
367         //          vec4     specular;
368         //          vec4     emission;
369         //          float    shininess;
370         //      };
371         GLboolean setUniformMaterial(const std::string &name,
372                                      const Material &material) const;

373         // The ambient and diffuse reflection coefficients of the
374         //
375         //      struct Material
376         //      {
377         //          vec4     ambient;
378         //          vec4     diffuse;
379         //          vec4     specular;
380         //          vec4     emission;
381         //          float    shininess;
382         //      };
383         //
384         // will track the specified ambient_and_diffuse_reflection_coefficients, the emission color components
385         // will be set to the fixed values (0.0f, 0.0f, 0.0f, 0.0f), while the specular reflection
386         // coefficients and the shininess will correspond to the remaining user-specified values.
387         GLboolean setUniformColorMaterial(
388                 const std::string &name,
389                 const Color4 &ambient_and_diffuse_reflection_coefficients,
390                 const Color4 &specular_reflection_coefficients =
391                         Color4(0.3f, 0.3f, 0.3f, 0.8f),
392                 GLfloat shininess = 128.0f) const;

393         // destructor
394         virtual    ~ShaderProgram();
395     };
396 }

397 #endif // SHADERPROGRAMS_H
```

Listing 2.36. Shader programs (**Core/Shaders/ShaderPrograms.cpp**)

```
1 #include "ShaderPrograms.h"
2 #include "../Utilities.h"
```


© Ágoston Róth, 2018

```
3  #include <fstream>

4  using namespace std;

5  namespace cagd
6  {
7      ShaderProgram::Shader::Shader(): _id(0), _compile_status(GL_FALSE), _type(VERTEX)
8      {
9      }

10     ShaderProgram::Shader::Shader(const ShaderProgram::Shader &shader):
11         _id(0), _compile_status(GL_FALSE), _type(shader._type), _source(shader._source)
12     {
13         switch (_type)
14         {
15         case VERTEX:
16             _id = glCreateShader(GL_VERTEX_SHADER);
17             break;

18         case FRAGMENT:
19             _id = glCreateShader(GL_FRAGMENT_SHADER);
20             break;

21         case COMPUTE:
22             _id = glCreateShader(GL_COMPUTE_SHADER);
23             break;

24         case GEOMETRY:
25             _id = glCreateShader(GL_GEOMETRY_SHADER);
26             break;

27         case TESS_CONTROL:
28             _id = glCreateShader(GL_TESS_CONTROL_SHADER);
29             break;

30         case TESS_EVALUATION:
31             _id = glCreateShader(GL_TESS_EVALUATION_SHADER);
32             break;
33         }

34         const GLchar *pointer_to_source = _source.c_str();
35         glShaderSource(_id, 1, &pointer_to_source, nullptr);

36         if (shader._compile_status)
37         {
38             glCompileShader(_id);
39             glGetShaderiv(_id, GL_COMPILE_STATUS, &_compile_status);
40         }
41         else
42         {
43             _compile_status = GL_FALSE;
44         }
45     }

46     ShaderProgram::Shader& ShaderProgram::Shader::operator =(
47             const ShaderProgram::Shader &rhs)
48     {
49         if (this != &rhs)
50         {
51             flagForDeletion();

52             _type   = rhs._type;
53             _source = rhs._source;

54             switch (_type)
55             {
56             case VERTEX:
57                 _id = glCreateShader(GL_VERTEX_SHADER);
58                 break;

59             case FRAGMENT:
60                 _id = glCreateShader(GL_FRAGMENT_SHADER);
61                 break;
```


© Ágoston Róth, 2018

```
62              case COMPUTE:
63                  _id = glCreateShader(GL_COMPUTE_SHADER);
64                  break;
65              case GEOMETRY:
66                  _id = glCreateShader(GL_GEOMETRY_SHADER);
67                  break;
68              case TESS_CONTROL:
69                  _id = glCreateShader(GL_TESS_CONTROL_SHADER);
70                  break;
71              case TESS_EVALUATION:
72                  _id = glCreateShader(GL_TESS_EVALUATION_SHADER);
73                  break;
74              }
75              const GLchar *pointer_to_source = _source.c_str();
76              glShaderSource(_id, 1, &pointer_to_source, nullptr);
77              if (rhs._compile_status)
78              {
79                  glCompileShader(_id);
80                  glGetShaderiv(_id, GL_COMPILE_STATUS, &_compile_status);
81              }
82              else
83              {
84                  _compile_status = GL_FALSE;
85              }
86          }
87          return *this;
88      }
89      GLboolean ShaderProgram::Shader::createLoadAndCompileFromSourceFile(
90              Type type, const std::string &file_name,
91              GLboolean logging_is_enabled, ostream &output)
92      {
93          flagForDeletion();
94          _type = type;
95          switch (_type)
96          {
97          case VERTEX:
98              _id = glCreateShader(GL_VERTEX_SHADER);
99              break;
100         case FRAGMENT:
101             _id = glCreateShader(GL_FRAGMENT_SHADER);
102             break;
103         case COMPUTE:
104             _id = glCreateShader(GL_COMPUTE_SHADER);
105             break;
106         case GEOMETRY:
107             _id = glCreateShader(GL_GEOMETRY_SHADER);
108             break;
109         case TESS_CONTROL:
110             _id = glCreateShader(GL_TESS_CONTROL_SHADER);
111             break;
112         case TESS_EVALUATION:
113             _id = glCreateShader(GL_TESS_EVALUATION_SHADER);
114             break;
115         }
116         string shader_type[6] = {"vertex", "fragment", "compute",
117                                  "geometry", "tess-control", "tess-evaluation"};
118         if (!_id)
119         {
```


© Ágoston Róth, 2018

```
120             if (logging_is_enabled)
121             {
122                 output << "Could not create an empty " << shader_type[type]
123                        << " shader object!" << endl;
124             }
125             return GL_FALSE;
126         }
127         fstream file(file_name.c_str(), ios_base::in);
128         if (!file)
129         {
130             if (logging_is_enabled)
131             {
132                 output << "The given file " << file_name << " does not exist!" << endl;
133             }
134             file.close();
135             return GL_FALSE;
136         }
137         if (logging_is_enabled)
138         {
139             string caption = "Source of the " + shader_type[type] + " shader";
140             string delimeter(caption.length(), '-');
141             output << caption << endl << delimeter << endl;
142         }
143         _source.clear();
144         string aux;
145         while (!file.eof())
146         {
147             getline(file, aux, '\n');
148             _source += aux + '\n';
149             if (logging_is_enabled)
150             {
151                 output << "\t" << aux << endl;
152             }
153         }
154         file.close();
155         if (logging_is_enabled)
156         {
157             output << endl;
158         }
159         const GLchar *pointer_to_source = _source.c_str();
160         glShaderSource(_id, 1, &pointer_to_source, nullptr);
161         glCompileShader(_id);
162         glGetShaderiv(_id, GL_COMPILE_STATUS, &_compile_status);
163         if (!_compile_status)
164         {
165             if (logging_is_enabled)
166             {
167                 GLint info_log_length;
168                 glGetShaderiv(_id, GL_INFO_LOG_LENGTH, &info_log_length);
169                 if (info_log_length > 0)
170                 {
171                     GLchar *info_log = new GLchar[info_log_length];
172                     glGetShaderInfoLog(_id, info_log_length, nullptr, info_log);
173                     output << "\t\\begin{" << shader_type[type]
174                            << " shader information log}" << endl << "\t\tid = " << _id
175                            << ", name = " << file_name << endl;
176                     output <<  "\t\t" << info_log << endl;
177                     output << "\t\\end{" << shader_type[type]
178                            << " shader information log}" << endl << endl;
```


© Ágoston Róth, 2018

```
179                     delete[] info_log;
180                 }
181             }
182             glDeleteShader(_id);
183             return GL_FALSE;
184         }
185         if (logging_is_enabled)
186         {
187             output << "The given " << shader_type[type]
188                    << " shader was successfully created, loaded and compiled."
189                    << endl << endl;
190         }
191         return GL_TRUE;
192     }
193     GLvoid ShaderProgram::Shader::flagForDeletion() const
194     {
195         if (_id)
196         {
197             glDeleteShader(_id);
198         }
199     }
200     ShaderProgram::Shader::~Shader()
201     {
202         flagForDeletion();
203     }
204     ShaderProgram::ActiveVariable::ActiveVariable(
205             GLenum type, GLint array_size, GLint location):
206         _type(type), _array_size(array_size), _location(location)
207     {
208     }
209     GLenum ShaderProgram::ActiveVariable::type() const
210     {
211         return _type;
212     }
213     const GLchar *ShaderProgram::ActiveVariable::typeDescription() const
214     {
215         return openGLTypeToString(_type);
216     }
217     GLint ShaderProgram::ActiveVariable::arraySize() const
218     {
219         return _array_size;
220     }
221     GLint ShaderProgram::ActiveVariable::location() const
222     {
223         return _location;
224     }
225     ShaderProgram::ShaderProgram():
226         _id(0), _link_status(GL_FALSE),
227         _position_attribute_name("position"),
228         _normal_attribute_name("normal"),
229         _color_attribute_name("color"),
230         _texture_attribute_name("texture")
231     {
232     }
233     ShaderProgram::ShaderProgram(const ShaderProgram &program):
234         _id(0), _link_status(GL_FALSE),
235         _shader(program._shader),
236         _attribute(program._attribute),
237         _uniform(program._uniform),
238         _position_attribute_name(program._position_attribute_name),
239         _normal_attribute_name(program._normal_attribute_name),
```


© Ágoston Róth, 2018

```
240         _color_attribute_name(program._color_attribute_name),
241         _texture_attribute_name(program._texture_attribute_name)
242     {
243         _id = glCreateProgram();
244         if (_id)
245         {
246             for (GLuint i = 0; i < (GLuint)_shader.size(); i++)
247             {
248                 if (_shader[i]._id)
249                 {
250                     glAttachShader(_id, _shader[i]._id);
251                 }
252             }
253             if (program._link_status)
254             {
255                 glLinkProgram(_id);
256                 glGetProgramiv(_id, GL_LINK_STATUS, &_link_status);
257             }
258         }
259     }
260     ShaderProgram& ShaderProgram::operator =(const ShaderProgram &rhs)
261     {
262         if (this != &rhs)
263         {
264             for (GLuint i = 0; i < _shader.size(); i++)
265             {
266                 if (_shader[i]._id)
267                 {
268                     glDetachShader(_id, _shader[i]._id);
269                     _shader[i].flagForDeletion();
270                 }
271             }
272             if (_id)
273             {
274                 glDeleteProgram(_id);
275                 _id = 0;
276             }
277             _link_status             = GL_FALSE;
278             _shader                  = rhs._shader;
279             _attribute               = rhs._attribute;
280             _uniform                 = rhs._uniform;
281             _position_attribute_name = rhs._position_attribute_name;
282             _normal_attribute_name   = rhs._normal_attribute_name;
283             _color_attribute_name    = rhs._color_attribute_name;
284             _texture_attribute_name  = rhs._texture_attribute_name;
285             _id = glCreateProgram();
286             if (_id)
287             {
288                 for (GLuint i = 0; i < (GLuint)_shader.size(); i++)
289                 {
290                     if (_shader[i]._id)
291                     {
292                         glAttachShader(_id, _shader[i]._id);
293                     }
294                 }
295                 if (rhs._link_status)
296                 {
297                     glLinkProgram(_id);
298                     glGetProgramiv(_id, GL_LINK_STATUS, &_link_status);
299                 }
300             }
301         }
302         return *this;
303     }
```


© Ágoston Róth, 2018

```
304     GLboolean ShaderProgram::attachNewShaderFromSourceFile(
305             Shader::Type type, const string &file_name,
306             GLboolean logging_is_enabled, ostream &output)
307     {
308         if (!_id)
309         {
310             _id = glCreateProgram();
311         }
312         GLuint old_size = (GLuint)_shader.size();
313         _shader.push_back(Shader());
314         if (!_shader[old_size].createLoadAndCompileFromSourceFile(
315                 type, file_name, logging_is_enabled, output))
316         {
317             _shader.resize(old_size);
318             return GL_FALSE;
319         }
320         glAttachShader(_id, _shader[old_size]._id);
321         return GL_TRUE;
322     }
323     GLboolean ShaderProgram::attachExistingShader(GLuint index)
324     {
325         if (!_id || index >= _shader.size())
326         {
327             return GL_FALSE;
328         }
329         GLboolean already_attached = GL_FALSE;
330         GLint attached_shader_count = 0;
331         glGetProgramiv(_id, GL_ATTACHED_SHADERS, &attached_shader_count);
332         if (attached_shader_count)
333         {
334             GLuint *attached_shaders = new GLuint[attached_shader_count];
335             glGetAttachedShaders(_id, attached_shader_count, nullptr, attached_shaders);
336             for (GLuint i = 0; i < (GLuint)attached_shader_count && !already_attached;
337                  i++)
338             {
339                 already_attached = (_shader[index]._id == attached_shaders[i]);
340             }
341             delete [] attached_shaders;
342         }
343         if (already_attached)
344         {
345             return GL_FALSE;
346         }
347         glAttachShader(_id, _shader[index]._id);
348         return GL_TRUE;
349     }
350     GLboolean ShaderProgram::detachExistingShader(GLuint index)
351     {
352         if (!_id || index >= _shader.size())
353         {
354             return GL_FALSE;
355         }
356         GLboolean attached = GL_FALSE;
357         GLint attached_shader_count = 0;
358         glGetProgramiv(_id, GL_ATTACHED_SHADERS, &attached_shader_count);
359         if (attached_shader_count)
```


© Ágoston Róth, 2018

```
360         {
361             GLuint *attached_shaders = new GLuint[attached_shader_count];
362             glGetAttachedShaders(_id, attached_shader_count, nullptr, attached_shaders);
363             for (GLuint i = 0; i < (GLuint)attached_shader_count && !attached; i++)
364             {
365                 attached = (_shader[index]._id == attached_shaders[i]);
366             }
367             delete [] attached_shaders;
368         }
369         if (!attached)
370         {
371             return GL_FALSE;
372         }
373         glDetachShader(_id, _shader[index]._id);
374         return GL_TRUE;
375     }
376     GLboolean ShaderProgram::linkAttachedShaders(
377             GLboolean logging_is_enabled, ostream &output)
378     {
379         if (!_id)
380         {
381             if (logging_is_enabled)
382             {
383                 output << "The shader program object does not exist!" << endl;
384             }
385             return GL_FALSE;
386         }
387         if (logging_is_enabled)
388         {
389             output << "Linking the attached shaders (" << _shader.size() << ")...";
390         }
391         glLinkProgram(_id);
392         glGetProgramiv(_id, GL_LINK_STATUS, &_link_status);
393         if (!_link_status)
394         {
395             if (logging_is_enabled)
396             {
397                 GLint info_log_length = 0;
398                 glGetProgramiv(_id, GL_INFO_LOG_LENGTH, &info_log_length);
399                 if (info_log_length > 0)
400                 {
401                     GLchar *info_log = new GLchar[info_log_length];
402                     glGetProgramInfoLog(_id, info_log_length, nullptr, info_log);
403                     output << "\t\\begin{program information log}" << endl
404                            << "\t\tid = " << _id << endl;
405                     output <<  "\t\t" << info_log << endl;
406                     output << "\t\\end{program information log}" << endl << endl;
407                     delete[] info_log;
408                 }
409                 output << "\tUnsuccessful." << endl << endl;
410             }
411             return GL_FALSE;
412         }
413         if (logging_is_enabled)
414         {
415             output << "\tsuccessful." << endl << endl;
416         }
```


© Ágoston Róth, 2018

```
417         // loading the names and locations of active uniform variables
418         if (logging_is_enabled)
419         {
420             output << "List of active uniform variables:" << endl;
421         }
422         _uniform.clear();
423         GLint uniform_count;
424         glGetProgramiv(_id, GL_ACTIVE_UNIFORMS, &uniform_count);
425         GLsizei max_uniform_name_length;
426         glGetProgramiv(_id, GL_ACTIVE_UNIFORM_MAX_LENGTH, &max_uniform_name_length);
427         GLchar *uniform_name_data = new GLchar[max_uniform_name_length];
428         for (GLint uniform = 0; uniform < uniform_count; uniform++)
429         {
430             GLsizei actual_length = 0;
431             GLint   array_size    = 0;
432             GLenum  type          = 0;
433             glGetActiveUniform(_id, uniform, max_uniform_name_length, &actual_length,
434                                &array_size, &type, uniform_name_data);
435             string name(&uniform_name_data[0], actual_length);
436             GLint  location = glGetUniformLocation(_id, name.c_str());
437             if (logging_is_enabled)
438             {
439                 output << "\tuniform: " << name
440                        << ", type: " << openGLTypeToString(type)
441                        << ", array size: " << array_size
442                        << ", location: " << location << endl;
443             }
444             if (location >= 0)
445             {
446                 _uniform[name] = Uniform(type, array_size, location);
447             }
448         }
449         delete[] uniform_name_data;
450         // loading the names and locations of active attributes
451         if (logging_is_enabled)
452         {
453             output << "List of active attributes:" << endl;
454         }
455         _attribute.clear();
456         GLint attribute_count;
457         glGetProgramiv(_id, GL_ACTIVE_ATTRIBUTES, &attribute_count);
458         GLsizei max_attribute_name_length;
459         glGetProgramiv(_id, GL_ACTIVE_ATTRIBUTE_MAX_LENGTH, &max_attribute_name_length);
460         GLchar *attribute_name_data = new GLchar[max_attribute_name_length];
461         for (GLint attribute = 0; attribute < attribute_count; attribute++)
462         {
463             GLsizei actual_length = 0;
464             GLint   array_size    = 0;
465             GLenum  type          = 0;
466             glGetActiveAttrib(_id, attribute, max_attribute_name_length, &actual_length,
467                               &array_size, &type, attribute_name_data);
468             string name(&attribute_name_data[0], actual_length);
469             GLint  location = glGetAttribLocation(_id, name.c_str());
470             if (logging_is_enabled)
471             {
472                 output << "\tattribute: " << name
473                        << ", type: " << openGLTypeToString(type)
```


© Ágoston Róth, 2018

```
474                         << ", array size: " << array_size
475                         << ", location: " << location << endl;
476             }
477             if (location >= 0)
478             {
479                 _attribute[name] = Attribute(type, array_size, location);
480             }
481         }
482         delete[] attribute_name_data;
483         return GL_TRUE;
484     }
485     GLboolean ShaderProgram::enable(
486             GLboolean logging_is_enabled, std::ostream &output) const
487     {
488         GLboolean all_shaders_were_compiled_successfully = GL_TRUE;
489         for (GLuint i = 0; i < _shader.size() && all_shaders_were_compiled_successfully;
490              i++)
491         {
492             all_shaders_were_compiled_successfully &= _shader[i]._compile_status;
493         }
494         if (_id && _link_status && all_shaders_were_compiled_successfully)
495         {
496             glUseProgram(_id);
497             if (logging_is_enabled)
498             {
499                 output << "Validating the program...";
500             }
501             glValidateProgram(_id);
502             GLint validate_status = 0;
503             glGetProgramiv(_id, GL_VALIDATE_STATUS, &validate_status);
504             if (!validate_status)
505             {
506                 if (logging_is_enabled)
507                 {
508                     GLint info_log_length = 0;
509                     glGetProgramiv(_id, GL_INFO_LOG_LENGTH, &info_log_length);
510                     if (info_log_length > 0)
511                     {
512                         GLchar *info_log = new GLchar[info_log_length];
513                         glGetProgramInfoLog(_id, info_log_length, nullptr, info_log);
514                         output << "\t\\begin{program validation information log}" << endl
515                                << "\t\tid = " << _id << endl;
516                         output <<  "\t\t" << info_log << endl;
517                         output << "\t\\end{program validation information log}" << endl
518                                << endl;
519                         delete[] info_log;
520                         output << "\tUnsuccessful." << endl << endl;
521                     }
522                 }
523                 return GL_FALSE;
524             }
525             if (logging_is_enabled)
526             {
527                 output << "\tsuccessful." << endl << endl;
528             }
529             return GL_TRUE;
530         }
```


© Ágoston Róth, 2018

```
531         return GL_FALSE;
532     }
533     GLvoid ShaderProgram::disable() const
534     {
535         glUseProgram(0);
536     }
537     // getters
538     GLint ShaderProgram::positionAttributeLocation() const
539     {
540         map<string, Attribute>::const_iterator iterator =
541                 _attribute.find(_position_attribute_name);
542         if (iterator == _attribute.end() || iterator->second.type() != GL_FLOAT_VEC3)
543         {
544             return -1;
545         }
546         return iterator->second.location();
547     }
548     GLint ShaderProgram::normalAttributeLocation() const
549     {
550         map<string, Attribute>::const_iterator iterator =
551                 _attribute.find(_normal_attribute_name);
552         if (iterator == _attribute.end() || iterator->second.type() != GL_FLOAT_VEC3)
553         {
554             return -1;
555         }
556         return iterator->second.location();
557     }
558     GLint ShaderProgram::colorAttributeLocation() const
559     {
560         map<string, Attribute>::const_iterator iterator =
561                 _attribute.find(_color_attribute_name);
562         if (iterator == _attribute.end() || iterator->second.type() != GL_FLOAT_VEC4)
563         {
564             return -1;
565         }
566         return iterator->second.location();
567     }
568     GLint ShaderProgram::textureAttributeLocation() const
569     {
570         map<string, Attribute>::const_iterator iterator =
571                 _attribute.find(_texture_attribute_name);
572         if (iterator == _attribute.end() || iterator->second.type() != GL_FLOAT_VEC4)
573         {
574             return -1;
575         }
576         return iterator->second.location();
577     }
578     ShaderProgram::Attribute ShaderProgram::activeAttribute(const string &name) const
579     {
580         map<string, Attribute>::const_iterator iterator = _attribute.find(name);
581         if (iterator == _attribute.end())
582         {
583             return Attribute();
584         }
585         return iterator->second;
586     }
```


© Ágoston Róth, 2018

```
587     ShaderProgram::Uniform ShaderProgram::activeUniform(const string &name) const
588     {
589         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
590         if (iterator == _uniform.end())
591         {
592             return Uniform();
593         }
594         return iterator->second;
595     }
596     // setters
597     GLvoid ShaderProgram::setPositionAttributeName(const string &name)
598     {
599         _position_attribute_name = name;
600     }
601     GLvoid ShaderProgram::setNormalAttributeName(const string &name)
602     {
603         _normal_attribute_name = name;
604     }
605     GLvoid ShaderProgram::setColorAttributeName(const string &name)
606     {
607         _color_attribute_name = name;
608     }
609     GLvoid ShaderProgram::setTextureAttributeName(const string &name)
610     {
611         _texture_attribute_name = name;
612     }
613     GLboolean ShaderProgram::setUniformValue1i(const string &name, GLint value) const
614     {
615         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
616         if (iterator == _uniform.end())
617         {
618             return GL_FALSE;
619         }
620         glUniform1i(iterator->second.location(), value);
621         return GL_TRUE;
622     }
623     GLboolean ShaderProgram::setUniformValue2i(
624         const string &name, GLint value_0, GLint value_1) const
625     {
626         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
627         if (iterator == _uniform.end())
628         {
629             return GL_FALSE;
630         }
631         glUniform2i(iterator->second.location(), value_0, value_1);
632         return GL_TRUE;
633     }
634     GLboolean ShaderProgram::setUniformValue3i(
635         const string &name, GLint value_0, GLint value_1, GLint value2) const
636     {
637         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
638         if (iterator == _uniform.end())
639         {
640             return GL_FALSE;
641         }
```


© Ágoston Róth, 2018

```
642         glUniform3i(iterator->second.location(), value_0, value_1, value2);

643         return GL_TRUE;
644     }

645     GLboolean ShaderProgram::setUniformValue4i(
646         const string &name,
647         GLint value_0, GLint value_1, GLint value2, GLint value_3) const
648     {
649         map<string, Uniform>::const_iterator iterator = _uniform.find(name);

650         if (iterator == _uniform.end())
651         {
652             return GL_FALSE;
653         }

654         glUniform4i(iterator->second.location(), value_0, value_1, value2, value_3);

655         return GL_TRUE;
656     }

657     GLboolean ShaderProgram::setUniformValue1ui(const string &name, GLuint value) const
658     {
659         map<string, Uniform>::const_iterator iterator = _uniform.find(name);

660         if (iterator == _uniform.end())
661         {
662             return GL_FALSE;
663         }

664         glUniform1ui(iterator->second.location(), value);

665         return GL_TRUE;
666     }

667     GLboolean ShaderProgram::setUniformValue2ui(
668         const string &name, GLuint value_0, GLuint value_1) const
669     {
670         map<string, Uniform>::const_iterator iterator = _uniform.find(name);

671         if (iterator == _uniform.end())
672         {
673             return GL_FALSE;
674         }

675         glUniform2ui(iterator->second.location(), value_0, value_1);

676         return GL_TRUE;
677     }

678     GLboolean ShaderProgram::setUniformValue3ui(
679         const string &name, GLuint value_0, GLuint value_1, GLuint value2) const
680     {
681         map<string, Uniform>::const_iterator iterator = _uniform.find(name);

682         if (iterator == _uniform.end())
683         {
684             return GL_FALSE;
685         }

686         glUniform3ui(iterator->second.location(), value_0, value_1, value2);

687         return GL_TRUE;
688     }

689     GLboolean ShaderProgram::setUniformValue4ui(
690         const string &name,
691         GLuint value_0, GLuint value_1, GLuint value2, GLuint value_3) const
692     {
693         map<string, Uniform>::const_iterator iterator = _uniform.find(name);

694         if (iterator == _uniform.end())
695         {
696             return GL_FALSE;
```


© Ágoston Róth, 2018

```
697         }
698         glUniform4ui(iterator->second.location(), value_0, value_1, value2, value_3);
699         return GL_TRUE;
700     }
701     GLboolean ShaderProgram::setUniformValue1f(const string &name, GLfloat value) const
702     {
703         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
704         if (iterator == _uniform.end())
705         {
706             return GL_FALSE;
707         }
708         glUniform1f(iterator->second.location(), value);
709         return GL_TRUE;
710     }
711     GLboolean ShaderProgram::setUniformValue2f(
712         const string &name, GLfloat value_0, GLfloat value_1) const
713     {
714         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
715         if (iterator == _uniform.end())
716         {
717             return GL_FALSE;
718         }
719         glUniform2f(iterator->second.location(), value_0, value_1);
720         return GL_TRUE;
721     }
722     GLboolean ShaderProgram::setUniformValue3f(
723         const string &name, GLfloat value_0, GLfloat value_1, GLfloat value2) const
724     {
725         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
726         if (iterator == _uniform.end())
727         {
728             return GL_FALSE;
729         }
730         glUniform3f(iterator->second.location(), value_0, value_1, value2);
731         return GL_TRUE;
732     }
733     GLboolean ShaderProgram::setUniformValue4f(
734         const string &name,
735         GLfloat value_0, GLfloat value_1, GLfloat value2, GLfloat value_3) const
736     {
737         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
738         if (iterator == _uniform.end())
739         {
740             return GL_FALSE;
741         }
742         glUniform4f(iterator->second.location(), value_0, value_1, value2, value_3);
743         return GL_TRUE;
744     }
745     GLboolean ShaderProgram::setUniformValue1iv(
746         const string &name, GLsizei count, const GLint *value) const
747     {
748         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
749         if (iterator == _uniform.end())
750         {
```


© Ágoston Róth, 2018

```
751             return GL_FALSE;
752         }
753         glUniform1iv(iterator->second.location(), count, value);
754         return GL_TRUE;
755     }
756     GLboolean ShaderProgram::setUniformValue2iv(
757         const string &name, GLsizei count, const GLint *value) const
758     {
759         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
760         if (iterator == _uniform.end())
761         {
762             return GL_FALSE;
763         }
764         glUniform2iv(iterator->second.location(), count, value);
765         return GL_TRUE;
766     }
767     GLboolean ShaderProgram::setUniformValue3iv(
768         const string &name, GLsizei count, const GLint *value) const
769     {
770         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
771         if (iterator == _uniform.end())
772         {
773             return GL_FALSE;
774         }
775         glUniform3iv(iterator->second.location(), count, value);
776         return GL_TRUE;
777     }
778     GLboolean ShaderProgram::setUniformValue4iv(
779         const string &name, GLsizei count, const GLint *value) const
780     {
781         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
782         if (iterator == _uniform.end())
783         {
784             return GL_FALSE;
785         }
786         glUniform4iv(iterator->second.location(), count, value);
787         return GL_TRUE;
788     }
789     GLboolean ShaderProgram::setUniformValue1uiv(
790         const string &name, GLsizei count, const GLuint *value) const
791     {
792         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
793         if (iterator == _uniform.end())
794         {
795             return GL_FALSE;
796         }
797         glUniform1uiv(iterator->second.location(), count, value);
798         return GL_TRUE;
799     }
800     GLboolean ShaderProgram::setUniformValue2uiv(
801         const string &name, GLsizei count, const GLuint *value) const
802     {
803         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
804         if (iterator == _uniform.end())
```


© Ágoston Róth, 2018

```
805         {
806             return GL_FALSE;
807         }
808         glUniform2uiv(iterator->second.location(), count, value);
809         return GL_TRUE;
810     }
811     GLboolean ShaderProgram::setUniformValue3uiv(
812         const string &name, GLsizei count, const GLuint *value) const
813     {
814         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
815         if (iterator == _uniform.end())
816         {
817             return GL_FALSE;
818         }
819         glUniform3uiv(iterator->second.location(), count, value);
820         return GL_TRUE;
821     }
822     GLboolean ShaderProgram::setUniformValue4uiv(
823         const string &name, GLsizei count, const GLuint *value) const
824     {
825         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
826         if (iterator == _uniform.end())
827         {
828             return GL_FALSE;
829         }
830         glUniform4uiv(iterator->second.location(), count, value);
831         return GL_TRUE;
832     }
833     GLboolean ShaderProgram::setUniformValue1fv(
834         const string &name, GLsizei count, const GLfloat *value) const
835     {
836         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
837         if (iterator == _uniform.end())
838         {
839             return GL_FALSE;
840         }
841         glUniform1fv(iterator->second.location(), count, value);
842         return GL_TRUE;
843     }
844     GLboolean ShaderProgram::setUniformValue2fv(
845         const string &name, GLsizei count, const GLfloat *value) const
846     {
847         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
848         if (iterator == _uniform.end())
849         {
850             return GL_FALSE;
851         }
852         glUniform2fv(iterator->second.location(), count, value);
853         return GL_TRUE;
854     }
855     GLboolean ShaderProgram::setUniformValue3fv(
856         const string &name, GLsizei count, const GLfloat *value) const
857     {
858         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
```


© Ágoston Róth, 2018

```
859         if (iterator == _uniform.end())
860         {
861             return GL_FALSE;
862         }
863         glUniform3fv(iterator->second.location(), count, value);
864         return GL_TRUE;
865     }
866     GLboolean ShaderProgram::setUniformValue4fv(
867         const string &name, GLsizei count, const GLfloat *value) const
868     {
869         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
870         if (iterator == _uniform.end())
871         {
872             return GL_FALSE;
873         }
874         glUniform4fv(iterator->second.location(), count, value);
875         return GL_TRUE;
876     }
877     GLboolean ShaderProgram::setUniformMatrix2fv(
878         const std::string &name, GLsizei count,
879         GLboolean transpose, const GLfloat *values) const
880     {
881         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
882         if (iterator == _uniform.end())
883         {
884             return GL_FALSE;
885         }
886         glUniformMatrix2fv(iterator->second.location(), count, transpose, values);
887         return GL_TRUE;
888     }
889     GLboolean ShaderProgram::setUniformMatrix3fv(
890         const std::string &name, GLsizei count,
891         GLboolean transpose, const GLfloat *values) const
892     {
893         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
894         if (iterator == _uniform.end())
895         {
896             return GL_FALSE;
897         }
898         glUniformMatrix3fv(iterator->second.location(), count, transpose, values);
899         return GL_TRUE;
900     }
901     GLboolean ShaderProgram::setUniformMatrix4fv(
902         const std::string &name, GLsizei count,
903         GLboolean transpose, const GLfloat *values) const
904     {
905         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
906         if (iterator == _uniform.end())
907         {
908             return GL_FALSE;
909         }
910         glUniformMatrix4fv(iterator->second.location(), count, transpose, values);
911         return GL_TRUE;
912     }
913     GLboolean ShaderProgram::setUniformMatrix2x3fv(
```


© Ágoston Róth, 2018

```
914         const std::string &name, GLsizei count,
915         GLboolean transpose, const GLfloat *values) const
916     {
917         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
918         if (iterator == _uniform.end())
919         {
920             return GL_FALSE;
921         }
922         glUniformMatrix2x3fv(iterator->second.location(), count, transpose, values);
923         return GL_TRUE;
924     }
925     GLboolean ShaderProgram::setUniformMatrix3x2fv(
926         const std::string &name, GLsizei count,
927         GLboolean transpose, const GLfloat *values) const
928     {
929         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
930         if (iterator == _uniform.end())
931         {
932             return GL_FALSE;
933         }
934         glUniformMatrix3x2fv(iterator->second.location(), count, transpose, values);
935         return GL_TRUE;
936     }
937     GLboolean ShaderProgram::setUniformMatrix2x4fv(
938         const std::string &name, GLsizei count,
939         GLboolean transpose, const GLfloat *values) const
940     {
941         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
942         if (iterator == _uniform.end())
943         {
944             return GL_FALSE;
945         }
946         glUniformMatrix2x4fv(iterator->second.location(), count, transpose, values);
947         return GL_TRUE;
948     }
949     GLboolean ShaderProgram::setUniformMatrix4x2fv(
950         const std::string &name, GLsizei count,
951         GLboolean transpose, const GLfloat *values) const
952     {
953         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
954         if (iterator == _uniform.end())
955         {
956             return GL_FALSE;
957         }
958         glUniformMatrix4x2fv(iterator->second.location(), count, transpose, values);
959         return GL_TRUE;
960     }
961     GLboolean ShaderProgram::setUniformMatrix3x4fv(
962         const std::string &name, GLsizei count,
963         GLboolean transpose, const GLfloat *values) const
964     {
965         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
966         if (iterator == _uniform.end())
967         {
968             return GL_FALSE;
969         }
```


© Ágoston Róth, 2018

```
970         glUniformMatrix3x4fv(iterator->second.location(), count, transpose, values);
971         return GL_TRUE;
972     }
973     GLboolean ShaderProgram::setUniformMatrix4x3fv(
974         const std::string &name, GLsizei count,
975         GLboolean transpose, const GLfloat *values) const
976     {
977         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
978         if (iterator == _uniform.end())
979         {
980             return GL_FALSE;
981         }
982         glUniformMatrix4x3fv(iterator->second.location(), count, transpose, values);
983         return GL_TRUE;
984     }
985     GLboolean ShaderProgram::setUniformColor(
986         const std::string &name, const Color4 &color) const
987     {
988         map<string, Uniform>::const_iterator iterator = _uniform.find(name);
989         if (iterator == _uniform.end())
990         {
991             return GL_FALSE;
992         }
993         glUniform4fv(iterator->second.location(), 1, color.address());
994         return GL_TRUE;
995     }
996     GLboolean ShaderProgram::setUniformDirectionalLight(
997         const string &name, const DirectionalLight &light) const
998     {
999         map<string, Uniform>::const_iterator position_iterator    =
1000                 _uniform.find(name + ".position");
1001         map<string, Uniform>::const_iterator half_vector_iterator =
1002                 _uniform.find(name + ".half_vector");
1003         map<string, Uniform>::const_iterator ambient_iterator     =
1004                 _uniform.find(name + ".ambient");
1005         map<string, Uniform>::const_iterator diffuse_iterator     =
1006                 _uniform.find(name + ".diffuse");
1007         map<string, Uniform>::const_iterator specular_iterator    =
1008                 _uniform.find(name + ".specular");
1009         map<string, Uniform>::const_iterator end = _uniform.end();
1010         if (position_iterator == end || half_vector_iterator == end ||
1011             ambient_iterator  == end || diffuse_iterator     == end ||
1012             specular_iterator == end)
1013         {
1014             return GL_FALSE;
1015         }
1016         glUniform4fv(position_iterator->second.location(), 1,
1017                      light.addressOfPosition());
1018         glUniform4fv(half_vector_iterator->second.location(), 1,
1019                      light.addressOfHalfVector());
1020         glUniform4fv(ambient_iterator->second.location(), 1,
1021                      light.addressOfAmbientIntensity());
1022         glUniform4fv(diffuse_iterator->second.location(), 1,
1023                      light.addressOfDiffuseIntensity());
1024         glUniform4fv(specular_iterator->second.location(), 1,
1025                      light.addressOfSpecularIntensity());
1026         map<string, Uniform>::const_iterator spot_cos_cutoff_iterator       =
1027                 _uniform.find(name + ".spot_cos_cutoff");
1028         map<string, Uniform>::const_iterator constant_attenuation_iterator  =
1029                 _uniform.find(name + ".constant_attenuation");
```


© Ágoston Róth, 2018

```
1030        map<string, Uniform>::const_iterator linear_attenuation_iterator    =
1031                _uniform.find(name + ".linear_attenuation");
1032        map<string, Uniform>::const_iterator quadratic_attenuation_iterator =
1033                _uniform.find(name + ".quadratic_attenuation");
1034        map<string, Uniform>::const_iterator spot_direction_iterator        =
1035                _uniform.find(name + ".spot_direction");
1036        map<string, Uniform>::const_iterator spot_exponent_iterator         =
1037                _uniform.find(name + ".spot_exponent");

1038        if (spot_cos_cutoff_iterator    == end || constant_attenuation_iterator  == end ||
1039            linear_attenuation_iterator == end || quadratic_attenuation_iterator == end ||
1040            spot_direction_iterator     == end || spot_exponent_iterator         == end)
1041        {
1042            return GL_FALSE;
1043        }

1044        glUniform1f(spot_cos_cutoff_iterator->second.location(), 180.0f);
1045        glUniform1f(constant_attenuation_iterator->second.location(), 1.0f);
1046        glUniform1f(linear_attenuation_iterator->second.location(), 0.0f);
1047        glUniform1f(quadratic_attenuation_iterator->second.location(), 0.0f);
1048        glUniform3f(spot_direction_iterator->second.location(), 0.0f, 0.0f, -1.0f);
1049        glUniform1f(spot_exponent_iterator->second.location(), 0.0);

1050        return GL_TRUE;
1051    }

1052    GLboolean ShaderProgram::setUniformPointLight(
1053        const string &name, const PointLight &light) const
1054    {
1055        map<string, Uniform>::const_iterator position_iterator              =
1056                _uniform.find(name + ".position");
1057        map<string, Uniform>::const_iterator half_vector_iterator           =
1058                _uniform.find(name + ".half_vector");
1059        map<string, Uniform>::const_iterator ambient_iterator               =
1060                _uniform.find(name + ".ambient");
1061        map<string, Uniform>::const_iterator diffuse_iterator               =
1062                _uniform.find(name + ".diffuse");
1063        map<string, Uniform>::const_iterator specular_iterator              =
1064                _uniform.find(name + ".specular");
1065        map<string, Uniform>::const_iterator spot_cos_cutoff_iterator       =
1066                _uniform.find(name + ".spot_cos_cutoff");
1067        map<string, Uniform>::const_iterator constant_attenuation_iterator  =
1068                _uniform.find(name + ".constant_attenuation");
1069        map<string, Uniform>::const_iterator linear_attenuation_iterator    =
1070                _uniform.find(name + ".linear_attenuation");
1071        map<string, Uniform>::const_iterator quadratic_attenuation_iterator =
1072                _uniform.find(name + ".quadratic_attenuation");

1073        map<string, Uniform>::const_iterator end = _uniform.end();

1074        if (position_iterator           == end || half_vector_iterator           == end ||
1075            ambient_iterator            == end || diffuse_iterator               == end ||
1076            specular_iterator           == end ||
1077            spot_cos_cutoff_iterator    == end || constant_attenuation_iterator  == end ||
1078            linear_attenuation_iterator == end || quadratic_attenuation_iterator == end)
1079        {
1080            return GL_FALSE;
1081        }

1082        glUniform4fv(position_iterator->second.location(), 1,
1083                     light.addressOfPosition());
1084        glUniform4fv(half_vector_iterator->second.location(), 1,
1085                     light.addressOfHalfVector());
1086        glUniform4fv(ambient_iterator->second.location(), 1,
1087                     light.addressOfAmbientIntensity());
1088        glUniform4fv(diffuse_iterator->second.location(), 1,
1089                     light.addressOfDiffuseIntensity());
1090        glUniform4fv(specular_iterator->second.location(), 1,
1091                     light.addressOfSpecularIntensity());

1092        glUniform1f(spot_cos_cutoff_iterator->second.location(), 180.0f);
1093        glUniform1f(constant_attenuation_iterator->second.location(),
1094                    light.constantAttenuation());
1095        glUniform1f(linear_attenuation_iterator->second.location(),
```


© Ágoston Róth, 2018

```
1096                    light.linearAttenuation());
1097        glUniform1f(quadratic_attenuation_iterator->second.location(),
1098                    light.quadraticAttenuation());
1099
1099        map<string, Uniform>::const_iterator spot_direction_iterator =
1100                _uniform.find(name + ".spot_direction");
1101        map<string, Uniform>::const_iterator spot_exponent_iterator  =
1102                _uniform.find(name + ".spot_exponent");
1103
1103        if (spot_direction_iterator == end || spot_exponent_iterator == end)
1104        {
1105            return GL_FALSE;
1106        }
1107
1107        glUniform3f(spot_direction_iterator->second.location(), 0.0f, 0.0f, -1.0f);
1108        glUniform1f( spot_exponent_iterator->second.location(), 0.0f);
1109
1109        return GL_TRUE;
1110    }
1111
1111    GLboolean ShaderProgram::setUniformSpotlight(
1112        const string &name, const Spotlight &light) const
1113    {
1114        map<string, Uniform>::const_iterator position_iterator              =
1115                _uniform.find(name + ".position");
1116        map<string, Uniform>::const_iterator half_vector_iterator           =
1117                _uniform.find(name + ".half_vector");
1118        map<string, Uniform>::const_iterator ambient_iterator               =
1119                _uniform.find(name + ".ambient");
1120        map<string, Uniform>::const_iterator diffuse_iterator               =
1121                _uniform.find(name + ".diffuse");
1122        map<string, Uniform>::const_iterator specular_iterator              =
1123                _uniform.find(name + ".specular");
1124        map<string, Uniform>::const_iterator spot_cos_cutoff_iterator       =
1125                _uniform.find(name + ".spot_cos_cutoff");
1126        map<string, Uniform>::const_iterator constant_attenuation_iterator  =
1127                _uniform.find(name + ".constant_attenuation");
1128        map<string, Uniform>::const_iterator linear_attenuation_iterator    =
1129                _uniform.find(name + ".linear_attenuation");
1130        map<string, Uniform>::const_iterator quadratic_attenuation_iterator =
1131                _uniform.find(name + ".quadratic_attenuation");
1132        map<string, Uniform>::const_iterator spot_direction_iterator        =
1133                _uniform.find(name + ".spot_direction");
1134        map<string, Uniform>::const_iterator spot_exponent_iterator         =
1135                _uniform.find(name + ".spot_exponent");
1136        map<string, Uniform>::const_iterator end = _uniform.end();
1137        if (position_iterator           == end || half_vector_iterator           == end ||
1138            ambient_iterator            == end || diffuse_iterator               == end ||
1139            specular_iterator           == end ||
1140            spot_cos_cutoff_iterator    == end || constant_attenuation_iterator  == end ||
1141            linear_attenuation_iterator == end || quadratic_attenuation_iterator == end ||
1142            spot_direction_iterator     == end || spot_exponent_iterator         == end)
1143        {
1144            return GL_FALSE;
1145        }
1146        const GLdouble *spot_direction = light.addressOfSpotDirection();
1147        glUniform4fv(position_iterator->second.location(), 1,
1148                     light.addressOfPosition());
1149        glUniform4fv(half_vector_iterator->second.location(), 1,
1150                     light.addressOfHalfVector());
1151        glUniform4fv(ambient_iterator->second.location(), 1,
1152                     light.addressOfAmbientIntensity());
1153        glUniform4fv(diffuse_iterator->second.location(), 1,
1154                     light.addressOfDiffuseIntensity());
1155        glUniform4fv(specular_iterator->second.location(), 1,
1156                     light.addressOfSpecularIntensity());
1157        glUniform1f (spot_cos_cutoff_iterator->second.location(),
1158                     light.spotCosCutoff());
1159        glUniform1f (constant_attenuation_iterator->second.location(),
1160                     light.constantAttenuation());
```


© Ágoston Róth, 2018

```
1161            glUniform1f (linear_attenuation_iterator->second.location(),
1162                         light.linearAttenuation());
1163            glUniform1f (quadratic_attenuation_iterator->second.location(),
1164                         light.quadraticAttenuation());
1165            glUniform3f (spot_direction_iterator->second.location(),
1166                         (GLfloat)spot_direction[0], (GLfloat)spot_direction[1],
1167                         (GLfloat)spot_direction[2]);
1168            glUniform1f (spot_exponent_iterator->second.location(), light.spotExponent());

1169            return GL_TRUE;
1170        }

1171        GLboolean ShaderProgram::setUniformMaterial(
1172            const std::string &name, const Material &material) const
1173        {
1174            map<string, Uniform>::const_iterator ambient_iterator   =
1175                    _uniform.find(name + ".ambient");
1176            map<string, Uniform>::const_iterator diffuse_iterator   =
1177                    _uniform.find(name + ".diffuse");
1178            map<string, Uniform>::const_iterator specular_iterator  =
1179                    _uniform.find(name + ".specular");
1180            map<string, Uniform>::const_iterator emission_iterator  =
1181                    _uniform.find(name + ".emission");
1182            map<string, Uniform>::const_iterator shininess_iterator =
1183                    _uniform.find(name + ".shininess");

1184            map<string, Uniform>::const_iterator end = _uniform.end();

1185            if (ambient_iterator  == end || diffuse_iterator  == end ||
1186                specular_iterator == end || emission_iterator == end ||
1187                shininess_iterator == end)
1188            {
1189                return GL_FALSE;
1190            }

1191            glUniform4fv(ambient_iterator->second.location(), 1,
1192                         material.addressOfAmbientReflectionCoefficients());
1193            glUniform4fv(diffuse_iterator->second.location(), 1,
1194                         material.addressOfDiffuseReflectionCoefficients());
1195            glUniform4fv(specular_iterator->second.location(), 1,
1196                         material.addressOfSpecularReflectionCoefficients());
1197            glUniform4fv(emission_iterator->second.location(), 1,
1198                         material.addressOfEmissionColor());
1199            glUniform1f (shininess_iterator->second.location(),
1200                         material.shininess());

1201            return GL_TRUE;
1202        }

1203        GLboolean ShaderProgram::setUniformColorMaterial(
1204            const std::string &name,
1205            const Color4 &ambient_and_diffuse_reflection_coefficients,
1206            const Color4 &specular_reflection_coefficients,
1207            GLfloat shininess) const
1208        {
1209            map<string, Uniform>::const_iterator ambient_iterator   =
1210                    _uniform.find(name + ".ambient");
1211            map<string, Uniform>::const_iterator diffuse_iterator   =
1212                    _uniform.find(name + ".diffuse");
1213            map<string, Uniform>::const_iterator specular_iterator  =
1214                    _uniform.find(name + ".specular");
1215            map<string, Uniform>::const_iterator emission_iterator  =
1216                    _uniform.find(name + ".emission");
1217            map<string, Uniform>::const_iterator shininess_iterator =
1218                    _uniform.find(name + ".shininess");

1219            map<string, Uniform>::const_iterator end = _uniform.end();

1220            if (ambient_iterator  == end || diffuse_iterator  == end ||
1221                specular_iterator == end || emission_iterator == end ||
1222                shininess_iterator == end)
1223            {
1224                return GL_FALSE;
1225            }
```


© Ágoston Róth, 2018

```
1226            glUniform4fv(ambient_iterator->second.location(), 1,
1227                         ambient_and_diffuse_reflection_coefficients.address());
1228            glUniform4fv(diffuse_iterator->second.location(), 1,
1229                         ambient_and_diffuse_reflection_coefficients.address());
1230            glUniform4fv(specular_iterator->second.location(), 1,
1231                         specular_reflection_coefficients.address());
1232            glUniform4f (emission_iterator->second.location(), 0.0f, 0.0f, 0.0f, 0.0f);
1233            glUniform1f (shininess_iterator->second.location(), shininess);

1234            return GL_TRUE;
1235        }

1236        // destructor
1237        ShaderProgram::~ShaderProgram()
1238        {
1239            for (GLuint i = 0; i < _shader.size(); i++)
1240            {
1241                glDetachShader(_id, _shader[i]._id);
1242                _shader[i].flagForDeletion();
1243            }

1244            if (_id)
1245            {
1246                glDeleteProgram(_id);
1247            }
1248        }
1249    }
```

2.12 Generic curves

In order to store in vertex buffer objects the points and higher order derivatives of arbitrary smooth parametric (basis) functions, (B-)curves and isoparametric lines of (B-)surfaces, we introduce a class for generic curves (GenericCurve3) that will be used for rendering purposes. Its diagram is illustrated in Fig. 2.19/158. Apart from vertex buffer object handling methods and some simple accessors, the class also provides overloaded function operators that can be used for reading or writing the derivatives associated with a curve point. Moreover, the class also provides a method by means of which one can generate Matlab source codes to plot the curve points and to create scalable vector graphic formats (like EPS). The declaration and implementation of the class can be found in Listings 2.37/157 and 2.38/159, respectively.

Listing 2.37. 3-dimensional generic curves (**Core/Geometry/Curves/GenericCurves3.h**)

```
1  #ifndef GENERICCURVES3_H
2  #define GENERICCURVES3_H

3  #include <GL/glew.h>
4  #include <iostream>

5  #include "../Coordinates/Cartesians3.h"
6  #include "../Coordinates/Colors4.h"
7  #include "../../Math/Matrices.h"
8  #include "../../Shaders/ShaderPrograms.h"

9  namespace cagd
10 {
11     class GenericCurve3
12     {
13         // friend classes that will be discussed later in Listings 2.42/185/2.43/188 and 2.44/194/2.45/197, respectively
14         friend class LinearCombination3;
15         friend class TensorProductSurface3;

16         // overloaded friend input/output from/to stream operators
17         friend std::ostream& operator <<(std::ostream& lhs, const GenericCurve3& rhs);
18         friend std::istream& operator >>(std::istream& lhs, GenericCurve3& rhs);
```


© Ágoston Róth, 2018



Fig. 2.19: Class diagram of generic curves

```
19      protected:
20          GLenum                  _usage_flag;
21          RowMatrix<GLuint>       _vbo_derivative;
22          Matrix<Cartesian3>      _derivative;

23      public:
24          // default/special constructor
25          GenericCurve3(GLint maximum_order_of_derivatives = 2,
26                        GLint point_count = 0,
27                        GLenum usage_flag = GL_STATIC_DRAW);

28          // copy constructor
29          GenericCurve3(const GenericCurve3& curve);

30          // assignment operator
31          GenericCurve3& operator =(const GenericCurve3& rhs);

32          // Deletes the vertex buffer objects of zeroth and higher order derivatives.
33          GLvoid      deleteVertexBufferObjects();

34          // The next rendering method will return GL_FALSE if:
35          //  - the given shader program is not active;
36          //  - the user-defined position attribute name cannot be found in the list of active attributes
37          //    of the provided shader program, or exists but is of incorrect type;
38          //  - the specified order of derivatives is greater than or equal to the row count of the
39          //    Matrix<Cartesian3> _derivative;
40          //  - the render_mode is incorrect (in case of zeroth order derivatives one should use one of the constants
41          //    GL_LINE_STRIP, GL_LINE_LOOP and GL_POINTS, while in case of higher order derivatives
42          //    one has to use either GL_LINES or GL_POINTS).
43          GLboolean renderDerivatives(const ShaderProgram &program, GLint order,
44                                      GLenum render_mode) const;

45          // If during rendering one intends to use shader program objects that are not instances of
46          // our class ShaderProgram, then one should specify an attribute location associated with
47          // positions of type vec3.
```


© Ágoston Róth, 2018

```
48          //
49          // The next rendering method will return GL_FALSE if:
50          //  - there is no active shader program object;
51          //  - the given position location either cannot be found in the list of active attribute locations,
52          //    or exists but is of incorrect type;
53          //  - the specified order of derivatives is greater than or equal to the row count of the
54          //    Matrix<Cartesian3> _derivative;
55          //  - the render_mode is incorrect (in case of zeroth order derivatives one should use one of the constants
56          //    GL_LINE_STRIP, GL_LINE_LOOP and GL_POINTS, while in case of higher order derivatives
57          //    one has to use either GL_LINES or GL_POINTS).
58          GLboolean renderDerivatives(GLint order, GLenum render_mode,
59                                      GLint vec3_position_location = 0) const;
60          // Updates the vertex buffer objects of the zeroth and higher order derivatives.
61          GLboolean updateVertexBufferObjects(GLenum usage_flag = GL_STATIC_DRAW);
62          // One can either map or unmap a vertex buffer object associated with a specific order of derivatives.
63          GLfloat*  mapDerivatives(GLint order, GLenum access_mode = GL_READ_ONLY) const;
64          GLboolean unmapDerivatives(GLint order) const;
65          // get derivative by constant reference
66          const Cartesian3& operator ()(GLint order, GLint index) const;
67          // get derivative by non-constant reference
68          Cartesian3& operator ()(GLint order, GLint index);
69          // other updating and querying methods
70          GLboolean setDerivative(GLint order, GLint index,
71                                  GLdouble x, GLdouble y, GLdouble z = 0.0);
72          GLboolean setDerivative(GLint order, GLint index, const Cartesian3& d);
73          GLboolean derivative(GLint order, GLint index,
74                               GLdouble &x, GLdouble &y, GLdouble &z) const;
75          GLboolean derivative(GLint order, GLint index, Cartesian3 &d) const;
76          GLint      maximumOrderOfDerivatives() const;
77          GLint      pointCount() const;
78          GLenum     usageFlag() const;
79          // The next method can be used for generating Matlab code, by using which one can create scalable
80          // vector graphic file formats like EPS.
81          // Variable file_name specifies the name of the output file. Make sure that its extension is ".m".
82          // Variable access_mode can be either std::ios_base::out, or std::ios_base::out | std::ios_base::app, in case of
83          // which the given file will be either overwritten, or appended.
84          // Variable line_color specifies the red, green and blue components of the color 'Color' property of the
85          // Matlab-commands plot and plot3.
86          // Variable line_style can be used to specify formatum strings for Matlab-like line styles, which are used by
87          // the Matlab-commands plot and plot3. It accepts one of the formatum strings "'-'", "'-.'", "':'" and "'.'".
88          // Variable line_width specifies the positive line width which will be used by the Matlab-commands plot
89          // and plot3.
90          // Variables {x|y|z}_coordinate_name store the names of arrays that will contain the {x|y|z}-coordinates
91          // of the curve points. These array names will be passed as input arguments to the plotting commands
92          // of Matlab.
93          GLboolean generateMatlabCodeForRendering(
94              const std::string &file_name,
95              std::ios_base::openmode access_mode = std::ios_base::out | std::ios_base::app,
96              const Color4 &line_color = colors::blue,
97              const std::string &line_style = "'-'",
98              const GLdouble &line_width = 1.0,
99              const std::string &x_coordinate_name = "x",
100             const std::string &y_coordinate_name = "y",
101             const std::string &z_coordinate_name = "z") const;
102         // clone function required by smart pointers based on the deep copy ownership policy
103         virtual GenericCurve3* clone() const;
104         // destructor
105         virtual ~GenericCurve3();
106     };
107 }
108 #endif // GENERICCURVES3_H
```


© Ágoston Róth, 2018

Listing 2.38. 3-dimensional generic curves (**Core/Geometry/Curves/GenericCurves3.cpp**)

```
#include "GenericCurves3.h"
#include <fstream>

using namespace std;

namespace cagd
{
    // default/special constructor
    GenericCurve3::GenericCurve3(
        GLint maximum_order_of_derivatives, GLint point_count, GLenum usage_flag):
        _usage_flag(usage_flag),
        _vbo_derivative(maximum_order_of_derivatives < 0 ?
                        0 : maximum_order_of_derivatives + 1),
        _derivative(maximum_order_of_derivatives < 0 ?
                    0 : maximum_order_of_derivatives + 1,
                    point_count < 0 ? 0 : point_count)
    {
        assert("The maximum order of derivatives should be a non-negative integer!" &&
               maximum_order_of_derivatives >= 0);

        assert("The number of subdivision points should be a non-negative integer!" &&
               point_count >= 0);

        assert("Invalid usage flag!" &&
               (usage_flag == GL_STREAM_DRAW  || usage_flag == GL_STREAM_READ  ||
                usage_flag == GL_STREAM_COPY  ||
                usage_flag == GL_DYNAMIC_DRAW || usage_flag == GL_DYNAMIC_READ ||
                usage_flag == GL_DYNAMIC_COPY ||
                usage_flag == GL_STATIC_DRAW  || usage_flag == GL_STATIC_READ  ||
                usage_flag == GL_STATIC_COPY));
    }

    // copy constructor
    GenericCurve3::GenericCurve3(const GenericCurve3& curve):
            _usage_flag(curve._usage_flag),
            _vbo_derivative(curve._vbo_derivative.columnCount()),
            _derivative(curve._derivative)
    {
        GLboolean vbo_update_is_possible = GL_TRUE;
        for (GLint i = 0; i < curve._vbo_derivative.columnCount(); i++)
        {
            vbo_update_is_possible &= curve._vbo_derivative[i];
        }

        if (vbo_update_is_possible)
        {
            updateVertexBufferObjects(_usage_flag);
        }
    }

    // assignment operator
    GenericCurve3& GenericCurve3::operator =(const GenericCurve3& rhs)
    {
        if (this != &rhs)
        {
            deleteVertexBufferObjects();

            _usage_flag = rhs._usage_flag;
            _derivative = rhs._derivative;

            GLboolean vbo_update_is_possible = GL_TRUE;
            for (GLint i = 0; i < rhs._vbo_derivative.columnCount(); i++)
            {
                vbo_update_is_possible &= rhs._vbo_derivative[i];
            }

            if (vbo_update_is_possible)
            {
                updateVertexBufferObjects(_usage_flag);
            }
        }
```


© Ágoston Róth, 2018

```
62          return *this;
63      }
64      // Deletes the vertex buffer objects of zeroth and higher order derivatives.
65      GLvoid GenericCurve3::deleteVertexBufferObjects()
66      {
67          for (GLint i = 0; i < _vbo_derivative.columnCount(); i++)
68          {
69              if (_vbo_derivative[i])
70              {
71                  glDeleteBuffers(1, &_vbo_derivative[i]);
72                  _vbo_derivative[i] = 0;
73              }
74          }
75      }
76      // The next rendering method will return GL_FALSE if:
77      // - the given shader program is not active;
78      // - the user-defined position attribute name cannot be found in the list of active attributes
79      //   of the provided shader program, or exists but is of incorrect type;
80      // - the specified order of derivatives is greater than or equal to the row count of the
81      //   Matrix<Cartesian3> _derivative;
82      // - the render_mode is incorrect (in case of zeroth order derivatives one should use one of the constants
83      //   GL_LINE_STRIP, GL_LINE_LOOP and GL_POINTS, while in case of higher order derivatives
84      //   one has to use either GL_LINES or GL_POINTS).
85      GLboolean GenericCurve3::renderDerivatives(
86          const ShaderProgram &program, GLint order, GLenum render_mode) const
87      {
88          assert("The given differentiation order is out of bounds!" &&
89                 (order >= 0 && order < _derivative.rowCount()));
90          assert("The vertex buffer object corresponding to the given differentiation "
91                 "order does not exist!" && _vbo_derivative[order]);
92          if (order < 0 || order >= _derivative.rowCount() || !_vbo_derivative[order])
93          {
94              return GL_FALSE;
95          }
96          GLint current_program;
97          glGetIntegerv(GL_CURRENT_PROGRAM, &current_program);
98          if (current_program != (GLint)program._id)
99          {
100             return GL_FALSE;
101         }
102         GLint position = program.positionAttributeLocation();
103         if (position < 0)
104         {
105             return GL_FALSE;
106         }
107         GLsizei point_count = _derivative.columnCount();
108         glEnableVertexAttribArray(position);
109         glBindBuffer(GL_ARRAY_BUFFER, _vbo_derivative[order]);
110         glVertexAttribPointer(position, 3, GL_FLOAT, GL_FALSE, 0, nullptr);
111         if (!order)
112         {
113             if (render_mode != GL_LINE_STRIP &&
114                 render_mode != GL_LINE_LOOP  &&
115                 render_mode != GL_POINTS)
116             {
117                 glBindBuffer(GL_ARRAY_BUFFER, 0);
118                 glDisableVertexAttribArray(position);
119                 return GL_FALSE;
120             }
121             glDrawArrays(render_mode, 0, point_count);
122         }
123         else
```


© Ágoston Róth, 2018

```
124         {
125             if (render_mode != GL_LINES && render_mode != GL_POINTS)
126             {
127                 glBindBuffer(GL_ARRAY_BUFFER, 0);
128                 glDisableVertexAttribArray(position);
129                 return GL_FALSE;
130             }
131             glDrawArrays(render_mode, 0, 2 * point_count);
132         }
133         glBindBuffer(GL_ARRAY_BUFFER, 0);
134         glDisableVertexAttribArray(position);
135         glVertexAttrib3f(position, 0.0f, 0.0f, 0.0f);
136         return GL_TRUE;
137     }
138     // If during rendering one intends to use shader program objects that are not instances of
139     // our class ShaderProgram, then one should specify an attribute location associated with
140     // positions of type vec3.
141     //
142     // The next rendering method will return GL_FALSE if:
143     // - there is no active shader program object;
144     // - the given position location either cannot be found in the list of active attribute locations,
145     //   or exists but is of incorrect type;
146     // - the specified order of derivatives is greater than or equal to the row count of the
147     //   Matrix<Cartesian3> _derivative;
148     // - the render_mode is incorrect (in case of zeroth order derivatives one should use one of the constants
149     //   GL_LINE_STRIP, GL_LINE_LOOP and GL_POINTS, while in case of higher order derivatives
150     //   one has to use either GL_LINES or GL_POINTS).
151     GLboolean GenericCurve3::renderDerivatives(
152         GLint order, GLenum render_mode, GLint vec3_position_location) const
153     {
154         assert("The given differentiation order is out of bounds!" &&
155                (order >= 0 && order < _derivative.rowCount()));
156         assert("The vertex buffer object corresponding to the given differentiation "
157                "order does not exist!" && _vbo_derivative[order]);
158         assert("The given position attribute location should be a non-negative "
159                "integer!" && vec3_position_location >= 0);
160         if (order < 0 || order >= _derivative.rowCount() || !_vbo_derivative[order] ||
161             vec3_position_location < 0)
162         {
163             return GL_FALSE;
164         }
165         GLint current_program = 0;
166         glGetIntegerv(GL_CURRENT_PROGRAM, &current_program);
167         if (!current_program)
168         {
169             return GL_FALSE;
170         }
171         GLint attribute_count;
172         glGetProgramiv(current_program, GL_ACTIVE_ATTRIBUTES, &attribute_count);
173         if (!attribute_count)
174         {
175             return GL_FALSE;
176         }
177         GLsizei max_attribute_name_length;
178         glGetProgramiv(current_program, GL_ACTIVE_ATTRIBUTE_MAX_LENGTH,
179                        &max_attribute_name_length);
180         GLchar *attribute_name_data = new GLchar[max_attribute_name_length];
181         GLboolean given_location_exists_and_is_of_type_vec3 = GL_FALSE;
182         for (GLint attribute = 0;
```


© Ágoston Róth, 2018

```
183             attribute < attribute_count && !given_location_exists_and_is_of_type_vec3;
184             attribute++)
185         {
186             GLsizei actual_length = 0;
187             GLint   array_size    = 0;
188             GLenum  type          = 0;
189             glGetActiveAttrib(current_program, attribute, max_attribute_name_length,
190                               &actual_length, &array_size, &type, attribute_name_data);
191             string name(&attribute_name_data[0], actual_length);
192             GLint  location = glGetAttribLocation(current_program, name.c_str());
193             if (type == GL_FLOAT_VEC3 && vec3_position_location == location)
194             {
195                 given_location_exists_and_is_of_type_vec3 = GL_TRUE;
196             }
197         }
198         delete[] attribute_name_data;
199         if (!given_location_exists_and_is_of_type_vec3)
200         {
201             return GL_FALSE;
202         }
203         GLint point_count = _derivative.columnCount();
204         glEnableVertexAttribArray(vec3_position_location);
205         glBindBuffer(GL_ARRAY_BUFFER, _vbo_derivative[order]);
206         glVertexAttribPointer(vec3_position_location, 3, GL_FLOAT, GL_FALSE, 0, nullptr);
207         if (!order)
208         {
209             if (render_mode != GL_LINE_STRIP &&
210                 render_mode != GL_LINE_LOOP  &&
211                 render_mode != GL_POINTS)
212             {
213                 glBindBuffer(GL_ARRAY_BUFFER, 0);
214                 glDisableVertexAttribArray(vec3_position_location);
215                 return GL_FALSE;
216             }
217             glDrawArrays(render_mode, 0, point_count);
218         }
219         else
220         {
221             if (render_mode != GL_LINES && render_mode != GL_POINTS)
222             {
223                 glBindBuffer(GL_ARRAY_BUFFER, 0);
224                 glDisableVertexAttribArray(vec3_position_location);
225                 return GL_FALSE;
226             }
227             glDrawArrays(render_mode, 0, 2 * point_count);
228         }
229         glBindBuffer(GL_ARRAY_BUFFER, 0);
230         glDisableVertexAttribArray(vec3_position_location);
231         glVertexAttrib3f(vec3_position_location, 0.0f, 0.0f, 0.0f);
232         return GL_TRUE;
233     }
234     // Updates the vertex buffer objects of the zeroth and higher order derivatives.
235     GLboolean GenericCurve3::updateVertexBufferObjects(GLenum usage_flag)
236     {
237         if (usage_flag != GL_STREAM_DRAW  && usage_flag != GL_STREAM_READ  &&
238             usage_flag != GL_STREAM_COPY  &&
239             usage_flag != GL_DYNAMIC_DRAW && usage_flag != GL_DYNAMIC_READ &&
240             usage_flag != GL_DYNAMIC_COPY &&
241             usage_flag != GL_STATIC_DRAW  && usage_flag != GL_STATIC_READ  &&
242             usage_flag != GL_STATIC_COPY)
243         {
244             return GL_FALSE;
```


© Ágoston Róth, 2018

```
245         }
246         deleteVertexBufferObjects();
247         _usage_flag = usage_flag;
248         for (GLint d = 0; d < _vbo_derivative.columnCount(); d++)
249         {
250             glGenBuffers(1, &_vbo_derivative[d]);
251             if (!_vbo_derivative[d])
252             {
253                 for (GLint i = 0; i < d; i++)
254                 {
255                     glDeleteBuffers(1, &_vbo_derivative[i]);
256                     _vbo_derivative[i] = 0;
257                 }
258                 return GL_FALSE;
259             }
260         }
261         GLint curve_point_count = _derivative.columnCount();
262         GLfloat *coordinate = nullptr;
263         // curve points
264         GLsizeiptr curve_point_byte_size = 3 * curve_point_count * sizeof(GLfloat);
265         glBindBuffer(GL_ARRAY_BUFFER, _vbo_derivative[0]);
266         glBufferData(GL_ARRAY_BUFFER, curve_point_byte_size, nullptr, _usage_flag);
267         coordinate = (GLfloat*)glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
268         if (!coordinate)
269         {
270             glBindBuffer(GL_ARRAY_BUFFER, 0);
271             deleteVertexBufferObjects();
272             return GL_FALSE;
273         }
274         for (GLint i = 0; i < curve_point_count; i++)
275         {
276             for (GLint j = 0; j < 3; j++, coordinate++)
277             {
278                 *coordinate = (GLfloat)_derivative(0, i)[j];
279             }
280         }
281         if (!glUnmapBuffer(GL_ARRAY_BUFFER))
282         {
283             glBindBuffer(GL_ARRAY_BUFFER, 0);
284             deleteVertexBufferObjects();
285             return GL_FALSE;
286         }
287         // higher order derivatives
288         GLsizeiptr higher_order_derivative_byte_size = 2 * curve_point_byte_size;
289         for (GLint d = 1; d < _derivative.rowCount(); d++)
290         {
291             glBindBuffer(GL_ARRAY_BUFFER, _vbo_derivative[d]);
292             glBufferData(GL_ARRAY_BUFFER, higher_order_derivative_byte_size, nullptr,
293                          _usage_flag);
294             coordinate = (GLfloat*)glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
295             if (!coordinate)
296             {
297                 glBindBuffer(GL_ARRAY_BUFFER, 0);
298                 deleteVertexBufferObjects();
299                 return GL_FALSE;
300             }
```


© Ágoston Róth, 2018

```
301             for (GLint i = 0; i < curve_point_count; i++, coordinate += 3)
302             {
303                 Cartesian3 sum = _derivative(0, i);
304                 sum += _derivative(d, i);
305 
306                 for (GLint j = 0; j < 3; j++, coordinate++)
307                 {
308                     *coordinate       = (GLfloat)_derivative(0, i)[j];
309                     *(coordinate + 3) = (GLfloat)sum[j];
310                 }
311             }
312 
313             if (!glUnmapBuffer(GL_ARRAY_BUFFER))
314             {
315                 glBindBuffer(GL_ARRAY_BUFFER, 0);
316                 deleteVertexBufferObjects();
317                 return GL_FALSE;
318             }
319         }
320 
321         glBindBuffer(GL_ARRAY_BUFFER, 0);
322 
323         return GL_TRUE;
324     }
325 
326     // One can either map or unmap a vertex buffer object associated with a specific order of derivatives.
327     GLfloat* GenericCurve3::mapDerivatives(GLint order, GLenum access_mode) const
328     {
329         assert("The given differentiation order is out of bounds!" &&
330                (order >= 0 && order < _vbo_derivative.columnCount()));
331 
332         assert("The vertex buffer object corresponding to the given differentiation "
333                "order does not exist!" && !_vbo_derivative[order]);
334 
335         if (order < 0 || order >= _vbo_derivative.columnCount() ||
336             !_vbo_derivative[order])
337         {
338             return nullptr;
339         }
340 
341         if (access_mode != GL_READ_ONLY && access_mode != GL_WRITE_ONLY &&
342             access_mode != GL_READ_WRITE)
343         {
344             return nullptr;
345         }
346 
347         glBindBuffer(GL_ARRAY_BUFFER, _vbo_derivative[order]);
348 
349         return (GLfloat*)glMapBuffer(GL_ARRAY_BUFFER, access_mode);
350     }
351 
352     GLboolean GenericCurve3::unmapDerivatives(GLint order) const
353     {
354         assert("The given differentiation order is out of bounds!" &&
355                (order >= 0 && order < _vbo_derivative.columnCount()));
356 
357         assert("The vertex buffer object corresponding to the given differentiation "
358                "order does not exist!" && !_vbo_derivative[order]);
359 
360         if (order < 0 || order >= _vbo_derivative.columnCount() ||
361             !_vbo_derivative[order])
362         {
363             return GL_FALSE;
364         }
365 
366         glBindBuffer(GL_ARRAY_BUFFER, _vbo_derivative[order]);
367 
368         return glUnmapBuffer(GL_ARRAY_BUFFER);
369     }
370 
371     // get derivative by constant reference
372     const Cartesian3& GenericCurve3::operator ()(GLint order, GLint index) const
373     {
374         assert("The given differentiation order is out of bounds!" &&
```


© Ágoston Róth, 2018

```
359             (order >= 0 && order < _derivative.rowCount()));

360         assert("The given curve point index is out of bounds!" &&
361                (index >= 0 && index < _derivative.columnCount()));

362         return _derivative(order, index);
363     }

364     // get derivative by non-constant reference
365     Cartesian3& GenericCurve3::operator ()(GLint order, GLint index)
366     {
367         assert("The given differentiation order is out of bounds!" &&
368                (order >= 0 && order < _derivative.rowCount()));

369         assert("The given curve point index is out of bounds!" &&
370                (index >= 0 && index < _derivative.columnCount()));

371         return _derivative(order, index);
372     }

373     // other updating and querying methods
374     GLboolean GenericCurve3::setDerivative(
375         GLint order, GLint index, GLdouble x, GLdouble y, GLdouble z)
376     {
377         assert("The given differentiation order is out of bounds!" &&
378                (order >= 0 && order < _derivative.rowCount()));

379         assert("The given curve point index is out of bounds!" &&
380                (index >= 0 && index < _derivative.columnCount()));

381         if (order < 0 || order >= _derivative.rowCount() ||
382             index < 0 || index >= _derivative.columnCount())
383         {
384             return GL_FALSE;
385         }

386         Cartesian3 &reference = _derivative(order, index);

387         reference[0] = x;
388         reference[1] = y;
389         reference[2] = z;

390         return GL_TRUE;
391     }

392     GLboolean GenericCurve3::setDerivative(GLint order, GLint index, const Cartesian3& d)
393     {
394         assert("The given differentiation order is out of bounds!" &&
395                (order >= 0 && order < _derivative.rowCount()));

396         assert("The given curve point index is out of bounds!" &&
397                (index >= 0 && index < _derivative.columnCount()));

398         if (order < 0 || order >= _derivative.rowCount() ||
399             index < 0 || index >= _derivative.columnCount())
400         {
401             return GL_FALSE;
402         }

403         _derivative(order, index) = d;

404         return GL_TRUE;
405     }

406     GLboolean GenericCurve3::derivative(
407         GLint order, GLint index, GLdouble& x, GLdouble& y, GLdouble& z) const
408     {
409         assert("The given differentiation order is out of bounds!" &&
410                (order >= 0 && order < _derivative.rowCount()));

411         assert("The given curve point index is out of bounds!" &&
412                (index >= 0 && index < _derivative.columnCount()));

413         if (order < 0 || order >= _derivative.rowCount() ||
```


© Ágoston Róth, 2018

```
414             index < 0 || index >= _derivative.columnCount())
415         {
416             return GL_FALSE;
417         }
418         const Cartesian3 &reference = _derivative(order, index);
419         x = reference[0];
420         y = reference[1];
421         z = reference[2];
422         return GL_TRUE;
423     }
424     GLboolean GenericCurve3::derivative(GLint order, GLint index, Cartesian3& d) const
425     {
426         assert("The given differentiation order is out of bounds!" &&
427                (order >= 0 && order < _derivative.rowCount()));
428         assert("The given curve point index is out of bounds!" &&
429                (index >= 0 && index < _derivative.columnCount()));
430         if (order < 0 || order >= _derivative.rowCount() ||
431             index < 0 || index >= _derivative.columnCount())
432         {
433             return GL_FALSE;
434         }
435         d = _derivative(order, index);
436         return GL_TRUE;
437     }
438     GLint GenericCurve3::maximumOrderOfDerivatives() const
439     {
440         return _derivative.rowCount() - 1;
441     }
442     GLint GenericCurve3::pointCount() const
443     {
444         return _derivative.columnCount();
445     }
446     GLenum GenericCurve3::usageFlag() const
447     {
448         return _usage_flag;
449     }
450     // The next method can be used for generating Matlab code, by using which one can create scalable
451     // vector graphic file formats like EPS.
452     // Variable file_name specifies the name of the output file. Make sure that its extension is ".m".
453     // Variable access_mode can be either std::ios_base::out, or std::ios_base::out | std::ios_base::app, in case of
454     // which the given file will be either overwritten, or appended.
455     // Variable line_color specifies the red, green and blue components of the color 'Color' property of the
456     // Matlab-commands plot and plot3.
457     // Variable line_style can be used to specify formatum strings for Matlab-like line styles, which are used by
458     // the Matlab-commands plot and plot3. It accepts one of the formatum strings "'-'", "'-.'", "':'" and "'.'".
459     // Variable line_width specifies the positive line width which will be used by the Matlab-commands plot
460     // and plot3.
461     // Variables {x|y|z}_coordinate_name store the names of arrays that will contain the {x|y|z}-coordinates
462     // of the curve points. These array names will be passed as input arguments to the plotting commands
463     // of Matlab.
464     GLboolean GenericCurve3::generateMatlabCodeForRendering(const string &file_name,
465         ios_base::openmode access_mode,
466         const Color4 &line_color,
467         const std::string &line_style,
468         const GLdouble &line_width,
469         const string &x_coordinate_name,
470         const string &y_coordinate_name,
471         const string &z_coordinate_name) const
472     {
473         if (file_name.empty() || x_coordinate_name.empty() ||
474             y_coordinate_name.empty() || z_coordinate_name.empty() ||
475             (x_coordinate_name == y_coordinate_name) ||
```


© Ágoston Róth, 2018

```
476             (x_coordinate_name == z_coordinate_name) ||
477             (y_coordinate_name == z_coordinate_name) ||
478             line_width <= 0.0 || _derivative.columnCount() == 0)
479         {
480             return GL_FALSE;
481         }
482         if (line_style != "'-'" && line_style != "'-.'" && line_style != "':'" &&
483             line_style != "'.'")
484         {
485             return GL_FALSE;
486         }
487         if ((access_mode != (fstream::out | fstream::app)) &&
488             (access_mode != fstream::out))
489         {
490             return GL_FALSE;
491         }
492         fstream f(file_name.c_str(), access_mode);
493         if (!f || !f.good())
494         {
495             f.close();
496             return GL_FALSE;
497         }
498         f << endl << "hold all;" << endl << endl;
499         f << x_coordinate_name << " = [..." << endl;
500         for (GLint i = 0; i < _derivative.columnCount(); i++)
501         {
502             f << _derivative(0, i)[0] << " ";
503         }
504         f << "..." << endl << "];" << endl;
505         f << endl;
506         f << y_coordinate_name << " = [..." << endl;
507         for (GLint i = 0; i < _derivative.columnCount(); i++)
508         {
509             f << _derivative(0, i)[1] << " ";
510         }
511         f << "..." << endl << "];" << endl;
512         f << endl;
513         f << z_coordinate_name << " = [..." << endl;
514         for (GLint i = 0; i < _derivative.columnCount(); i++)
515         {
516             f << _derivative(0, i)[2] << " ";
517         }
518         f << "..." << endl << "];" << endl;
519         f << endl << "plot3(" << x_coordinate_name << ", "
520                               << y_coordinate_name << ", "
521                               << z_coordinate_name << ", "
522                               << line_style << ", "
523                               << "'Color', ["
524                               << line_color[0] << ", "
525                               << line_color[1] << ", "
526                               << line_color[2] << "], "
527                               << "'LineWidth', " << line_width
528                               << ");" << endl;
529         f << endl << "axis equal;" << endl;
530         return GL_TRUE;
531     }
```


© Ágoston Róth, 2018

```
532     // clone function required by smart pointers based on the deep copy ownership policy
533     GenericCurve3* GenericCurve3::clone() const
534     {
535         return new (nothrow) GenericCurve3(*this);
536     }
537     // destructor
538     GenericCurve3::~GenericCurve3()
539     {
540         deleteVertexBufferObjects();
541     }
542     // overloaded friend input/output from/to stream operators
543     ostream& operator <<(ostream& lhs, const GenericCurve3& rhs)
544     {
545         return lhs << rhs._usage_flag << " " << rhs._derivative << endl;
546     }
547     std::istream& operator >>(std::istream& lhs, GenericCurve3& rhs)
548     {
549         rhs.deleteVertexBufferObjects();
550         return lhs >> rhs._usage_flag >> rhs._derivative;
551     }
552 }
```

2.13 Simple triangle meshes

We also provide classes for triangular faces (TriangularFace) and simple triangle meshes (TriangleMesh3), by means of which one can store in vertex buffer objects the attributes (i.e., position, normal and texture coordinates, color components and connectivity information) of vertices that form the triangular faces of the mesh.

The diagram of the class TriangularFace is shown in Fig. 2.20/169, while both of its definition and implementation can be found in Listing 2.39/169.

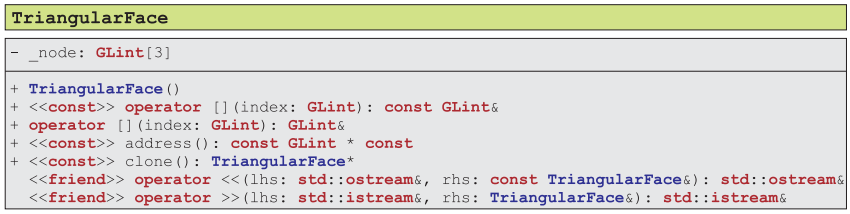


Fig. 2.20: Class diagram of triangular faces

Listing 2.39. Triangular faces (**Core/Geometry/Surfaces/TriangularFaces.h**)

```
1  #ifndef TRIANGULARFACES_H
2  #define TRIANGULARFACES_H
3  #include <GL/glew.h>
4  #include <cassert>
5  #include <iostream>
6  #include <new>
7  namespace cagd
8  {
9      class TriangularFace
10     {
11     private:
12         GLint _node[3];
```


© Ágoston Róth, 2018

```
13      public:
14          // default constructor
15          TriangularFace();
16          // get node identifier by constant reference
17          const GLint& operator [](GLint i) const;
18          // get node identifier by non-constant reference
19          GLint& operator [](GLint i);
20          // get constant pointer to constant data
21          const GLint* address() const;
22          // clone function required by smart pointers based on the deep copy ownership policy
23          TriangularFace* clone() const;
24      };
25      // default constructor
26      inline TriangularFace::TriangularFace()
27      {
28          _node[0] = _node[1] = _node[2] = -1;
29      }
30      // get node identifier by constant reference
31      inline const GLint& TriangularFace::operator [](GLint i) const
32      {
33          assert("The given node index is out of bounds!" && (i >= 0 && i < 3));
34          return _node[i];
35      }
36      // get node identifier by non-constant reference
37      inline GLint& TriangularFace::operator [](GLint i)
38      {
39          assert("The given node index is out of bounds!" && (i >= 0 && i < 3));
40          return _node[i];
41      }
42      // get constant pointer to constant data
43      inline const GLint* TriangularFace::address() const
44      {
45          return _node;
46      }
47      // clone function required by smart pointers based on the deep copy ownership policy
48      inline TriangularFace* TriangularFace::clone() const
49      {
50          return new (std::nothrow) TriangularFace(*this);
51      }
52      // overloaded output to stream operator
53      inline std::ostream& operator <<(std::ostream& lhs, const TriangularFace& rhs)
54      {
55          lhs << 3;
56          for (GLint i = 0; i < 3; i++)
57          {
58              lhs  << " " << rhs[i];
59          }
60          return lhs;
61      }
62      // overloaded input from stream operator
63      inline std::istream& operator >>(std::istream& lhs, TriangularFace& rhs)
64      {
65          GLint vertex_count;
66          return lhs >> vertex_count >> rhs[0] >> rhs[1] >> rhs[2];
67      }
68  }
69  #endif // TRIANGULARFACES_H
```

The diagram of the class TriangleMesh3 is illustrated in Fig. 2.21/171, while its definition and implementation can be found in Listings 2.40/170 and 2.41/173, respectively.

© Ágoston Róth, 2018

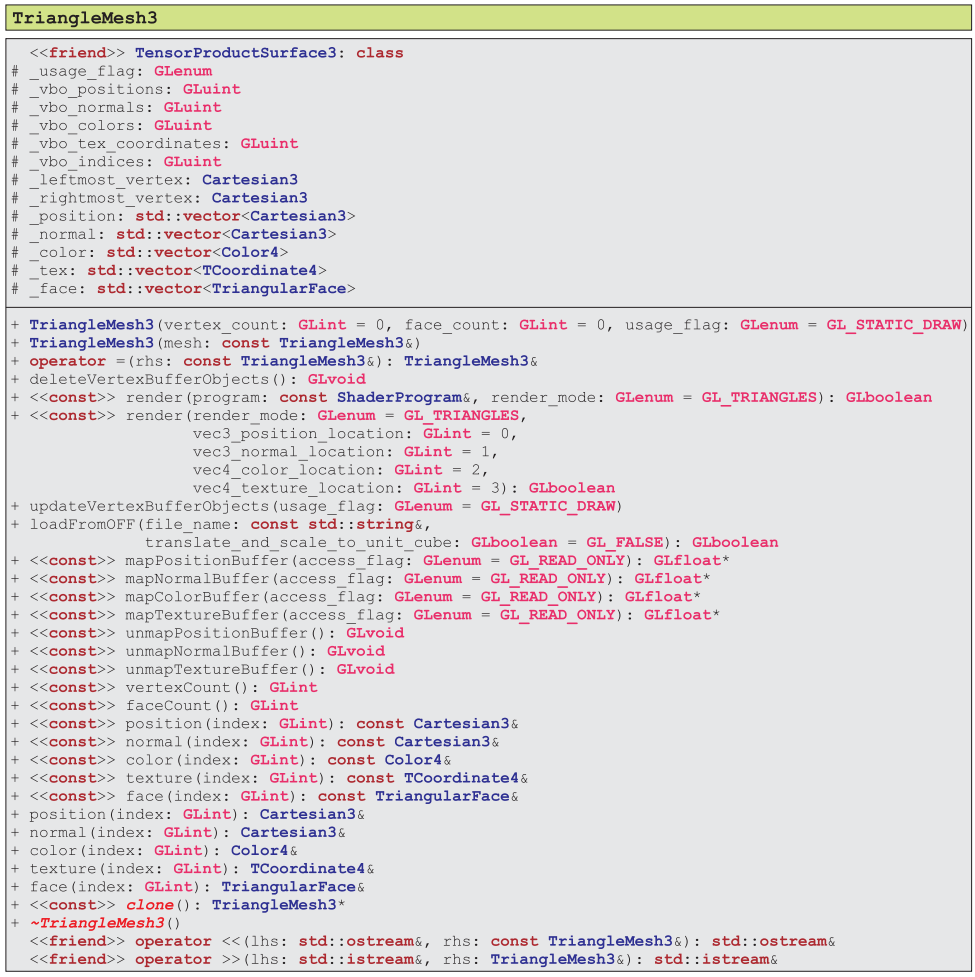


Fig. 2.21: Class diagram of simple triangle meshes

Listing 2.40. Simple triangle meshes (**Core/Geometry/Surfaces/TriangleMeshes3.h**)

```
1  #ifndef TRIANGLEMESHES3_H
2  #define TRIANGLEMESHES3_H

3  #include <GL/glew.h>

4  #include "../Coordinates/Cartesians3.h"
5  #include "../Coordinates/Colors4.h"
6  #include "../Coordinates/TCoordinates4.h"
7  #include "../../Shaders/ShaderPrograms.h"
8  #include "TriangularFaces.h"

9  #include <iostream>
10 #include <string>
11 #include <vector>

12 namespace cagd
13 {
14     class TriangleMesh3
15     {
16         // a friend class that will be discussed later in Listings 2.44/194 and 2.45/197
```


© Ágoston Róth, 2018

```
17          friend class TensorProductSurface3;

18          // overloaded friend input/output from/to stream operators
19          friend std::ostream& operator <<(std::ostream& lhs, const TriangleMesh3& rhs);
20          friend std::istream& operator >>(std::istream& lhs, TriangleMesh3& rhs);

21      protected:
22          // vertex buffer object identifiers and their common usage flag
23          GLenum                          _usage_flag;
24          GLuint                          _vbo_positions;
25          GLuint                          _vbo_normals;
26          GLuint                          _vbo_colors;
27          GLuint                          _vbo_tex_coordinates;
28          GLuint                          _vbo_indices;

29          // corners of the bounding box
30          Cartesian3                      _leftmost_position;
31          Cartesian3                      _rightmost_position;

32          // geometry
33          std::vector<Cartesian3>         _position;
34          std::vector<Cartesian3>         _normal;
35          std::vector<Color4>             _color;
36          std::vector<TCoordinate4>       _tex;
37          std::vector<TriangularFace>     _face;

38      public:
39          // default/special constructor
40          TriangleMesh3(GLint vertex_count = 0, GLint face_count = 0,
41                        GLenum usage_flag = GL_STATIC_DRAW);

42          // copy constructor
43          TriangleMesh3(const TriangleMesh3& mesh);

44          // assignment operator
45          TriangleMesh3& operator =(const TriangleMesh3& rhs);

46          // Deletes all vertex buffer objects.
47          GLvoid deleteVertexBufferObjects();

48          // Geometry rendering methods.

49          // The next rendering method will return GL_FALSE if:
50          //   - the given shader program is not active;
51          //   - the user-defined position, normal and texture attribute names cannot all be found in the
52          //     list of active attributes of the provided shader program, or they exist but all of
53          //     them have incorrect types;
54          //   - the render_mode is incorrect (one should use either GL_TRIANGLES or GL_POINTS).
55          GLboolean render(const ShaderProgram &program,
56                           GLenum render_mode = GL_TRIANGLES) const;

57          // If during rendering one intends to use shader program objects that are not instances of
58          // our class ShaderProgram, then one should specify attribute locations associated with
59          // positions of type vec3, unit normals of type vec3 and (possibly projective) texture
60          // coordinates of type vec4.
61          //
62          // The next rendering method will return GL_FALSE if:
63          //   - there is no active shader program object;
64          //   - the given position, normal and texture locations either cannot all be found in the list
65          //     of active attribute locations, or they exists but all of them have incorrect types;
66          //   - the render_mode is incorrect (one should use either GL_TRIANGLES or GL_POINTS).
67          GLboolean render(GLenum render_mode             = GL_TRIANGLES,
68                           GLint  vec3_position_location  = 0,
69                           GLint  vec3_normal_location    = 1,
70                           GLint  vec4_color_location     = 2,
71                           GLint  vec4_texture_location   = 3) const;

72          // Updates all vertex buffer objects.
73          GLboolean updateVertexBufferObjects(GLenum usage_flag = GL_STATIC_DRAW);

74          // Loads the geometry (i.e., the array of vertices and faces) stored in an OFF file,
75          // by calculating at the same time the unit normal vectors associated with vertices.
76          GLboolean loadFromOFF(const std::string &file_name,
77                                GLboolean translate_and_scale_to_unit_cube = GL_FALSE);
```


© Ágoston Róth, 2018

```
78          // for mapping vertex buffer objects
79          GLfloat* mapPositionBuffer(GLenum access_flag = GL_READ_ONLY) const;
80          GLfloat* mapNormalBuffer(GLenum access_flag = GL_READ_ONLY) const;
81          GLfloat* mapColorBuffer(GLenum access_flag = GL_READ_ONLY) const;
82          GLfloat* mapTextureBuffer(GLenum access_flag = GL_READ_ONLY) const;

83          // for unmapping vertex buffer objects
84          GLvoid unmapPositionBuffer() const;
85          GLvoid unmapNormalBuffer() const;
86          GLvoid unmapColorBuffer() const;
87          GLvoid unmapTextureBuffer() const;

88          // get properties of the geometry
89          GLint vertexCount() const;
90          GLint faceCount() const;

91          Cartesian3& position(GLint index);
92          const Cartesian3& position(GLint index) const;

93          Cartesian3& normal(GLint index);
94          const Cartesian3& normal(GLint index) const;

95          Color4& color(GLint index);
96          const Color4& color(GLint index) const;

97          TCoordinate4& texture(GLint index);
98          const TCoordinate4& texture(GLint index) const;

99          TriangularFace& face(GLint index);
100         const TriangularFace& face(GLint index) const;

101         // clone function required by smart pointers based on the deep copy ownership policy
102         virtual TriangleMesh3* clone() const;

103         // destructor
104         virtual ~TriangleMesh3();
105     };
106 }

107 #endif // TRIANGLEMESHES3_H
```

Listing 2.41. Simple triangle meshes (**Core/Geometry/Surfaces/TriangleMeshes3.cpp**)

```
1  #include "TriangleMeshes3.h"

2  #include <algorithm>
3  #include <cassert>
4  #include <cstring>
5  #include <fstream>
6  #include <limits>

7  using namespace std;

8  namespace cagd
9  {
10     // default/special constructor
11     TriangleMesh3::TriangleMesh3(GLint vertex_count, GLint face_count, GLenum usage_flag):
12         _usage_flag(usage_flag),
13         _vbo_positions(0),
14         _vbo_normals(0),
15         _vbo_colors(0),
16         _vbo_tex_coordinates(0),
17         _vbo_indices(0),
18         _position(vertex_count < 0 ? 0 : vertex_count),
19         _normal(vertex_count < 0 ? 0 : vertex_count),
20         _color(vertex_count < 0 ? 0 : vertex_count),
21         _tex(vertex_count < 0 ? 0 : vertex_count),
22         _face(face_count < 0 ? 0 : face_count)
23     {
24         assert("The number of vertices should be non-negative!" && vertex_count >= 0);
```


© Ágoston Róth, 2018

```
25          assert("The number of triangular faces should be non-negative!" &&
26                 face_count >= 0);

27          assert("Invalid usage flag!" &&
28                 (usage_flag == GL_STREAM_DRAW  || usage_flag == GL_STREAM_READ  ||
29                  usage_flag == GL_STREAM_COPY  ||
30                  usage_flag == GL_DYNAMIC_DRAW || usage_flag == GL_DYNAMIC_READ ||
31                  usage_flag == GL_DYNAMIC_COPY ||
32                  usage_flag == GL_STATIC_DRAW  || usage_flag == GL_STATIC_READ  ||
33                  usage_flag == GL_STATIC_COPY));
34      }

35      // copy constructor
36      TriangleMesh3::TriangleMesh3(const TriangleMesh3 &mesh):
37              _usage_flag(mesh._usage_flag),
38              _vbo_positions(0),
39              _vbo_normals(0),
40              _vbo_colors(0),
41              _vbo_tex_coordinates(0),
42              _vbo_indices(0),
43              _position(mesh._position),
44              _normal(mesh._normal),
45              _color(mesh._color),
46              _tex(mesh._tex),
47              _face(mesh._face)
48      {
49          if (mesh._vbo_positions &&
50              mesh._vbo_normals &&
51              mesh._vbo_colors &&
52              mesh._vbo_tex_coordinates &&
53              mesh._vbo_indices)
54          {
55              updateVertexBufferObjects(mesh._usage_flag);
56          }
57      }

58      // assignment operator
59      TriangleMesh3& TriangleMesh3::operator =(const TriangleMesh3 &rhs)
60      {
61          if (this != &rhs)
62          {
63              deleteVertexBufferObjects();

64              _usage_flag = rhs._usage_flag;
65              _position   = rhs._position;
66              _normal     = rhs._normal;
67              _color      = rhs._color;
68              _tex        = rhs._tex;
69              _face       = rhs._face;

70              if (rhs._vbo_positions &&
71                  rhs._vbo_normals &&
72                  rhs._vbo_colors &&
73                  rhs._vbo_tex_coordinates &&
74                  rhs._vbo_indices)
75              {
76                  updateVertexBufferObjects(_usage_flag);
77              }
78          }

79          return *this;
80      }

81      // Deletes all vertex buffer objects.
82      GLvoid TriangleMesh3::deleteVertexBufferObjects()
83      {
84          if (_vbo_positions)
85          {
86              glDeleteBuffers(1, &_vbo_positions);
87              _vbo_positions = 0;
88          }

89          if (_vbo_normals)
90          {
```


© Ágoston Róth, 2018

```
91              glDeleteBuffers(1, &_vbo_normals);
92              _vbo_normals = 0;
93          }
94          if (_vbo_colors)
95          {
96              glDeleteBuffers(1, &_vbo_colors);
97              _vbo_colors = 0;
98          }
99          if (_vbo_tex_coordinates)
100         {
101             glDeleteBuffers(1, &_vbo_tex_coordinates);
102             _vbo_tex_coordinates = 0;
103         }
104         if (_vbo_indices)
105         {
106             glDeleteBuffers(1, &_vbo_indices);
107             _vbo_indices = 0;
108         }
109     }
110     // Geometry rendering methods.
111     // The next rendering method will return GL_FALSE if:
112     //  - the given shader program is not active;
113     //  - the user-defined position, normal and texture attribute names cannot all be found in the
114     //    list of active attributes of the provided shader program, or they exist but all of
115     //    them have incorrect types;
116     //  - the render_mode is incorrect (one should use either GL_TRIANGLES or GL_POINTS).
117     GLboolean TriangleMesh3::render(const ShaderProgram &program,
118                                     GLenum render_mode) const
119     {
120         GLint current_program;
121         glGetIntegerv(GL_CURRENT_PROGRAM, &current_program);
122         if (current_program != (GLint)program._id)
123         {
124             return GL_FALSE;
125         }
126         GLint position = program.positionAttributeLocation();
127         GLint normal   = program.normalAttributeLocation();
128         GLint color    = program.colorAttributeLocation();
129         GLint texture  = program.textureAttributeLocation();
130         if (!_vbo_positions || !_vbo_normals || !_vbo_colors || !_vbo_tex_coordinates ||
131             !_vbo_indices || (position < 0 && normal < 0 && color < 0 && texture < 0))
132         {
133             return GL_FALSE;
134         }
135         if (render_mode != GL_TRIANGLES && render_mode != GL_POINTS)
136         {
137             return GL_FALSE;
138         }
139         // enable array of texture, color, normal and position attributes
140         if (texture >= 0)
141         {
142             glEnableVertexAttribArray(texture);
143             // activate the VBO of texture coordinates
144             glBindBuffer(GL_ARRAY_BUFFER, _vbo_tex_coordinates);
145             // specify the location and data format of texture coordinates
146             glVertexAttribPointer(texture, 4, GL_FLOAT, GL_FALSE, 0, nullptr);
147         }
148         if (color >= 0)
149         {
150             glEnableVertexAttribArray(color);
151             // activate the VBO of colors
152             glBindBuffer(GL_ARRAY_BUFFER, _vbo_colors);
153             // specify the location and data format of colors
```


© Ágoston Róth, 2018

```
154             glVertexAttribPointer(color, 4, GL_FLOAT, GL_FALSE, 0, nullptr);
155         }
156         if (normal >= 0)
157         {
158             glEnableVertexAttribArray(normal);
159             // activate the VBO of normal vectors
160             glBindBuffer(GL_ARRAY_BUFFER, _vbo_normals);
161             // specify the location and data format of normal vectors
162             glVertexAttribPointer(normal, 3, GL_FLOAT, GL_FALSE, 0, nullptr);
163         }
164         if (position >= 0)
165         {
166             glEnableVertexAttribArray(position);
167             // activate the VBO of vertices
168             glBindBuffer(GL_ARRAY_BUFFER, _vbo_positions);
169             // specify the location and data format of vertices
170             glVertexAttribPointer(position, 3, GL_FLOAT, GL_FALSE, 0, nullptr);
171         }
172         // activate the element array buffer for indexed vertices of triangular faces
173         glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _vbo_indices);
174         // render primitives
175         glDrawElements(render_mode, 3 * (GLsizei)_face.size(), GL_UNSIGNED_INT, nullptr);
176         // disable individual attribute arrays
177         if (position >= 0)
178         {
179             glDisableVertexAttribArray(position);
180             glVertexAttrib3f(position, 0.0f, 0.0f, 0.0f);
181         }
182         if (normal >= 0)
183         {
184             glDisableVertexAttribArray(normal);
185             glVertexAttrib3f(normal, 0.0f, 0.0f, 0.0f);
186         }
187         if (color >= 0)
188         {
189             glDisableVertexAttribArray(color);
190             glVertexAttrib4f(color, 0.0f, 0.0f, 0.0f, 1.0f);
191         }
192         if (texture >= 0)
193         {
194             glDisableVertexAttribArray(texture);
195             glVertexAttrib4f(texture, 0.0f, 0.0f, 0.0f, 1.0f);
196         }
197         // unbind any buffer object previously bound and restore client memory usage
198         // for these buffer object targets
199         glBindBuffer(GL_ARRAY_BUFFER, 0);
200         glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
201         return GL_TRUE;
202     }
203     // If during rendering one intends to use shader program objects that are not instances of
204     // our class ShaderProgram, then one should specify attribute locations associated with
205     // positions of type vec3, unit normals of type vec3 and (possibly projective) texture
206     // coordinates of type vec4.
207     //
208     // The next rendering method will return GL_FALSE if:
209     //   - there is no active shader program object;
210     //   - the given position, normal and texture locations either cannot all be found in the list
211     //     of active attribute locations, or they exists but all of them have incorrect types;
212     //   - the render_mode is incorrect (one should use either GL_TRIANGLES or GL_POINTS).
213     GLboolean TriangleMesh3::render(
214             GLenum render_mode,
215             GLint vec3_position_location,
216             GLint vec3_normal_location,
```


© Ágoston Róth, 2018

```
217             GLint vec4_color_location,
218             GLint vec4_texture_location) const
219     {
220         if (!_vbo_positions || !_vbo_normals || !_vbo_colors || !_vbo_tex_coordinates ||
221             !_vbo_indices ||
222             (vec3_position_location < 0 && vec3_normal_location < 0 &&
223              vec4_color_location < 0 && vec4_texture_location < 0))
224         {
225             return GL_FALSE;
226         }
227         if (render_mode != GL_TRIANGLES && render_mode != GL_POINTS)
228         {
229             return GL_FALSE;
230         }
231         GLint current_program = 0;
232         glGetIntegerv(GL_CURRENT_PROGRAM, &current_program);
233         if (!current_program)
234         {
235             return GL_FALSE;
236         }
237         GLint attribute_count;
238         glGetProgramiv(current_program, GL_ACTIVE_ATTRIBUTES, &attribute_count);
239         if (!attribute_count)
240         {
241             return GL_FALSE;
242         }
243         GLsizei max_attribute_name_length;
244         glGetProgramiv(current_program, GL_ACTIVE_ATTRIBUTE_MAX_LENGTH,
245                        &max_attribute_name_length);
246         GLchar *attribute_name_data = new GLchar[max_attribute_name_length];
247         GLint      given_attribute_count = 4;
248         GLint      given_locations[] = {vec3_position_location, vec3_normal_location,
249                                         vec4_color_location, vec4_texture_location};
250         GLenum     expected_types[]  = {GL_FLOAT_VEC3, GL_FLOAT_VEC3,
251                                         GL_FLOAT_VEC4, GL_FLOAT_VEC4};
252         GLboolean given_locations_exist_and_have_proper_types[] = {GL_FALSE, GL_FALSE,
253                                                                    GL_FALSE, GL_FALSE};
254         for (GLint attribute = 0;
255              attribute < attribute_count &&
256              (!given_locations_exist_and_have_proper_types[0] ||
257               !given_locations_exist_and_have_proper_types[1] ||
258               !given_locations_exist_and_have_proper_types[2] ||
259               !given_locations_exist_and_have_proper_types[3]);
260              attribute++)
261         {
262             GLsizei actual_length = 0;
263             GLint   array_size    = 0;
264             GLenum  type          = 0;
265             glGetActiveAttrib(current_program, attribute, max_attribute_name_length,
266                               &actual_length, &array_size, &type, attribute_name_data);
267             string name(&attribute_name_data[0], actual_length);
268             GLint  location = glGetAttribLocation(current_program, name.c_str());
269             for (GLint i = 0; i < given_attribute_count; i++)
270             {
271                 if (type == expected_types[i] && given_locations[i] == location)
272                 {
273                     given_locations_exist_and_have_proper_types[i] = GL_TRUE;
274                 }
275             }
276         }
```


© Ágoston Róth, 2018

```
277         delete[] attribute_name_data;
278         if (!given_locations_exist_and_have_proper_types[0] &&
279             !given_locations_exist_and_have_proper_types[1] &&
280             !given_locations_exist_and_have_proper_types[2] &&
281             !given_locations_exist_and_have_proper_types[3])
282         {
283             return GL_FALSE;
284         }
285         // enable array of position, normal, color and texture attributes
286         for (GLint i = 0; i < given_attribute_count; i++)
287         {
288             if (given_locations_exist_and_have_proper_types[i] &&
289                 (given_locations[i] >= 0))
290             {
291                 glEnableVertexAttribArray(given_locations[i]);
292             }
293         }
294         GLuint vbos[]              = {_vbo_positions, _vbo_normals,
295                                       _vbo_colors, _vbo_tex_coordinates};
296         GLint  component_counts[] = {3, 3, 4, 4};
297         for (GLint i = 0; i < given_attribute_count; i++)
298         {
299             if (given_locations_exist_and_have_proper_types[i] &&
300                 (given_locations[i] >= 0))
301             {
302                 // activate the ith VBO
303                 glBindBuffer(GL_ARRAY_BUFFER, vbos[i]);
304                 // and specify its location and data format
305                 glVertexAttribPointer(given_locations[i], component_counts[i],
306                                       GL_FLOAT, GL_FALSE, 0, nullptr);
307             }
308         }
309         // activate the element array buffer for indexed vertices of triangular faces
310         glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _vbo_indices);
311         // render primitives
312         glDrawElements(render_mode, 3 * (GLsizei)_face.size(), GL_UNSIGNED_INT, nullptr);
313         // unbind any buffer object previously bound and restore client memory usage
314         // for these buffer object targets
315         glBindBuffer(GL_ARRAY_BUFFER, 0);
316         glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
317         // disable individual attribute arrays
318         for (GLint i = 0; i < given_attribute_count; i++)
319         {
320             if (given_locations_exist_and_have_proper_types[i] &&
321                 (given_locations[i] >= 0))
322             {
323                 glDisableVertexAttribArray(given_locations[i]);
324                 switch (expected_types[i])
325                 {
326                 case GL_FLOAT_VEC3:
327                     glVertexAttrib3f(given_locations[i], 0.0f, 0.0f, 0.0f);
328                     break;
329                 default:
330                     glVertexAttrib4f(given_locations[i], 0.0f, 0.0f, 0.0f, 1.0f);
331                     break;
332                 }
333             }
334         }
335         return GL_TRUE;
336     }
```


© Ágoston Róth, 2018

```
337     // Updates all vertex buffer objects.
338     GLboolean TriangleMesh3::updateVertexBufferObjects(GLenum usage_flag)
339     {
340         if (usage_flag != GL_STREAM_DRAW  && usage_flag != GL_STREAM_READ  &&
341             usage_flag != GL_STREAM_COPY  &&
342             usage_flag != GL_STATIC_DRAW  && usage_flag != GL_STATIC_READ  &&
343             usage_flag != GL_STATIC_COPY  &&
344             usage_flag != GL_DYNAMIC_DRAW && usage_flag != GL_DYNAMIC_READ &&
345             usage_flag != GL_DYNAMIC_COPY)
346         {
347             return GL_FALSE;
348         }
349         // updating the usage flag
350         _usage_flag = usage_flag;
351         // deleting old vertex buffer objects
352         deleteVertexBufferObjects();
353         // creating vertex buffer objects of mesh vertices, unit normal vectors, colors, texture coordinates
354         // and element indices
355         glGenBuffers(1, &_vbo_positions);
356         if (!_vbo_positions)
357         {
358             return GL_FALSE;
359         }
360         glGenBuffers(1, &_vbo_normals);
361         if (!_vbo_normals)
362         {
363             glDeleteBuffers(1, &_vbo_positions);
364             _vbo_positions = 0;
365             return GL_FALSE;
366         }
367         glGenBuffers(1, &_vbo_colors);
368         if (!_vbo_colors)
369         {
370             glDeleteBuffers(1, &_vbo_positions);
371             _vbo_positions = 0;
372             glDeleteBuffers(1, &_vbo_normals);
373             _vbo_normals = 0;
374             return GL_FALSE;
375         }
376         glGenBuffers(1, &_vbo_tex_coordinates);
377         if (!_vbo_tex_coordinates)
378         {
379             glDeleteBuffers(1, &_vbo_positions);
380             _vbo_positions = 0;
381             glDeleteBuffers(1, &_vbo_normals);
382             _vbo_normals = 0;
383             glDeleteBuffers(1, &_vbo_colors);
384             _vbo_colors = 0;
385             return GL_FALSE;
386         }
387         glGenBuffers(1, &_vbo_indices);
388         if (!_vbo_indices)
389         {
390             glDeleteBuffers(1, &_vbo_positions);
391             _vbo_positions = 0;
392             glDeleteBuffers(1, &_vbo_normals);
393             _vbo_normals = 0;
```


© Ágoston Róth, 2018

```
394             glDeleteBuffers(1, &_vbo_colors);
395             _vbo_colors = 0;
396             glDeleteBuffers(1, &_vbo_tex_coordinates);
397             _vbo_tex_coordinates = 0;
398             return GL_FALSE;
399         }
400         // For efficiency reasons we convert all GLdouble coordinates to GLfloat ones: we will use
401         // auxiliar pointers for buffer data loading and functions glMapBuffer/glUnmapBuffer.
402         // Note that, multiple buffers can be mapped simultaneously.
403         GLsizeiptr vertex_byte_size = 3 * _position.size() * sizeof(GLfloat);
404         glBindBuffer(GL_ARRAY_BUFFER, _vbo_positions);
405         glBufferData(GL_ARRAY_BUFFER, vertex_byte_size, nullptr, _usage_flag);
406         GLfloat *vertex_coordinate = (GLfloat*)glMapBuffer(GL_ARRAY_BUFFER,
407                                                            GL_WRITE_ONLY);
408         glBindBuffer(GL_ARRAY_BUFFER, _vbo_normals);
409         glBufferData(GL_ARRAY_BUFFER, vertex_byte_size, nullptr, _usage_flag);
410         GLfloat *normal_coordinate = (GLfloat*)glMapBuffer(GL_ARRAY_BUFFER,
411                                                            GL_WRITE_ONLY);
412         for (vector<Cartesian3>::const_iterator
413              vit = _position.begin(),
414              nit = _normal.begin(); vit != _position.end(); vit++, nit++)
415         {
416             for (GLint component = 0; component < 3; component++,
417                  vertex_coordinate++, normal_coordinate++)
418             {
419                 *vertex_coordinate = (GLfloat)(*vit)[component];
420                 *normal_coordinate = (GLfloat)(*nit)[component];
421             }
422         }
423         GLsizeiptr color_or_tex_byte_size = 4 * _color.size() * sizeof(GLfloat);
424         glBindBuffer(GL_ARRAY_BUFFER, _vbo_colors);
425         glBufferData(GL_ARRAY_BUFFER, color_or_tex_byte_size, nullptr, _usage_flag);
426         GLfloat *color_coordinate = (GLfloat*)glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
427         memcpy(color_coordinate, &_color[0][0], color_or_tex_byte_size);
428         glBindBuffer(GL_ARRAY_BUFFER, _vbo_tex_coordinates);
429         glBufferData(GL_ARRAY_BUFFER, color_or_tex_byte_size, nullptr, _usage_flag);
430         GLfloat *tex_coordinate = (GLfloat*)glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
431         memcpy(tex_coordinate, &_tex[0][0], color_or_tex_byte_size);
432         GLsizeiptr index_byte_size = 3 * (GLsizeiptr)(_face.size() * sizeof(GLint));
433         glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _vbo_indices);
434         glBufferData(GL_ELEMENT_ARRAY_BUFFER, index_byte_size, nullptr, _usage_flag);
435         GLuint *element = (GLuint*)glMapBuffer(GL_ELEMENT_ARRAY_BUFFER, GL_WRITE_ONLY);
436         for (vector<TriangularFace>::const_iterator fit = _face.begin();
437              fit != _face.end(); fit++)
438         {
439             for (GLint node = 0; node < 3; node++, element++)
440             {
441                 *element = (*fit)[node];
442             }
443         }
444         // unmap all VBOs
445         glBindBuffer(GL_ARRAY_BUFFER, _vbo_positions);
446         if (!glUnmapBuffer(GL_ARRAY_BUFFER))
447             return GL_FALSE;
```


© Ágoston Róth, 2018

```
448         glBindBuffer(GL_ARRAY_BUFFER, _vbo_normals);
449         if (!glUnmapBuffer(GL_ARRAY_BUFFER))
450             return GL_FALSE;
451         glBindBuffer(GL_ARRAY_BUFFER, _vbo_colors);
452         if (!glUnmapBuffer(GL_ARRAY_BUFFER))
453             return GL_FALSE;
454         glBindBuffer(GL_ARRAY_BUFFER, _vbo_tex_coordinates);
455         if (!glUnmapBuffer(GL_ARRAY_BUFFER))
456             return GL_FALSE;
457         glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _vbo_indices);
458         if (!glUnmapBuffer(GL_ELEMENT_ARRAY_BUFFER))
459             return GL_FALSE;
460         // unbind any buffer object previously bound and restore client memory usage for these buffer object
461         // targets
462         glBindBuffer(GL_ARRAY_BUFFER, 0);
463         glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
464         return GL_TRUE;
465     }
466     // loads the geometry (i.e., the array of vertices and faces) stored in an OFF file
467     // at the same time calculates the unit normal vectors associated with vertices
468     GLboolean TriangleMesh3::loadFromOFF(
469             const string &file_name, GLboolean translate_and_scale_to_unit_cube)
470     {
471         fstream f(file_name.c_str(), ios_base::in);
472         if (!f || !f.good())
473         {
474             return GL_FALSE;
475         }
476         // loading the header
477         string header;
478         f >> header;
479         if (header != "OFF")
480         {
481             return GL_FALSE;
482         }
483         // loading number of vertices, faces, and edges
484         GLint vertex_count, face_count, edge_count;
485         f >> vertex_count >> face_count >> edge_count;
486         // allocating memory for vertices, unit normal vectors, colors, texture coordinates and faces
487         _position.resize(vertex_count);
488         _normal.resize(vertex_count);
489         _color.resize(vertex_count);
490         _tex.resize(vertex_count);
491         _face.resize(face_count);
492         // initializing the leftmost and rightmost corners of the bounding box
493         _leftmost_position.x() = _leftmost_position.y() = _leftmost_position.z() =
494                 numeric_limits<GLdouble>::max();
495         _rightmost_position.x() = _rightmost_position.y() = _rightmost_position.z() =
496                 -numeric_limits<GLdouble>::max();
497         // loading vertices and correcting the leftmost and rightmost corners of the bounding box
498         for (vector<Cartesian3>::iterator vit = _position.begin();
499              vit != _position.end(); vit++)
500         {
501             f >> *vit;
502             if (vit->x() < _leftmost_position.x()) {_leftmost_position.x() = vit->x();}
503             if (vit->y() < _leftmost_position.y()) {_leftmost_position.y() = vit->y();}
504             if (vit->z() < _leftmost_position.z()) {_leftmost_position.z() = vit->z();}
```


© Ágoston Róth, 2018

```
505             if (vit->x() > _rightmost_position.x()) {_rightmost_position.x() = vit->x();}
506             if (vit->y() > _rightmost_position.y()) {_rightmost_position.y() = vit->y();}
507             if (vit->z() > _rightmost_position.z()) {_rightmost_position.z() = vit->z();}
508         }
509         // if we do not want to preserve the original positions and coordinates of vertices
510         if (translate_and_scale_to_unit_cube)
511         {
512             Cartesian3 diagonal(_rightmost_position);
513             diagonal -= _leftmost_position;
514             GLdouble scale = 1.0 / max(diagonal[0], max(diagonal[1], diagonal[2]));
515             Cartesian3 middle(_leftmost_position);
516             middle += _rightmost_position;
517             middle *= 0.5;
518             for (vector<Cartesian3>::iterator vit = _position.begin();
519                  vit != _position.end(); vit++)
520             {
521                 *vit -= middle;
522                 *vit *= scale;
523             }
524         }
525         // loading faces
526         for (vector<TriangularFace>::iterator fit = _face.begin();
527              fit != _face.end(); fit++)
528         {
529             f >> *fit;
530         }
531         // calculating average unit normal vectors associated with vertices
532         for (vector<TriangularFace>::const_iterator fit = _face.begin();
533              fit != _face.end(); fit++)
534         {
535             Cartesian3 n = _position[(*fit)[1]];
536             n -= _position[(*fit)[0]];
537             Cartesian3 p = _position[(*fit)[2]];
538             p -= _position[(*fit)[0]];
539             n ^= p;
540             for (GLint node = 0; node < 3; node++)
541             {
542                 _normal[(*fit)[node]] += n;
543             }
544         }
545         for (vector<Cartesian3>::iterator nit = _normal.begin();
546              nit != _normal.end(); nit++)
547         {
548             nit->normalize();
549         }
550         f.close();
551         return GL_TRUE;
552     }
553     // for mapping vertex buffer objects
554     GLfloat* TriangleMesh3::mapPositionBuffer(GLenum access_flag) const
555     {
556         if (access_flag != GL_READ_ONLY && access_flag != GL_WRITE_ONLY &&
557             access_flag != GL_READ_WRITE)
558         {
559             return nullptr;
560         }
561         glBindBuffer(GL_ARRAY_BUFFER, _vbo_positions);
562         GLfloat* result = (GLfloat*)glMapBuffer(GL_ARRAY_BUFFER, access_flag);
563         glBindBuffer(GL_ARRAY_BUFFER, 0);
564         return result;
```


© Ágoston Róth, 2018

```
565     }
566     GLfloat* TriangleMesh3::mapNormalBuffer(GLenum access_flag) const
567     {
568         if (access_flag != GL_READ_ONLY && access_flag != GL_WRITE_ONLY &&
569             access_flag != GL_READ_WRITE)
570         {
571             return nullptr;
572         }
573         glBindBuffer(GL_ARRAY_BUFFER, _vbo_normals);
574         GLfloat* result = (GLfloat*)glMapBuffer(GL_ARRAY_BUFFER, access_flag);
575         glBindBuffer(GL_ARRAY_BUFFER, 0);
576         return result;
577     }
578     GLfloat* TriangleMesh3::mapColorBuffer(GLenum access_flag) const
579     {
580         if (access_flag != GL_READ_ONLY && access_flag != GL_WRITE_ONLY &&
581             access_flag != GL_READ_WRITE)
582         {
583             return nullptr;
584         }
585         glBindBuffer(GL_ARRAY_BUFFER, _vbo_colors);
586         GLfloat* result = (GLfloat*)glMapBuffer(GL_ARRAY_BUFFER, access_flag);
587         glBindBuffer(GL_ARRAY_BUFFER, 0);
588         return result;
589     }
590     GLfloat* TriangleMesh3::mapTextureBuffer(GLenum access_flag) const
591     {
592         if (access_flag != GL_READ_ONLY && access_flag != GL_WRITE_ONLY &&
593             access_flag != GL_READ_WRITE)
594         {
595             return nullptr;
596         }
597         glBindBuffer(GL_ARRAY_BUFFER, _vbo_tex_coordinates);
598         GLfloat* result = (GLfloat*)glMapBuffer(GL_ARRAY_BUFFER, access_flag);
599         glBindBuffer(GL_ARRAY_BUFFER, 0);
600         return result;
601     }
602     // for unmapping vertex buffer objects
603     GLvoid TriangleMesh3::unmapPositionBuffer() const
604     {
605         glBindBuffer(GL_ARRAY_BUFFER, _vbo_positions);
606         glUnmapBuffer(GL_ARRAY_BUFFER);
607         glBindBuffer(GL_ARRAY_BUFFER, 0);
608     }
609     GLvoid TriangleMesh3::unmapNormalBuffer() const
610     {
611         glBindBuffer(GL_ARRAY_BUFFER, _vbo_normals);
612         glUnmapBuffer(GL_ARRAY_BUFFER);
613         glBindBuffer(GL_ARRAY_BUFFER, 0);
614     }
615     GLvoid TriangleMesh3::unmapColorBuffer() const
616     {
617         glBindBuffer(GL_ARRAY_BUFFER, _vbo_normals);
618         glUnmapBuffer(GL_ARRAY_BUFFER);
619         glBindBuffer(GL_ARRAY_BUFFER, 0);
620     }
621     GLvoid TriangleMesh3::unmapTextureBuffer() const
622     {
623         glBindBuffer(GL_ARRAY_BUFFER, _vbo_tex_coordinates);
624         glUnmapBuffer(GL_ARRAY_BUFFER);
625         glBindBuffer(GL_ARRAY_BUFFER, 0);
```


© Ágoston Róth, 2018

```
626     }
627
628     // get properties of the geometry
629     GLint TriangleMesh3::vertexCount() const
630     {
631         return (GLint)_position.size();
632     }
633
634     GLint TriangleMesh3::faceCount() const
635     {
636         return (GLint)_face.size();
637     }
638
639     Cartesian3& TriangleMesh3::position(GLint index)
640     {
641         assert("The given position index is out of bounds!" &&
642                (index >= 0 && index < (GLint)_position.size()));
643         return _position[index];
644     }
645
646     const Cartesian3& TriangleMesh3::position(GLint index) const
647     {
648         assert("The given position index is out of bounds!" &&
649                (index >= 0 && index < (GLint)_position.size()));
650         return _position[index];
651     }
652
653     Cartesian3& TriangleMesh3::normal(GLint index)
654     {
655         assert("The given normal index is out of bounds!" &&
656                (index >= 0 && index < (GLint)_normal.size()));
657         return _normal[index];
658     }
659
660     const Cartesian3& TriangleMesh3::normal(GLint index) const
661     {
662         assert("The given normal index is out of bounds!" &&
663                (index >= 0 && index < (GLint)_normal.size()));
664         return _normal[index];
665     }
666
667     Color4& TriangleMesh3::color(GLint index)
668     {
669         assert("The given color index is out of bounds!" &&
670                (index >= 0 && index < (GLint)_color.size()));
671         return _color[index];
672     }
673
674     const Color4& TriangleMesh3::color(GLint index) const
675     {
676         assert("The given color index is out of bounds!" &&
677                (index >= 0 && index < (GLint)_color.size()));
678         return _color[index];
679     }
680
681     TCoordinate4& TriangleMesh3::texture(GLint index)
682     {
683         assert("The given texture index is out of bounds!" &&
684                (index >= 0 && index < (GLint)_tex.size()));
685         return _tex[index];
686     }
687
688     const TCoordinate4& TriangleMesh3::texture(GLint index) const
689     {
690         assert("The given texture index is out of bounds!" &&
691                (index >= 0 && index < (GLint)_tex.size()));
692         return _tex[index];
693     }
694
695     TriangularFace& TriangleMesh3::face(GLint index)
696     {
697         assert("The given face index is out of bounds!" &&
698                (index >= 0 && index < (GLint)_face.size()));
699         return _face[index];
```


© Ágoston Róth, 2018

```
689     }
690     const TriangularFace& TriangleMesh3::face(GLint index) const
691     {
692         assert("The given face index is out of bounds!" &&
693                (index >= 0 && index < (GLint)_face.size()));
694         return _face[index];
695     }
696     // clone function required by smart pointers based on the deep copy ownership policy
697     TriangleMesh3* TriangleMesh3::clone() const
698     {
699         return new (nothrow) TriangleMesh3(*this);
700     }
701     // destructor
702     TriangleMesh3::~TriangleMesh3()
703     {
704         deleteVertexBufferObjects();
705     }
706 }
```

2.14 Abstract linear combinations and tensor product surfaces

We also ensure abstract base classes for arbitrary linear combinations (LinearCombination3) and tensor product surfaces (TensorProductSurface3) that are able to generate their images, to update and render their control polygons or nets and to solve curve or surface interpolation problems – provided that the user redeclares and defines in derived classes those pure virtual methods that appear in the interfaces of these abstract classes and are responsible for the evaluation of blending functions and of (partial) derivatives up to a specified maximum order of differentiation.

Pure virtual methods that have to bo redeclared and defined in derived classes are:

- LinearCombination3::*blendingFunctionValues*;
- LinearCombination3::*calculateDerivatives*;
- TensorProductSurface3::*blendingFunctionValues*;
- TensorProductSurface3::*calculateAllPartialDerivatives*; and
- TensorProductSurface3::*calculateDirectionalDerivatives*.

As we will see in Listings 2.50/241 and 2.52/250, B-curves and B-surfaces of type (1.19/4) and (1.20/5) will be derived from classes LinearCombination3 and TensorProductSurface3, respectively.

The diagram of the abstract class LinearCombination3 is illustrated in Fig. 2.22/186, while its definition and implementation can be found in Listings 2.42/185 and 2.43/188, respectively.

Listing 2.42. Abstract linear combinations (**Core/Geometry/Curves/LinearCombinations3.h**)

```
1  #ifndef LINEARCOMBINATIONS3_H
2  #define LINEARCOMBINATIONS3_H
3  #include "../Coordinates/Cartesians3.h"
4  #include "GenericCurves3.h"
5  #include "../../Math/Matrices.h"
6  #include "../../Shaders/ShaderPrograms.h"
7  namespace cagd
8  {
9      // Represents the linear combination c(u) = sum_{i=0}^{n} p_i b_{n,i}(u; u_min, u_max), where the vectors [p_i]_{i=0}^{n} in M_{1,n+1}(R^3)
10     // usually form a control polygon of points (but they may also denote other user-defined information vectors
11     // like first and higher order derivatives as in the case of polynomial Hermite curves), while the function system
12     // {b_{n,i}(u; u_min, u_max) : u in [u_min, u_max]}_{i=0}^{n} forms the basis of some not necessarily EC vector space
```


© Ágoston Róth, 2018

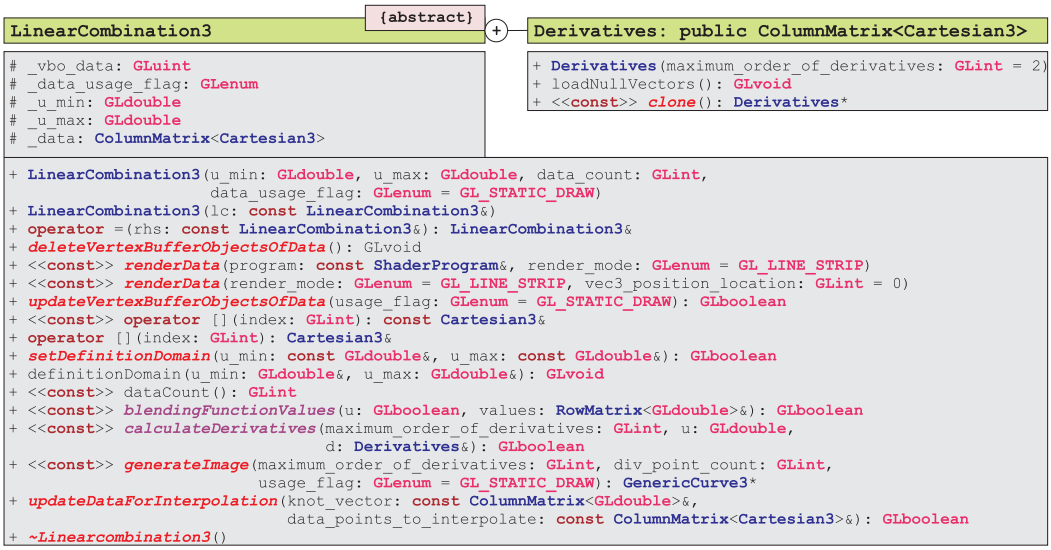


Fig. 2.22: Class diagram of abstract linear combinations

```
13      // of functions.
14      // If vectors [p_i]_{i=0}^n do not describe a control polygon, then the virtual vertex buffer object handling methods
15      // deleteVertexBufferObjectsOfData, renderData and updateVertexBufferObjectsOfData should be redeclared
16      // and redefined in derived classes.
17      // Virtual methods generateImage and updateDataForInterpolation should be redeclared and redefined in
18      // derived classes only when the user knows more efficient techniques for these tasks.
19      // Pure virtual methods blendingFunctionValues and calculateDerivatives and should be obligatorily redeclared
20      // and defined in derived classes, otherwise the user cannot instantiate objects from them.
21      class LinearCombination3
22      {
23      public:
24          // public nested class that represents a curve point and its associated first and higher order derivatives
25          class Derivatives: public ColumnMatrix<Cartesian3>
26          {
27          public:
28              // default/special constructor
29              Derivatives(GLint maximum_order_of_derivatives = 2);

30              // when called, all inherited Cartesian coordinates are set to the origin
31              GLvoid loadNullVectors();

32              // clone function required by smart pointers based on the deep copy ownership policy
33              Derivatives* clone() const;
34          };

35      protected:
36          GLuint                          _vbo_data;          // vertex buffer object of the user-specified
37                                                              // data that usually represents a control polygon
38          GLenum                          _data_usage_flag;   // usage flag of the previous vertex buffer object
39          GLdouble                        _u_min, _u_max;     // endpoints of the definition domain
40          ColumnMatrix<Cartesian3> _data;                     // vectors (usually representing control points)
41                                                              // that have to be blended

42      public:
43          // special constructor
44          LinearCombination3(GLdouble u_min, GLdouble u_max,
45                             GLint data_count, GLenum data_usage_flag = GL_STATIC_DRAW);

46          // copy constructor
47          LinearCombination3(const LinearCombination3& lc);

48          // assignment operator
49          LinearCombination3& operator =(const LinearCombination3& rhs);
```


© Ágoston Róth, 2018

```
50          // By default, it is assumed that the vectors stored in ColumnMatrix<Cartesian3> _data
51          // represent the vertices of a control polygon. If this is not the case, the next four virtual
52          // VBO handling methods can be redeclared and defined in derived classes.

53          // Deletes the vertex buffer objects of the control polygon.
54          virtual GLvoid      deleteVertexBufferObjectsOfData();

55          // Control polygon rendering methods.

56          // The next rendering method will return GL_FALSE if:
57          //   - the given shader program is not active;
58          //   - the user-defined position attribute name cannot be found in the list of active attributes
59          //     of the provided shader program, or exists but is of incorrect type;
60          //   - the render_mode is incorrect (one should use one of the constants GL_LINE_STRIP, GL_LINE_LOOP
61          //     and GL_POINTS).
62          virtual GLboolean renderData(const ShaderProgram &program,
63                                       GLenum render_mode = GL_LINE_STRIP) const;

64          // If during rendering one intends to use shader program objects that are not instances of
65          // our class ShaderProgram, then one should specify an attribute location associated with
66          // positions of type vec3.
67          //
68          // The next rendering method will return GL_FALSE if:
69          //   - there is no active shader program object;
70          //   - the given position location either cannot be found in the list of active attribute locations,
71          //     or exists but is of incorrect type;
72          //   - the render_mode is incorrect (one should use one of the constants GL_LINE_STRIP, GL_LINE_LOOP
73          //     and GL_POINTS).
74          virtual GLboolean renderData(GLenum render_mode = GL_LINE_STRIP,
75                                       GLint vec3_position_location = 0) const;

76          // Updates the vertex buffer objects of the control polygon.
77          virtual GLboolean updateVertexBufferObjectsOfData(
78                  GLenum usage_flag = GL_STATIC_DRAW);

79          // get data by constant reference
80          const Cartesian3& operator [](const GLint &index) const;

81          // get data by non-constant reference
82          Cartesian3& operator [](const GLint &index);

83          // sets/returns the endpoints of the definition domain
84          virtual GLboolean setDefinitionDomain(const GLdouble &u_min,
85                                                const GLdouble &u_max);
86          GLvoid definitionDomain(GLdouble& u_min, GLdouble& u_max) const;

87          // get number of data
88          GLint dataCount() const;

89          // abstract method that should calculate a row matrix consisting of the blending function values
90          // {b_{n,i}(u) : u ∈ [u_min, u_max]}_{i=0}^n, where n + 1 denotes the number of the user-specified data
91          virtual GLboolean blendingFunctionValues(GLdouble u,
92                                                   RowMatrix<GLdouble>& values) const = 0;

93          // abstract method that should calculate the point and its associated higher order order derivatives of the
94          // linear combination c(u) = Σ_{i=0}^n p_i b_{n,i}(u) at the parameter value u ∈ [u_min, u_max], where [p_i]_{i=0}^n
95          // denotes the user-specified data
96          virtual GLboolean calculateDerivatives(GLint maximum_order_of_derivatives,
97                                                 GLdouble u, Derivatives& d) const = 0;

98          // generates the image of the given linear combination
99          virtual GenericCurve3* generateImage(
100                 GLint maximum_order_of_derivatives,
101                 GLint div_point_count, GLenum usage_flag = GL_STATIC_DRAW) const;

102         // Solves the user-specified curve interpolation problem c(u_j) = d_j ∈ R^3, j = 0, 1, ..., n, where [u_j]_{j=0}^n
103         // forms a knot vector consisting of strictly increasing subdivision points of the definition domain [u_min, u_max],
104         // while [d_j]_{j=0}^n denotes the data points that have to be interpolated.
105         // In case of success, the solution [p_i]_{i=0}^n will be stored in the column matrix _data.
106         virtual GLboolean updateDataForInterpolation(
107                 const ColumnMatrix<GLdouble>& knot_vector,
108                 const ColumnMatrix<Cartesian3>& data_points_to_interpolate);
```


© Ágoston Róth, 2018

```
109         // destructor
110         virtual ~LinearCombination3();
111     };
112 }

113 #endif // LINEARCOMBINATIONS3_H
```

Listing 2.43. Abstract linear combinations (**Core/Geometry/Curves/LinearCombinations3.cpp**)

```
1  #include "LinearCombinations3.h"

2  #include "../../Math/RealMatrices.h"
3  #include "../../Math/RealMatrixDecompositions.h"

4  #include <algorithm>

5  using namespace std;

6  namespace cagd
7  {
8      // default/special constructor
9      LinearCombination3::Derivatives::Derivatives(GLint maximum_order_of_derivatives):
10         ColumnMatrix<Cartesian3>(maximum_order_of_derivatives + 1)
11     {
12     }

13     // when called, all inherited Cartesian coordinates are set to the origin
14     GLvoid LinearCombination3::Derivatives::loadNullVectors()
15     {
16         #pragma omp parallel for
17         for (GLint i = 0; i < _row_count; i++)
18         {
19             Cartesian3 &p_i = _data[i];

20             for (GLint j = 0; j < 3; j++)
21             {
22                 p_i[j] = 0.0;
23             }
24         }
25     }

26     // clone function required by smart pointers based on the deep copy ownership policy
27     LinearCombination3::Derivatives* LinearCombination3::Derivatives::clone() const
28     {
29         return new (nothrow) Derivatives(*this);
30     }

31     // special constructor
32     LinearCombination3::LinearCombination3(GLdouble u_min, GLdouble u_max,
33                                            GLint data_count, GLenum data_usage_flag):
34         _vbo_data(0),
35         _data_usage_flag(data_usage_flag),
36         _u_min(u_min), _u_max(u_max),
37         _data(data_count)
38     {
39     }

40     // copy constructor
41     LinearCombination3::LinearCombination3(const LinearCombination3 &lc):
42         _vbo_data(0),
43         _data_usage_flag(lc._data_usage_flag),
44         _u_min(lc._u_min), _u_max(lc._u_max),
45         _data(lc._data)
46     {
47         if (lc._vbo_data)
48         {
49             updateVertexBufferObjectsOfData(_data_usage_flag);
50         }
51     }

52     // assignment operator
```


© Ágoston Róth, 2018

```
53      LinearCombination3& LinearCombination3::operator =(const LinearCombination3& rhs)
54      {
55          if (this != &rhs)
56          {
57              deleteVertexBufferObjectsOfData();
58              _data_usage_flag = rhs._data_usage_flag;
59              _u_min = rhs._u_min;
60              _u_max = rhs._u_max;
61              _data = rhs._data;
62              if (rhs._vbo_data)
63              {
64                  updateVertexBufferObjectsOfData(_data_usage_flag);
65              }
66          }
67          return *this;
68      }
69      // Deletes the vertex buffer objects of the control polygon.
70      GLvoid LinearCombination3::deleteVertexBufferObjectsOfData()
71      {
72          if (_vbo_data)
73          {
74              glDeleteBuffers(1, &_vbo_data);
75              _vbo_data = 0;
76          }
77      }
78      // Control polygon rendering methods.
79      // The next rendering method will return GL_FALSE if:
80      //   - the given shader program is not active;
81      //   - the user-defined position attribute name cannot be found in the list of active attributes
82      //     of the provided shader program, or exists but is of incorrect type;
83      //   - the render_mode is incorrect (one should use one of the constants GL_LINE_STRIP, GL_LINE_LOOP
84      //     and GL_POINTS).
85      GLboolean LinearCombination3::renderData(const ShaderProgram &program,
86                                               GLenum render_mode) const
87      {
88          GLint current_program;
89          glGetIntegerv(GL_CURRENT_PROGRAM, &current_program);
90          if (current_program != (GLint)program._id)
91          {
92              return GL_FALSE;
93          }
94          GLint position = program.positionAttributeLocation();
95          if (!_vbo_data || position < 0)
96          {
97              return GL_FALSE;
98          }
99          if (render_mode != GL_LINE_STRIP && render_mode != GL_LINE_LOOP &&
100             render_mode != GL_POINTS)
101         {
102             return GL_FALSE;
103         }
104         glEnableVertexAttribArray(position);
105         glBindBuffer(GL_ARRAY_BUFFER, _vbo_data);
106         glVertexAttribPointer(position, 3, GL_FLOAT, GL_FALSE, 0, nullptr);
107         glDrawArrays(render_mode, 0, _data.rowCount());
108         glBindBuffer(GL_ARRAY_BUFFER, 0);
109         glDisableVertexAttribArray(position);
110         glVertexAttrib3f(position, 0.0f, 0.0f, 0.0f);
111         return GL_TRUE;
112     }
```


© Ágoston Róth, 2018

```
113     // If during rendering one intends to use shader program objects that are not instances of
114     // our class ShaderProgram, then one should specify an attribute location associated with
115     // positions of type vec3.
116     //
117     // The next rendering method will return GL_FALSE if:
118     //  - there is no active shader program object;
119     //  - the given position location either cannot be found in the list of active attribute locations,
120     //    or exists but is of incorrect type;
121     //  - the render_mode is incorrect (one should use one of the constants GL_LINE_STRIP, GL_LINE_LOOP
122     //    and GL_POINTS).
123     GLboolean LinearCombination3::renderData(GLenum render_mode,
124                                              GLint vec3_position_location) const
125     {
126         if (!_vbo_data || vec3_position_location < 0)
127         {
128             return GL_FALSE;
129         }
130         if (render_mode != GL_LINE_STRIP && render_mode != GL_LINE_LOOP &&
131             render_mode != GL_POINTS)
132         {
133             return GL_FALSE;
134         }
135         GLint current_program = 0;
136         glGetIntegerv(GL_CURRENT_PROGRAM, &current_program);
137         if (!current_program)
138         {
139             return GL_FALSE;
140         }
141         GLint attribute_count;
142         glGetProgramiv(current_program, GL_ACTIVE_ATTRIBUTES, &attribute_count);
143         if (!attribute_count)
144         {
145             return GL_FALSE;
146         }
147         GLsizei max_attribute_name_length;
148         glGetProgramiv(current_program, GL_ACTIVE_ATTRIBUTE_MAX_LENGTH,
149                        &max_attribute_name_length);
150         GLchar *attribute_name_data = new GLchar[max_attribute_name_length];
151         GLboolean given_location_exists_and_is_of_type_vec3 = GL_FALSE;
152         for (GLint attribute = 0;
153              attribute < attribute_count && !given_location_exists_and_is_of_type_vec3;
154              attribute++)
155         {
156             GLsizei actual_length = 0;
157             GLint   array_size    = 0;
158             GLenum  type          = 0;
159             glGetActiveAttrib(current_program, attribute, max_attribute_name_length,
160                               &actual_length, &array_size, &type, attribute_name_data);
161             string name(&attribute_name_data[0], actual_length);
162             GLint  location = glGetAttribLocation(current_program, name.c_str());
163             if (type == GL_FLOAT_VEC3 && vec3_position_location == location)
164             {
165                 given_location_exists_and_is_of_type_vec3 = GL_TRUE;
166             }
167         }
168         delete[] attribute_name_data;
169         if (!given_location_exists_and_is_of_type_vec3)
170         {
171             return GL_FALSE;
172         }
```


© Ágoston Róth, 2018

```
173         glEnableVertexAttribArray(vec3_position_location);
174         glBindBuffer(GL_ARRAY_BUFFER, _vbo_data);
175         glVertexAttribPointer(vec3_position_location, 3, GL_FLOAT, GL_FALSE, 0, nullptr);
176         glDrawArrays(render_mode, 0, (GLsizei)_data.rowCount());
177         glBindBuffer(GL_ARRAY_BUFFER, 0);
178         glDisableVertexAttribArray(vec3_position_location);
179         glVertexAttrib3f(vec3_position_location, 0.0f, 0.0f, 0.0f);
180         return GL_TRUE;
181     }
182     // Updates the vertex buffer objects of the control polygon.
183     GLboolean LinearCombination3::updateVertexBufferObjectsOfData(GLenum usage_flag)
184     {
185         GLint data_count = _data.rowCount();
186         if (!data_count)
187         {
188             return GL_FALSE;
189         }
190         if (usage_flag != GL_STREAM_DRAW  && usage_flag != GL_STREAM_READ  &&
191             usage_flag != GL_STREAM_COPY  &&
192             usage_flag != GL_DYNAMIC_DRAW && usage_flag != GL_DYNAMIC_READ &&
193             usage_flag != GL_DYNAMIC_COPY &&
194             usage_flag != GL_STATIC_DRAW  && usage_flag != GL_STATIC_READ  &&
195             usage_flag != GL_STATIC_COPY)
196         {
197             return GL_FALSE;
198         }
199         _data_usage_flag = usage_flag;
200         deleteVertexBufferObjectsOfData();
201         glGenBuffers(1, &_vbo_data);
202         if (!_vbo_data)
203             return GL_FALSE;
204         glBindBuffer(GL_ARRAY_BUFFER, _vbo_data);
205         glBufferData(GL_ARRAY_BUFFER, data_count * 3 * sizeof(GLfloat), nullptr,
206                      _data_usage_flag);
207         GLfloat *coordinate = (GLfloat*)glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
208         if (!coordinate)
209         {
210             glBindBuffer(GL_ARRAY_BUFFER, 0);
211             deleteVertexBufferObjectsOfData();
212             return GL_FALSE;
213         }
214         for (GLint i = 0; i < data_count; i++)
215         {
216             Cartesian3 &p_i = _data[i];
217             for (GLint j = 0; j < 3; j++, coordinate++)
218             {
219                 *coordinate = (GLfloat)p_i[j];
220             }
221         }
222         if (!glUnmapBuffer(GL_ARRAY_BUFFER))
223         {
224             glBindBuffer(GL_ARRAY_BUFFER, 0);
225             deleteVertexBufferObjectsOfData();
226             return GL_FALSE;
227         }
228         glBindBuffer(GL_ARRAY_BUFFER, 0);
```


© Ágoston Róth, 2018

```
229         return GL_TRUE;
230     }
231     // get data by constant reference
232     const Cartesian3& LinearCombination3::operator [](const GLint &index) const
233     {
234         assert("The given index is out of bounds!" &&
235                (index >= 0 && index < _data.rowCount()));
236         return _data[index];
237     }
238     // get data by non-constant reference
239     Cartesian3& LinearCombination3::operator [](const GLint &index)
240     {
241         assert("The given index is out of bounds!" &&
242                (index >= 0 && index < _data.rowCount()));
243         return _data[index];
244     }
245     // sets/returns the endpoints of the definition domain
246     GLboolean LinearCombination3::setDefinitionDomain(
247         const GLdouble &u_min, const GLdouble &u_max)
248     {
249         if (u_min >= u_max)
250         {
251             return GL_FALSE;
252         }
253         _u_min = u_min;
254         _u_max = u_max;
255         return GL_TRUE;
256     }
257     GLvoid LinearCombination3::definitionDomain(GLdouble& u_min, GLdouble& u_max) const
258     {
259         u_min = _u_min;
260         u_max = _u_max;
261     }
262     // get number of data
263     GLint LinearCombination3::dataCount() const
264     {
265         return _data.rowCount();
266     }
267     // generates the image of the given linear combination
268     GenericCurve3* LinearCombination3::generateImage(
269         GLint maximum_order_of_derivatives,
270         GLint div_point_count, GLenum usage_flag) const
271     {
272         GenericCurve3 *result = new (nothrow) GenericCurve3(
273                 maximum_order_of_derivatives, div_point_count, usage_flag);
274         if (!result)
275         {
276             return nullptr;
277         }
278         GLdouble step = (_u_max - _u_min) / (div_point_count - 1);
279         GLboolean aborted = GL_FALSE;
280         #pragma omp parallel for
281         for (GLint i = 0; i < div_point_count; i++)
282         {
283             #pragma omp flush (aborted)
284             if (!aborted)
285             {
286                 GLdouble u = min(_u_max, _u_min + i * step);
287                 Derivatives d;
```


© Ágoston Róth, 2018

```
288                 if (!calculateDerivatives(maximum_order_of_derivatives, u, d))
289                 {
290                     aborted = GL_TRUE;
291                     #pragma omp flush (aborted)
292                 }
293                 result->_derivative.setColumn(i, d);
294             }
295         }
296         if (aborted)
297         {
298             delete result, result = nullptr;
299         }
300         return result;
301     }
302     // Solves the user-specified curve interpolation problem c(u_j) = d_j ∈ R^3, j = 0, 1, ..., n, where [u_j]_{j=0}^n
303     // forms a knot vector consisting of strictly increasing subdivision points of the definition domain [u_min, u_max],
304     // while [d_j]_{j=0}^n denotes the data points that have to be interpolated.
305     // In case of success, the solution [p_i]_{i=0}^n will be stored in the column matrix _data.
306     GLboolean LinearCombination3::updateDataForInterpolation(
307         const ColumnMatrix<GLdouble>& knot_vector,
308         const ColumnMatrix<Cartesian3>& data_points_to_interpolate)
309     {
310         GLint data_count = _data.rowCount();
311         if (data_count != knot_vector.rowCount() ||
312             data_count != data_points_to_interpolate.rowCount())
313         {
314             return GL_FALSE;
315         }
316         GLboolean  collocation_matrix_construction_aborted = GL_FALSE;
317         RealMatrix collocation_matrix(data_count, data_count);
318         #pragma omp parallel for
319         for (GLint r = 0; r < knot_vector.rowCount(); r++)
320         {
321             #pragma omp flush (collocation_matrix_construction_aborted)
322             if (!collocation_matrix_construction_aborted)
323             {
324                 RowMatrix<GLdouble> current_blending_function_values(data_count);
325                 if (!blendingFunctionValues(knot_vector[r],
326                                             current_blending_function_values))
327                 {
328                     collocation_matrix_construction_aborted = GL_TRUE;
329                     #pragma omp flush (collocation_matrix_construction_aborted)
330                 }
331                 else
332                 {
333                     collocation_matrix.setRow(r, current_blending_function_values);
334                 }
335             }
336         }
337         if (collocation_matrix_construction_aborted)
338         {
339             return GL_FALSE;
340         }
341         PLUDecomposition PLUD(collocation_matrix);
342         return PLUD.solveLinearSystem(data_points_to_interpolate, _data);
343     }
344     // destructor
345     LinearCombination3::~LinearCombination3()
346     {
347         deleteVertexBufferObjectsOfData();
348     }
349 }
```

The diagram of the abstract class TensorProductSurface3 is illustrated in Fig. 2.23/194, while

© Ágoston Róth, 2018

its definition and implementation can be found in Listings 2.44/194 and 2.45/197, respectively.



Fig. 2.23: Class diagram of abstract tensor product surfaces

Listing 2.44. Abstract tensor product surfaces (**Core/Geometry/Surfaces/TensorProductSurfaces3.h**)

```
1 #ifndef TENSORPRODUCTSURFACES3_H
2 #define TENSORPRODUCTSURFACES3_H

3 #include <GL/glew.h>
```


© Ágoston Róth, 2018

```
4  #include "../Coordinates/Cartesians3.h"
5  #include "../../Math/Constants.h"
6  #include "../../Math/Matrices.h"
7  #include "../../Geometry/Curves/GenericCurves3.h"
8  #include "../../SmartPointers/SpecializedSmartPointers.h"
9  #include "../../Shaders/ShaderPrograms.h"
10 #include "TriangleMeshes3.h"

11 #include <iostream>
12 #include <vector>

13 namespace cagd
14 {
15     // Represents the tensor product surface s(u,v) = sum_{i=0}^{n} sum_{j=0}^{m} p_{i,j} b_{n,i}(u; u_min, u_max) b_{m,j}(v; v_min, v_max),
16     // where the vectors [p_{i,j}]_{i=0,j=0}^{n,m} in M_{n+1,m+1}(R^3) usually form a control net of points (but they may also
17     // denote other user-defined information vectors like first and higher order partial derivatives like in the case
18     // of polynomial Hermite patches), while function systems {b_{n,i}(u; u_min, u_max) : u in [u_min, u_max]}_{i=0}^{n} and
19     // {b_{m,j}(v; v_min, v_max) : v in [v_min, v_max]}_{j=0}^{m} form the bases of some not necessarily EC vector spaces of functions.
20     // If vectors [p_{i,j}]_{i=0,j=0}^{n,m} do not form a control net, then the vertex buffer object handling virtual methods
21     // deleteVertexBufferObjectsOfData, renderData and updateVertexBufferObjectsOfData should be redeclared
22     // and redefined in derived classes.
23     // Virtual methods generateImage and updateDataForInterpolation should be redeclared and redefined in
24     // derived classes only when the user knows more efficient techniques for these tasks.
25     // Pure virtual methods blendingFunctionValues, calculateAllPartialDerivatives and calculateDirectional-
26     // Derivatives should be obligatorily redeclared and defined in derived classes, otherwise the user cannot
27     // instantiate objects from them.
28     class TensorProductSurface3
29     {
30     public:
31         // When generating the image of the underlying tensor product surface, the user is able to choose different
32         // color schemes based on the following predefined pointwise zeroth, first and second order energy variations.
33         // In each case the obtained color variation is determined by using the utility function coldToHotColorMap
34         // presented in Listing 2.34/127 that generates a rainbow-like color map based on the minimum and maximum
35         // value of the applied energy.
36         // If x(u,v), y(u,v), z(u,v), ||n(u,v)||, K(u,v) and H(u,v) denote the x-, y-, z-coordinates, the length of
37         // the normal vector, the Gaussian- and mean curvatures of the surface point s(u,v), respectively, then the
38         // following color schemes reflect the pointwise variation of the energy quantity phi(u,v), where:
39         enum ImageColorScheme
40         {
41             DEFAULT_NULL_FRAGMENT = 0,                  // phi(u,v) = 0
42             X_VARIATION_FRAGMENT,                       // phi(u,v) = x(u,v)
43             Y_VARIATION_FRAGMENT,                       // phi(u,v) = y(u,v)
44             Z_VARIATION_FRAGMENT,                       // phi(u,v) = z(u,v)
45             NORMAL_LENGTH_FRAGMENT,                     // phi(u,v) = ||n(u,v)||
46             GAUSSIAN_CURVATURE_FRAGMENT,                // phi(u,v) = K(u,v) ||n(u,v)||
47             MEAN_CURVATURE_FRAGMENT,                    // phi(u,v) = H(u,v) ||n(u,v)||
48             WILLMORE_ENERGY_FRAGMENT,                   // phi(u,v) = H^2(u,v) ||n(u,v)||
49             LOG_WILLMORE_ENERGY_FRAGMENT,               // phi(u,v) = ln(1 + H^2(u,v)) ||n(u,v)||
50             UMBILIC_DEVIATION_ENERGY_FRAGMENT,          // phi(u,v) = 4 (H^2(u,v) - K(u,v)) ||n(u,v)||
51             LOG_UMBILIC_DEVIATION_ENERGY_FRAGMENT,      // phi(u,v) = ln(1 + 4 (H^2(u,v) - K(u,v))) ||n(u,v)||
52             TOTAL_CURVATURE_ENERGY_FRAGMENT,            // phi(u,v) = (3/2 H^2(u,v) - 1/2 K(u,v)) ||n(u,v)||
53             LOG_TOTAL_CURVATURE_ENERGY_FRAGMENT         // phi(u,v) = ln(1 + 3/2 H^2(u,v) - 1/2 K(u,v)) ||n(u,v)||
54         };

55         // a nested public class that represents a triangular matrix that consists of all (mixed) partial derivatives
56         // up to a maximum order
57         class PartialDerivatives: public TriangularMatrix<Cartesian3>
58         {
59         public:
60             // default/special constructor
61             PartialDerivatives(GLint maximum_order_of_partial_derivatives = 0);
62             // when called, all inherited Cartesian coordinates are set to the origin
63             GLvoid loadNullVectors();
64             // clone function required by smart pointers based on the deep copy ownership policy
65             PartialDerivatives* clone() const;
66         };

67         // nested public class that represents a column matrix consisting of either u- or v-directional partial
68         // derivatives up to a maximum order
69         class DirectionalDerivatives: public ColumnMatrix<Cartesian3>
70         {
71         public:
```


© Ágoston Róth, 2018

```
72              // default/special constructor
73              DirectionalDerivatives(GLint maximum_order_of_directional_derivatives = 0);
74              // when called, all inherited Cartesian coordinates are set to the origin
75              GLvoid loadNullVectors();
76              // clone function required by smart pointers based on the deep copy ownership policy
77              DirectionalDerivatives* clone() const;
78          };
79      protected:
80          // endpoints of the u- and v-intervals of the definition domain
81          std::vector<GLdouble>    _min_value, _max_value;
82          // boolean flags that indicate whether the surface is closed in a given direction
83          std::vector<GLboolean>   _closed;
84          // vertex buffer object of the user-specified _data that usually describes a control net
85          GLuint                   _vbo_data;
86          Matrix<Cartesian3>       _data; // [p_{i,j}]_{i=0,j=0}^{n,m} ∈ M_{n+1,m+1}(R^3)
87      public:
88          // special constructor
89          TensorProductSurface3(
90                  GLdouble u_min, GLdouble u_max,
91                  GLdouble v_min, GLdouble v_max,
92                  GLint row_count = 4, GLint column_count = 4,
93                  GLboolean u_closed = GL_FALSE, GLboolean v_closed = GL_FALSE);
94          // copy constructor
95          TensorProductSurface3(const TensorProductSurface3& surface);
96          // assignment operator
97          TensorProductSurface3& operator =(const TensorProductSurface3& surface);
98          // sets/queries the definition domain of the surface in a given direction
99          virtual GLboolean setInterval(variable::Type type,
100                                        GLdouble min_value, GLdouble max_value);
101         GLvoid interval(variable::Type type,
102                         GLdouble &min_value, GLdouble &max_value) const;
103         // get data by constant reference
104         const Cartesian3& operator ()(GLint row, GLint column) const;
105         // get data by non-constant reference
106         Cartesian3& operator ()(GLint row, GLint column);
107         // a pure virtual method that should calculate a row matrix that consists of either u- or v-directional
108         // blending function values
109         virtual GLboolean blendingFunctionValues(
110                 variable::Type type,
111                 GLdouble parameter_value, RowMatrix<GLdouble>& values) const = 0;
112         // pure virtual methods that should calculate the point and higher order (mixed) partial derivatives of the
113         // given surface
114         virtual GLboolean calculateAllPartialDerivatives(
115                 GLint maximum_order_of_partial_derivatives,
116                 GLdouble u, GLdouble v, PartialDerivatives& pd) const = 0;
117         virtual GLboolean calculateDirectionalDerivatives(
118                 variable::Type direction,
119                 GLint maximum_order_of_directional_derivatives,
120                 GLdouble u, GLdouble v, DirectionalDerivatives& d) const = 0;
121         // generates a triangle mesh that approximates the shape of the given tensor product surface
122         virtual TriangleMesh3* generateImage(
123                 GLint u_div_point_count, GLint v_div_point_count,
124                 ImageColorScheme color_scheme = DEFAULT_NULL_FRAGMENT,
125                 GLenum usage_flag = GL_STATIC_DRAW) const;
126         // generates either u- or v-directional isoparametric lines of the given tensor product surface
127         virtual RowMatrix<SP<GenericCurve3>::Default>* generateIsoparametricLines(
128                 variable::Type type, GLint iso_line_count,
129                 GLint maximum_order_of_derivatives, GLint div_point_count,
130                 GLenum usage_flag = GL_STATIC_DRAW) const;
131         // Tries to solve the surface interpolation problem s(u_i, v_j) = d_{i,j}, i = 0,1,...,n, j = 0,1,...,m,
132         // where [u_i]_{i=0}^{n} ⊂ [u_min, u_max] and [v_j]_{j=0}^{m} ⊂ [v_min, v_max] are two strictly increasing knot vectors,
```


© Ágoston Róth, 2018

```
133         // while [d_{i,j}]_{i=0,j=0}^{n,m} denotes the grid of data points that has to be interpolated.
134         // In case of success, the solution will be stored in the member field _data that corresponds to [p_{i,j}]_{i=0,j=0}^{n,m}.
135         GLboolean updateDataForInterpolation(
136                 const RowMatrix<GLdouble> &u_knot_vector,
137                 const ColumnMatrix<GLdouble> &v_knot_vector,
138                 Matrix<Cartesian3> &data_points_to_interpolate);
139         // By default, it is assumed that the vectors stored in Matrix<Cartesian3> _data
140         // represent the vertices of a control net. If this is not the case, the next virtual
141         // VBO handling methods can be redeclared and redefined in derived classes.
142         // Deletes the vertex buffer objects of the control net.
143         virtual GLvoid      deleteVertexBufferObjectsOfData();
144         // Control net rendering methods.
145         // The next rendering method will return GL_FALSE if:
146         //   - the given shader program is not active;
147         //   - the user-defined position attribute name cannot be found in the list of active attributes
148         //     of the provided shader program, or exists but is of incorrect type;
149         //   - the u_render_mode or v_render_mode is incorrect (one should use one of the constants GL_LINE_STRIP,
150         //     GL_LINE_LOOP and GL_POINTS).
151         virtual GLboolean renderData(
152                 const ShaderProgram &program,
153                 GLenum u_render_mode = GL_LINE_STRIP,
154                 GLenum v_render_mode = GL_LINE_STRIP) const;
155         // If during rendering one intends to use shader program objects that are not instances of
156         // our class ShaderProgram, then one should specify an attribute location associated with
157         // positions of type vec3.
158         //
159         // The next rendering method will return GL_FALSE if:
160         //   - there is no active shader program object;
161         //   - the given position location either cannot be found in the list of active attribute locations,
162         //     or exists but is of incorrect type;
163         //   - the u_render_mode or v_render_mode is incorrect (one should use one of the constants GL_LINE_STRIP,
164         //     GL_LINE_LOOP and GL_POINTS).
165         virtual GLboolean renderData(
166                 GLenum u_render_mode = GL_LINE_STRIP,
167                 GLenum v_render_mode = GL_LINE_STRIP,
168                 GLint vec3_position_location = 0) const;
169         // Updates the vertex buffer objects of the control net.
170         virtual GLboolean updateVertexBufferObjectsOfData(
171                 GLenum usage_flag = GL_STATIC_DRAW);
172         // get the row count of the data grid
173         GLint rowCount() const;
174         // get the column count of the data grid
175         GLint columnCount() const;
176         // destructor
177         virtual ~TensorProductSurface3();
178     };
179 }
180 #endif // TENSORPRODUCTSURFACES3_H
```

Listing 2.45. Abstract tensor product surfaces (**Core/Geometry/Surfaces/TensorProductSurfaces3.cpp**)

```
1 #include "TensorProductSurfaces3.h"
2 #include "../../Math/RealMatrices.h"
3 #include "../../Math/RealMatrixDecompositions.h"
4 #include "../../Utilities.h"
5 #include <algorithm>
6 using namespace std;
7 namespace cagd
```


© Ágoston Róth, 2018

```
8  {
9      // default/special constructor
10     TensorProductSurface3::PartialDerivatives::PartialDerivatives(
11         GLint maximum_order_of_partial_derivatives):
12         TriangularMatrix<Cartesian3>(
13             maximum_order_of_partial_derivatives < 0 ?
14             0 : maximum_order_of_partial_derivatives + 1)
15     {
16     }

17     // when called, all inherited Cartesian coordinates are set to the origin
18     GLvoid TensorProductSurface3::PartialDerivatives::loadNullVectors()
19     {
20         #pragma omp parallel for
21         for (GLint r = 0; r < _row_count; r++)
22         {
23             for (GLint c = 0; c <= r; c++)
24             {
25                 Cartesian3 &partial_derivative = _data[r][c];

26                 for (GLint i = 0; i < 3; i++)
27                 {
28                     partial_derivative[i] = 0.0;
29                 }
30             }
31         }
32     }

33     // clone function required by smart pointers based on the deep copy ownership policy
34     TensorProductSurface3::PartialDerivatives*
35         TensorProductSurface3::PartialDerivatives::clone() const
36     {
37         return new (nothrow) TensorProductSurface3::PartialDerivatives(*this);
38     }

39     // default/special constructor
40     TensorProductSurface3::DirectionalDerivatives::DirectionalDerivatives(
41         GLint maximum_order_of_directional_derivatives):
42         ColumnMatrix<Cartesian3>(
43             maximum_order_of_directional_derivatives < 0 ?
44             0 : maximum_order_of_directional_derivatives + 1)
45     {
46     }

47     // when called, all inherited Cartesian coordinates are set to the origin
48     GLvoid TensorProductSurface3::DirectionalDerivatives::loadNullVectors()
49     {
50         #pragma omp parallel for
51         for (GLint r = 0; r < _row_count; r++)
52         {
53             Cartesian3 &directional_derivative = _data[r];

54             for (GLint i = 0; i < 3; i++)
55             {
56                 directional_derivative[i] = 0.0;
57             }
58         }
59     }

60     // clone function required by smart pointers based on the deep copy ownership policy
61     TensorProductSurface3::DirectionalDerivatives*
62         TensorProductSurface3::DirectionalDerivatives::clone() const
63     {
64         return new (nothrow) TensorProductSurface3::DirectionalDerivatives(*this);
65     }

66     // special constructor
67     TensorProductSurface3::TensorProductSurface3(GLdouble u_min, GLdouble u_max,
68             GLdouble v_min, GLdouble v_max,
69             GLint row_count, GLint column_count,
70             GLboolean u_closed, GLboolean v_closed):
71         _min_value(2), _max_value(2), _closed(2),
72         _vbo_data(0),
73         _data(row_count, column_count)
```


© Ágoston Róth, 2018

```
74      {
75          _min_value[variable::U] = u_min,     _max_value[variable::U] = u_max;
76          _min_value[variable::V] = v_min,     _max_value[variable::V] = v_max;
77          _closed[variable::U]    = u_closed,  _closed[variable::V]    = v_closed;
78      }

79      // copy constructor
80      TensorProductSurface3::TensorProductSurface3(const TensorProductSurface3 &surface):
81          _min_value(surface._min_value),
82          _max_value(surface._max_value),
83          _closed(surface._closed),
84          _vbo_data(0),
85          _data(surface._data)
86      {
87          if (surface._vbo_data)
88          {
89              updateVertexBufferObjectsOfData();
90          }
91      }

92      // assignment operator
93      TensorProductSurface3& TensorProductSurface3::operator =(
94          const TensorProductSurface3 &rhs)
95      {
96          if (this != &rhs)
97          {
98              deleteVertexBufferObjectsOfData();

99              _min_value = rhs._min_value;
100             _max_value = rhs._max_value;
101             _closed    = rhs._closed;

102             _data = rhs._data;

103             if (rhs._vbo_data)
104             {
105                 updateVertexBufferObjectsOfData();
106             }
107         }

108         return *this;
109     }

110     // sets/queries the definition domain of the surface in a given direction
111     GLboolean TensorProductSurface3::setInterval(
112         variable::Type type,
113         GLdouble min_value, GLdouble max_value)
114     {
115         _min_value[type] = min_value;
116         _max_value[type] = max_value;

117         return GL_TRUE;
118     }

119     void TensorProductSurface3::interval(
120         variable::Type type,
121         GLdouble &min_value, GLdouble &max_value) const
122     {
123         min_value = _min_value[type];
124         max_value = _max_value[type];
125     }

126     // get data by constant reference
127     const Cartesian3& TensorProductSurface3::operator ()(GLint row, GLint column) const
128     {
129         assert("The given row index is out of bounds!" &&
130                (row >= 0 && row < _data.rowCount()));

131         assert("The given column index is out of bounds!" &&
132                (column >= 0 && column < _data.columnCount()));

133         return _data(row, column);
134     }
```


© Ágoston Róth, 2018

```
135     // get data by non-constant reference
136     Cartesian3& TensorProductSurface3::operator ()(GLint row, GLint column)
137     {
138         assert("The given row index is out of bounds!" &&
139                (row >= 0 && row < _data.rowCount()));
140         assert("The given column index is out of bounds!" &&
141                (column >= 0 && column < _data.columnCount()));
142         return _data(row, column);
143     }
144     // generates a triangle mesh that approximates the shape of the given tensor product surface
145     TriangleMesh3* TensorProductSurface3::generateImage(
146         GLint u_div_point_count, GLint v_div_point_count,
147         ImageColorScheme color_scheme, GLenum usage_flag) const
148     {
149         if (u_div_point_count <= 1 || v_div_point_count <= 1)
150         {
151             return GL_FALSE;
152         }
153         // calculating the number of vertices, unit normal vectors and texture coordinates
154         GLint vertex_count = u_div_point_count * v_div_point_count;
155         // calculating the number of triangular faces
156         GLint face_count = 2 * (u_div_point_count - 1) * (v_div_point_count - 1);
157         TriangleMesh3 *result = nullptr;
158         result = new (nothrow) TriangleMesh3(vertex_count, face_count, usage_flag);
159         if (!result)
160         {
161             return nullptr;
162         }
163         // step sizes of a uniform subdivision grid in the definition domain
164         GLdouble du = (_max_value[variable::U] - _min_value[variable::U]) /
165                       (u_div_point_count - 1);
166         GLdouble dv = (_max_value[variable::V] - _min_value[variable::V]) /
167                       (v_div_point_count - 1);
168         // step sizes of a uniform subdivision grid in the unit square
169         GLfloat sdu = 1.0f / (u_div_point_count - 1);
170         GLfloat tdv = 1.0f / (v_div_point_count - 1);
171         // values used for color scheme generation
172         GLint maximum_order_of_partial_derivatives = 1;
173         switch (color_scheme)
174         {
175         case DEFAULT_NULL_FRAGMENT:
176         case X_VARIATION_FRAGMENT:
177         case Y_VARIATION_FRAGMENT:
178         case Z_VARIATION_FRAGMENT:
179         case NORMAL_LENGTH_FRAGMENT:
180             maximum_order_of_partial_derivatives = 1;
181             break;
182         case GAUSSIAN_CURVATURE_FRAGMENT:
183         case MEAN_CURVATURE_FRAGMENT:
184         case WILLMORE_ENERGY_FRAGMENT:
185         case LOG_WILLMORE_ENERGY_FRAGMENT:
186         case UMBILIC_DEVIATION_ENERGY_FRAGMENT:
187         case LOG_UMBILIC_DEVIATION_ENERGY_FRAGMENT:
188         case TOTAL_CURVATURE_ENERGY_FRAGMENT:
189         case LOG_TOTAL_CURVATURE_ENERGY_FRAGMENT:
190             maximum_order_of_partial_derivatives = 2;
191             break;
192         }
193         std::vector<GLdouble> fragment(color_scheme > DEFAULT_NULL_FRAGMENT ?
194                                        vertex_count : 0);
```


© Ágoston Róth, 2018

```
195         // error flag
196         GLboolean aborted = GL_FALSE;
197         #pragma omp parallel for
198         for (GLint i_j = 0; i_j < vertex_count; i_j++)
199         {
200             #pragma omp flush (aborted)
201             if (!aborted)
202             {
203                 GLint i = i_j / v_div_point_count;
204                 GLint j = i_j % v_div_point_count;
205                 GLdouble u = min(_min_value[variable::U] + i * du,
206                                  _max_value[variable::U]);
207                 GLfloat  s = min(i * sdu, 1.0f);
208                 GLdouble v = min(_min_value[variable::V] + j * dv,
209                                  _max_value[variable::V]);
210                 GLfloat  t = min(j * tdv, 1.0f);
211                 GLint vertex_id[4];
212                 vertex_id[0] = i_j;
213                 vertex_id[1] = vertex_id[0] + 1;
214                 vertex_id[2] = vertex_id[1] + v_div_point_count;
215                 vertex_id[3] = vertex_id[2] - 1;
216                 // calculating all needed surface data (i.e., partial derivatives)
217                 PartialDerivatives pd;
218                 if (!calculateAllPartialDerivatives(
219                         maximum_order_of_partial_derivatives, u, v, pd))
220                 {
221                     aborted = GL_TRUE;
222                     #pragma omp flush (aborted)
223                 }
224                 else
225                 {
226                     // surface point
227                     (*result)._position[vertex_id[0]] = pd(0, 0);
228                     // unit surface normal
229                     Cartesian3 &normal = (*result)._normal[vertex_id[0]];
230                     normal  = pd(1, 0);
231                     normal ^= pd(1, 1);
232                     GLdouble normal_length = normal.length();
233                     if (normal_length && normal_length != 1.0)
234                     {
235                         normal /= normal_length;
236                     }
237                     // coefficients of the first fundamental form
238                     GLdouble e_1 = 0.0, f_1 = 0.0, g_1 = 0.0;
239                     // coefficients of the second fundamental form
240                     GLdouble e_2 = 0.0, f_2 = 0.0, g_2 = 0.0;
241                     // Gaussian curvature
242                     GLdouble K   = 0.0;
243                     // mean curvature
244                     GLdouble H   = 0.0;
245                     if (color_scheme > NORMAL_LENGTH_FRAGMENT)
246                     {
247                         // coefficients of the first fundamental form
248                         e_1 = pd(1, 0) * pd(1, 0);
249                         f_1 = pd(1, 0) * pd(1, 1);
250                         g_1 = pd(1, 1) * pd(1, 1);
```


© Ágoston Róth, 2018

```
251                     // coefficients of the second fundamental form
252                     e_2 = normal * pd(2, 0);
253                     f_2 = normal * pd(2, 1);
254                     g_2 = normal * pd(2, 2);
255                 }

256                 if (color_scheme >= GAUSSIAN_CURVATURE_FRAGMENT)
257                 {
258                     K = (e_2 * g_2 - f_2 * f_2) / (e_1 * g_1 - f_1 * f_1);
259                 }

260                 if (color_scheme >= MEAN_CURVATURE_FRAGMENT)
261                 {
262                     H = (g_1 * e_2 - 2.0 * f_1 * f_2 + e_1 * g_2) /
263                         (e_1 * g_1 - f_1 * f_1);
264                 }

265                 // texture coordinates
266                 TCoordinate4 &tex = (*result)._tex[vertex_id[0]];
267                 tex.s() = s;
268                 tex.t() = t;

269                 // connectivity information
270                 if (i < u_div_point_count - 1 && j < v_div_point_count - 1)
271                 {
272                     GLint face_index = 2 * (i_j - i);

273                     TriangularFace &lower_face = (*result)._face[face_index];
274                     lower_face[0] = vertex_id[0];
275                     lower_face[1] = vertex_id[1];
276                     lower_face[2] = vertex_id[2];

277                     TriangularFace &upper_face = (*result)._face[face_index + 1];
278                     upper_face[0] = vertex_id[0];
279                     upper_face[1] = vertex_id[2];
280                     upper_face[2] = vertex_id[3];
281                 }

282                 // fragment generation based on the selected color scheme
283                 if (color_scheme > DEFAULT_NULL_FRAGMENT)
284                 {
285                     switch (color_scheme)
286                     {
287                     case X_VARIATION_FRAGMENT:
288                         fragment[vertex_id[0]] = pd(0, 0).x();
289                         break;

290                     case Y_VARIATION_FRAGMENT:
291                         fragment[vertex_id[0]] = pd(0, 0).y();
292                         break;

293                     case Z_VARIATION_FRAGMENT:
294                         fragment[vertex_id[0]] = pd(0, 0).z();
295                         break;

296                     case NORMAL_LENGTH_FRAGMENT:
297                         fragment[vertex_id[0]] = normal_length;
298                         break;

299                     case GAUSSIAN_CURVATURE_FRAGMENT:
300                         fragment[vertex_id[0]] = K * normal_length;
301                         break;

302                     case MEAN_CURVATURE_FRAGMENT:
303                         fragment[vertex_id[0]] = H * normal_length;
304                         break;

305                     case WILLMORE_ENERGY_FRAGMENT:
306                         fragment[vertex_id[0]] = H * H * normal_length;
307                         break;

308                     case LOG_WILLMORE_ENERGY_FRAGMENT:
309                         fragment[vertex_id[0]] = log(1.0 + H * H) * normal_length;
310                         break;
```


© Ágoston Róth, 2018

```
311                         case UMBILIC_DEVIATION_ENERGY_FRAGMENT:
312                             fragment[vertex_id[0]] = 4.0 * (H * H - K) * normal_length;
313                             break;
314                         case LOG_UMBILIC_DEVIATION_ENERGY_FRAGMENT:
315                             fragment[vertex_id[0]] = log(1.0 + 4.0 * (H * H - K)) *
316                                                      normal_length;
317                             break;
318                         case TOTAL_CURVATURE_ENERGY_FRAGMENT:
319                             fragment[vertex_id[0]] = (1.5 * H * H - 0.5 * K) *
320                                                      normal_length;
321                             break;
322                         case LOG_TOTAL_CURVATURE_ENERGY_FRAGMENT:
323                             fragment[vertex_id[0]] = log(1.0 + 1.5 * H * H - 0.5 * K) *
324                                                      normal_length;
325                             break;
326                         default:
327                             break;
328                         }
329                     }
330                 }
331             }
332         }
333         if (aborted)
334         {
335             delete result, result = nullptr;
336             return result;
337         }
338         // generating energy based pointwise color variation
339         if (color_scheme > DEFAULT_NULL_FRAGMENT)
340         {
341             GLdouble min_fragment_value =  numeric_limits<GLdouble>::max();
342             GLdouble max_fragment_value = -numeric_limits<GLdouble>::max();
343             for (GLint i = 0; i < vertex_count; i++)
344             {
345                 if (min_fragment_value > fragment[i])
346                 {
347                     min_fragment_value = fragment[i];
348                 }
349                 if (max_fragment_value < fragment[i])
350                 {
351                     max_fragment_value = fragment[i];
352                 }
353             }
354             #pragma omp parallel for
355             for (GLint i = 0; i < vertex_count; i++)
356             {
357                 (*result)._color[i] = coldToHotColormap((GLfloat)fragment[i],
358                                                         (GLfloat)min_fragment_value,
359                                                         (GLfloat)max_fragment_value);
360             }
361         }
362         return result;
363     }
364     // generates either u- or v-directional isoparametric lines of the given tensor product surface
365     RowMatrix<SP<GenericCurve3>::Default>*
366         TensorProductSurface3::generateIsoparametricLines(
367                 variable::Type type, GLint iso_line_count,
368                 GLint maximum_order_of_derivatives, GLint div_point_count,
369                 GLenum usage_flag) const
370     {
371         if (iso_line_count < 2 || maximum_order_of_derivatives < 0 ||
372             div_point_count < 2)
```


© Ágoston Róth, 2018

```
373         {
374             return nullptr;
375         }
376         if (usage_flag != GL_STREAM_DRAW  && usage_flag != GL_STREAM_READ  &&
377             usage_flag != GL_STREAM_COPY  &&
378             usage_flag != GL_DYNAMIC_DRAW && usage_flag != GL_DYNAMIC_READ &&
379             usage_flag != GL_DYNAMIC_COPY &&
380             usage_flag != GL_STATIC_DRAW  && usage_flag != GL_STATIC_READ  &&
381             usage_flag != GL_STATIC_COPY)
382         {
383             return nullptr;
384         }
385         RowMatrix<SP<GenericCurve3>::Default> *result =
386             new (nothrow) RowMatrix<SP<GenericCurve3>::Default>(iso_line_count);
387         if (!result)
388         {
389             return nullptr;
390         }
391         GLdouble iso_line_step = (_max_value[1-type] - _min_value[1-type]) /
392                                  (iso_line_count - 1);
393         GLdouble div_step = (_max_value[type] - _min_value[type]) /
394                             (div_point_count - 1);
395         for (GLint l = 0; l < iso_line_count; l++)
396         {
397             (*result)[l] = SP<GenericCurve3>::Default(new (nothrow) GenericCurve3(
398                     maximum_order_of_derivatives, div_point_count, usage_flag));
399             if (!(*result)[l])
400             {
401                 delete result, result = nullptr;
402                 return nullptr;
403             }
404             GenericCurve3 &iso_line_l = *(*result)[l];
405             GLdouble iso_value = min(_min_value[1-type] + l * iso_line_step,
406                                      _max_value[1-type]);
407             // error flag
408             GLboolean aborted = GL_FALSE;
409             #pragma omp parallel for
410             for (GLint k = 0; k < div_point_count; k++)
411             {
412                 #pragma omp flush (aborted)
413                 if (!aborted)
414                 {
415                     GLdouble div_value = min(_min_value[type] + k * div_step,
416                                              _max_value[type]);
417                     DirectionalDerivatives d;
418                     if (!calculateDirectionalDerivatives(
419                             type, maximum_order_of_derivatives,
420                             (type == variable::V ? iso_value : div_value),
421                             (type == variable::U ? iso_value : div_value), d))
422                     {
423                         aborted = GL_TRUE;
424                         #pragma omp flush (aborted)
425                     }
426                     else
427                     {
428                         iso_line_l._derivative.setColumn(k, d);
429                     }
430                 }
431             }
432             if (aborted)
433             {
```


© Ágoston Róth, 2018

```
434                 delete result, result = nullptr;
435                 return nullptr;
436             }
437         }

438         return result;
439     }

440     // Tries to solve the surface interpolation problem s(u_i, v_j) = d_{i,j}, i = 0,1,...,n, j = 0,1,...,m,
441     // where [u_i]_{i=0}^n ⊂ [u_min, u_max] and [v_j]_{j=0}^m ⊂ [v_min, v_max] are two strictly increasing knot vectors,
442     // while [d_{i,j}]_{i=0, j=0}^{n, m} denotes the grid of data points that has to be interpolated.
443     // In case of success, the solution will be stored in the member field _data that corresponds to [p_{i,j}]_{i=0, j=0}^{n,m}.
444     GLboolean TensorProductSurface3::updateDataForInterpolation(
445             const RowMatrix<GLdouble> &u_knot_vector,
446             const ColumnMatrix<GLdouble> &v_knot_vector,
447             Matrix<Cartesian3> &data_points_to_interpolate)
448     {
449         GLint row_count = _data.rowCount();
450         if (!row_count)
451         {
452             return GL_FALSE;
453         }

454         GLint column_count = _data.columnCount();
455         if (!column_count)
456         {
457             return GL_FALSE;
458         }

459         if (u_knot_vector.columnCount() != row_count ||
460             v_knot_vector.rowCount() != column_count ||
461             data_points_to_interpolate.rowCount() != row_count ||
462             data_points_to_interpolate.columnCount() != column_count)
463         {
464             return GL_FALSE;
465         }

466         // 1: calculate the u-collocation matrix
467         GLboolean   u_collocation_matrix_construction_aborted = GL_FALSE;
468         RealMatrix u_collocation_matrix(row_count, row_count);

469         #pragma omp parallel for
470         for (GLint i = 0; i < row_count; i++)
471         {
472             #pragma omp flush (u_collocation_matrix_construction_aborted)
473             if (!u_collocation_matrix_construction_aborted)
474             {
475                 RowMatrix<GLdouble> u_blending_values;
476                 if (!blendingFunctionValues(
477                         variable::U, u_knot_vector[i], u_blending_values))
478                 {
479                     u_collocation_matrix_construction_aborted = GL_TRUE;
480                     #pragma omp flush (u_collocation_matrix_construction_aborted)
481                 }
482                 else
483                 {
484                     u_collocation_matrix.setRow(i, u_blending_values);
485                 }
486             }
487         }

488         if (u_collocation_matrix_construction_aborted)
489         {
490             return GL_FALSE;
491         }

492         // 2: calculate the v-collocation matrix and perform PLU-decomposition on it
493         GLboolean   v_collocation_matrix_construction_aborted = GL_FALSE;
494         RealMatrix v_collocation_matrix(column_count, column_count);

495         #pragma omp parallel for
496         for (GLint j = 0; j < column_count; ++j)
497         {
```


© Ágoston Róth, 2018

```
498             #pragma omp flush (v_collocation_matrix_construction_aborted)
499             if (!v_collocation_matrix_construction_aborted)
500             {
501                 RowMatrix<GLdouble> v_blending_values;
502                 if (!blendingFunctionValues(
503                         variable::V, v_knot_vector[j], v_blending_values))
504                 {
505                     v_collocation_matrix_construction_aborted = GL_TRUE;
506                     #pragma omp flush (v_collocation_matrix_construction_aborted)
507                 }
508                 else
509                 {
510                     v_collocation_matrix.setRow(j, v_blending_values);
511                 }
512             }
513         }

514         if (v_collocation_matrix_construction_aborted)
515         {
516             return GL_FALSE;
517         }

518         // 3: perfom PLU-decomposition on the u-collocation matrix and for all fixed j ∈ {0,1,...,m}
519         //    determine intermediate vectors a_k(v_j) = Σ_{ℓ=0}^{m} p_{k,ℓ} b_{m,ℓ}(v_j), k = 0,1,...,n such that
520         //    Σ_{k=0}^{n} a_k(v_j) b_{n,k}(u_i) = d_{i,j} for all i = 0,1,...,n
521         PLUDecomposition    u_PLUD(u_collocation_matrix);
522         Matrix<Cartesian3> a(row_count, column_count);
523         if (!u_PLUD.solveLinearSystem(data_points_to_interpolate, a))
524         {
525             return GL_FALSE;
526         }

527         // 4: perfom PLU-decomposition on the v-collocation matrix and for all fixed i ∈ {0,1,...,n}
528         //    determine control points p_{i,j}, j = 0,1,...,m such that Σ_{ℓ=0}^{m} p_{i,ℓ} b_{m,ℓ}(v_j) = a_i(v_j) for all j = 0,1,...,m
529         PLUDecomposition v_PLUD(v_collocation_matrix);
530         if (!v_PLUD.solveLinearSystem(a, _data, GL_FALSE))
531         {
532             return GL_FALSE;
533         }

534         return GL_TRUE;
535     }

536     // By default, it is assumed that the vectors stored in Matrix<Cartesian3> _data
537     // represent the vertices of a control net. If this is not the case, the next virtual
538     // VBO handling methods can be redeclared and redefined in derived classes.

539     // Deletes the vertex buffer objects of the control net.
540     void TensorProductSurface3::deleteVertexBufferObjectsOfData()
541     {
542         if (_vbo_data)
543         {
544             glDeleteBuffers(1, &_vbo_data);
545             _vbo_data = 0;
546         }
547     }

548     // Control net rendering methods.

549     // The next rendering method will return GL_FALSE if:
550     //  - the given shader program is not active;
551     //  - the user-defined position attribute name cannot be found in the list of active attributes
552     //    of the provided shader program, or exists but is of incorrect type;
553     //  - the u_render_mode or v_render_mode is incorrect (one should use one of the constants GL_LINE_STRIP,
554     //    GL_LINE_LOOP and GL_POINTS).
555     GLboolean TensorProductSurface3::renderData(
556         const ShaderProgram &program,
557         GLenum u_render_mode, GLenum v_render_mode) const
558     {
559         GLint current_program;
560         glGetIntegerv(GL_CURRENT_PROGRAM, &current_program);

561         if (current_program != (GLint)program._id)
562         {
```


© Ágoston Róth, 2018

```
563             return GL_FALSE;
564         }
565         GLint position = program.positionAttributeLocation();
566         if (!_vbo_data || position < 0)
567         {
568             return GL_FALSE;
569         }
570         if (u_render_mode != GL_LINE_STRIP && u_render_mode != GL_LINE_LOOP &&
571             u_render_mode != GL_POINTS)
572         {
573             return GL_FALSE;
574         }
575         if (v_render_mode != GL_LINE_STRIP && v_render_mode != GL_LINE_LOOP &&
576             v_render_mode != GL_POINTS)
577         {
578             return GL_FALSE;
579         }
580         if (u_render_mode == GL_LINE_STRIP && _closed[variable::U])
581         {
582             u_render_mode = GL_LINE_LOOP;
583         }
584         if (v_render_mode == GL_LINE_STRIP && _closed[variable::V])
585         {
586             v_render_mode = GL_LINE_LOOP;
587         }
588         if (u_render_mode == GL_LINE_LOOP && !_closed[variable::U])
589         {
590             u_render_mode = GL_LINE_STRIP;
591         }
592         if (v_render_mode == GL_LINE_LOOP && !_closed[variable::V])
593         {
594             v_render_mode = GL_LINE_STRIP;
595         }
596         GLint row_count = _data.rowCount();
597         if (!row_count)
598         {
599             return GL_FALSE;
600         }
601         GLint column_count = _data.columnCount();
602         if (!column_count)
603         {
604             return GL_FALSE;
605         }
606         GLsizei size = row_count * column_count;
607         glEnableVertexAttribArray(position);
608             glBindBuffer(GL_ARRAY_BUFFER, _vbo_data);
609             glVertexAttribPointer(position, 3, GL_FLOAT, GL_FALSE, 0, nullptr);
610             if (u_render_mode == GL_POINTS && v_render_mode == GL_POINTS)
611             {
612                 glDrawArrays(u_render_mode, 0, size);
613             }
614             else
615             {
616                 GLint offset = 0;
617                 for (GLint i = 0; i < row_count; i++, offset += column_count)
618                 {
```


© Ágoston Róth, 2018

```
619                     glDrawArrays(u_render_mode, offset, column_count);
620                 }
621                 for (GLint j = 0; j < column_count; j++, offset += row_count)
622                 {
623                     glDrawArrays(v_render_mode, offset, row_count);
624                 }
625             }
626             glBindBuffer(GL_ARRAY_BUFFER, 0);
627         glDisableVertexAttribArray(position);
628         glVertexAttrib3f(position, 0.0f, 0.0f, 0.0f);
629         return GL_TRUE;
630     }
631     // If during rendering one intends to use shader program objects that are not instances of
632     // our class ShaderProgram, then one should specify an attribute location associated with
633     // positions of type vec3.
634     //
635     // The next rendering method will return GL_FALSE if:
636     //  - there is no active shader program object;
637     //  - the given position location either cannot be found in the list of active attribute locations,
638     //    or exists but is of incorrect type;
639     //  - the u_render_mode or v_render_mode is incorrect (one should use one of the constants GL_LINE_STRIP,
640     //    GL_LINE_LOOP and GL_POINTS).
641     GLboolean TensorProductSurface3::renderData(
642             GLenum u_render_mode, GLenum v_render_mode,
643             GLint vec3_position_location) const
644     {
645         if (!_vbo_data || vec3_position_location < 0)
646         {
647             return GL_FALSE;
648         }
649         if (u_render_mode != GL_LINE_STRIP && u_render_mode != GL_LINE_LOOP &&
650             u_render_mode != GL_POINTS)
651         {
652             return GL_FALSE;
653         }
654         if (v_render_mode != GL_LINE_STRIP && v_render_mode != GL_LINE_LOOP &&
655             v_render_mode != GL_POINTS)
656         {
657             return GL_FALSE;
658         }
659         if (u_render_mode == GL_LINE_STRIP && _closed[variable::U])
660         {
661             u_render_mode = GL_LINE_LOOP;
662         }
663         if (v_render_mode == GL_LINE_STRIP && _closed[variable::V])
664         {
665             v_render_mode = GL_LINE_LOOP;
666         }
667         if (u_render_mode == GL_LINE_LOOP && !_closed[variable::U])
668         {
669             u_render_mode = GL_LINE_STRIP;
670         }
671         if (v_render_mode == GL_LINE_LOOP && !_closed[variable::V])
672         {
673             v_render_mode = GL_LINE_STRIP;
674         }
675         GLint row_count = _data.rowCount();
676         if (!row_count)
677         {
678             return GL_FALSE;
679         }
```


© Ágoston Róth, 2018

```
680         GLint column_count = _data.columnCount();
681         if (!column_count)
682         {
683             return GL_FALSE;
684         }
685         GLint current_program = 0;
686         glGetIntegerv(GL_CURRENT_PROGRAM, &current_program);
687         if (!current_program)
688         {
689             return GL_FALSE;
690         }
691         GLint attribute_count;
692         glGetProgramiv(current_program, GL_ACTIVE_ATTRIBUTES, &attribute_count);
693         if (!attribute_count)
694         {
695             return GL_FALSE;
696         }
697         GLsizei max_attribute_name_length;
698         glGetProgramiv(current_program, GL_ACTIVE_ATTRIBUTE_MAX_LENGTH,
699                        &max_attribute_name_length);
700         GLchar *attribute_name_data = new GLchar[max_attribute_name_length];
701         GLboolean given_location_exists_and_is_of_type_vec3 = GL_FALSE;
702         for (GLint attribute = 0;
703              attribute < attribute_count && !given_location_exists_and_is_of_type_vec3;
704              attribute++)
705         {
706             GLsizei actual_length = 0;
707             GLint   array_size    = 0;
708             GLenum  type          = 0;
709             glGetActiveAttrib(current_program, attribute, max_attribute_name_length,
710                               &actual_length, &array_size, &type, attribute_name_data);
711             string name(&attribute_name_data[0], actual_length);
712             GLint  location = glGetAttribLocation(current_program, name.c_str());
713             if (type == GL_FLOAT_VEC3 && vec3_position_location == location)
714             {
715                 given_location_exists_and_is_of_type_vec3 = GL_TRUE;
716             }
717         }
718         delete[] attribute_name_data;
719         if (!given_location_exists_and_is_of_type_vec3)
720         {
721             return GL_FALSE;
722         }
723         GLsizei size = row_count * column_count;
724         glEnableVertexAttribArray(vec3_position_location);
725             glBindBuffer(GL_ARRAY_BUFFER, _vbo_data);
726             glVertexAttribPointer(vec3_position_location, 3,
727                                   GL_FLOAT, GL_FALSE, 0, nullptr);
728             if (u_render_mode == GL_POINTS && v_render_mode == GL_POINTS)
729             {
730                 glDrawArrays(u_render_mode, 0, size);
731             }
732             else
733             {
734                 GLint offset = 0;
```


© Ágoston Róth, 2018

```
735                 for (GLint i = 0; i < row_count; i++, offset += column_count)
736                 {
737                     glDrawArrays(u_render_mode, offset, column_count);
738                 }
739                 for (GLint j = 0; j < column_count; j++, offset += row_count)
740                 {
741                     glDrawArrays(v_render_mode, offset, row_count);
742                 }
743             }
744             glBindBuffer(GL_ARRAY_BUFFER, 0);
745         glDisableVertexAttribArray(vec3_position_location);
746         glVertexAttrib3f(vec3_position_location, 0.0f, 0.0f, 0.0f);
747         return GL_TRUE;
748     }
749     // Updates the vertex buffer objects of the control net.
750     GLboolean TensorProductSurface3::updateVertexBufferObjectsOfData(GLenum usage_flag)
751     {
752         if (usage_flag != GL_STREAM_DRAW  && usage_flag != GL_STREAM_READ  &&
753             usage_flag != GL_STREAM_COPY  &&
754             usage_flag != GL_DYNAMIC_DRAW && usage_flag != GL_DYNAMIC_READ &&
755             usage_flag != GL_DYNAMIC_COPY &&
756             usage_flag != GL_STATIC_DRAW  && usage_flag != GL_STATIC_READ  &&
757             usage_flag != GL_STATIC_COPY)
758         {
759             return GL_FALSE;
760         }
761         if (!_data.rowCount() || !_data.columnCount())
762         {
763             return GL_FALSE;
764         }
765         deleteVertexBufferObjectsOfData();
766         glGenBuffers(1, &_vbo_data);
767         if (!_vbo_data)
768         {
769             return GL_FALSE;
770         }
771         GLsizeiptr vertex_byte_size =
772             6 * (GLsizeiptr)(_data.rowCount() * _data.columnCount() * sizeof(GLfloat));
773         glBindBuffer(GL_ARRAY_BUFFER, _vbo_data);
774         glBufferData(GL_ARRAY_BUFFER, vertex_byte_size, nullptr, usage_flag);
775         GLfloat *vertex_coordinate = (GLfloat*)glMapBuffer(GL_ARRAY_BUFFER,
776                                                            GL_WRITE_ONLY);
777         for (GLint i = 0; i < _data.rowCount(); i++)
778         {
779             for (GLint j = 0; j < _data.columnCount(); j++)
780             {
781                 for (GLint c = 0; c < 3; c++, vertex_coordinate++)
782                 {
783                     *vertex_coordinate = (GLfloat)_data(i, j)[c];
784                 }
785             }
786         }
787         for (GLint j = 0; j < _data.columnCount(); j++)
788         {
789             for (GLint i = 0; i < _data.rowCount(); i++)
790             {
791                 for (GLint c = 0; c < 3; c++, vertex_coordinate++)
792                 {
793                     *vertex_coordinate = (GLfloat)_data(i, j)[c];
```


© Ágoston Róth, 2018

```
794                     }
795                 }
796             }
797             glBindBuffer(GL_ARRAY_BUFFER, _vbo_data);
798             if (!glUnmapBuffer(GL_ARRAY_BUFFER))
799             {
800                 return GL_FALSE;
801             }
802             return GL_TRUE;
803         }
804         // get the row count of the data grid
805         GLint TensorProductSurface3::rowCount() const
806         {
807             return _data.rowCount();
808         }
809         // get the column count of the data grid
810         GLint TensorProductSurface3::columnCount() const
811         {
812             return _data.columnCount();
813         }
814         // destructor
815         TensorProductSurface3::~TensorProductSurface3()
816         {
817             deleteVertexBufferObjectsOfData();
818         }
819 }
```

2.15 Characteristic polynomials

Characteristic polynomials of type (1.10/2) will be instances of the class CharacteristicPolynomial that is able to store and update the factorization of (1.10/2) and also provides an overloaded function operator for evaluation purposes. The diagram of the class is illustrated in Fig. 2.24/211, while its definition and implementation can be found in Listings 2.46/211 and 2.47/213, respectively.

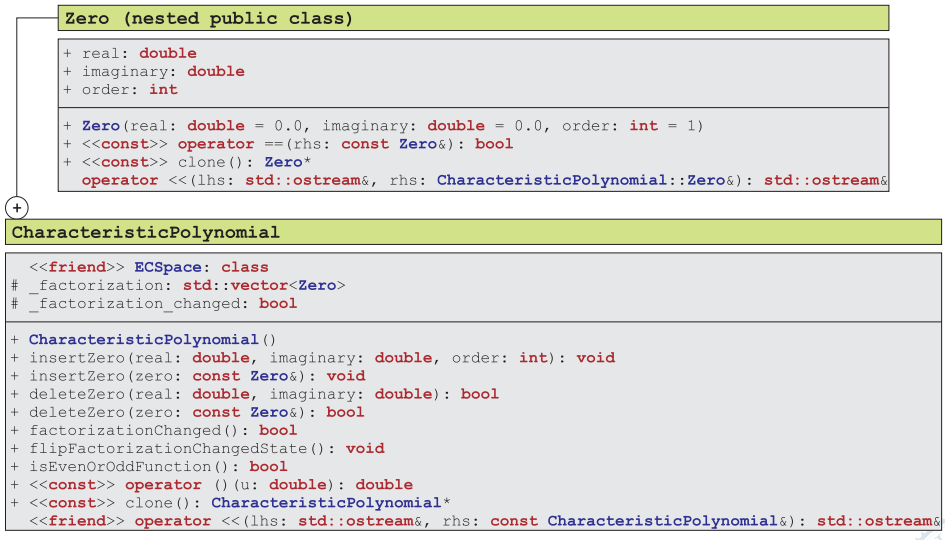


Fig. 2.24: Class diagrams of characteristic polynomials and of their (higher order) zeros

© Ágoston Róth, 2018

Listing 2.46. Characteristic polynomials (**EC/CharacteristicPolynomials.h**)

```
1  #ifndef CHARACTERISTICPOLYNOMIALS_H
2  #define CHARACTERISTICPOLYNOMIALS_H

3  #include <iostream>
4  #include <vector>

5  namespace cagd
6  {
7      // represents the characteristic polynomial (1.10/2) of the constant-coefficient linear homogeneous differential
8      // equation (1.9/2)
9      class CharacteristicPolynomial
10     {
11         // friend class that represents an extended Chebyshev (EC) vector space of functions
12         friend class ECSpace;

13         // friend output to stream operator
14         friend std::ostream& operator <<(std::ostream &lhs,
15                                          const CharacteristicPolynomial &rhs);

16     public:
17         // the nested public class Zero represents a higher order complex root of the characteristic polynomial (1.10/2)
18         class Zero
19         {
20         public:
21             double  real;       // the real part of the complex root
22             double  imaginary;  // the imaginary part of the complex root
23             int     order;      // the multiplicity or the order of the complex root, its value does
24                                 // not affect the comparison operator == declared below in line 28/212

25             // special/default constructor
26             explicit Zero(double real = 0.0, double imaginary = 0.0, int order = 1);

27             // binary logical operator for possible comparisons (conjugate roots are considered to be equivalent)
28             bool operator ==(const Zero &rhs) const;

29             // clone function required by smart pointers based on the deep copy ownership policy
30             Zero* clone() const;
31         };

32     protected:
33         // the factorization of the characteristic polynomial (1.10/2) (we only store conjugately pairwise different zeros)
34         std::vector<Zero> _factorization;

35         // a flag that will be set to true whenever the factorization of the characteristic polynomial (1.10/2) or the
36         // definition domain of the underlying EC space is changed
37         bool              _factorization_changed;

38     public:
39         // default constructor
40         CharacteristicPolynomial();

41         // If during insertion the given zero (or its conjugate) already exists, the value of its order will be changed
42         // to the maximum of its formerly and newly given orders.
43         void insertZero(double real, double imaginary, int order);
44         void insertZero(const Zero &zero);

45         // If exists, deletes the given zero (or its conjugate) independently of its order.
46         bool deleteZero(double real, double imaginary);
47         bool deleteZero(const Zero &zero);

48         bool factorizationChanged() const;     // returns the state of the flag _factorization_changed
49         void flipFactorizationChangedState(); // flips the state of the flag _factorization_changed

50         // overloaded function operator that evaluates the characteristic polynomial (1.10/2) at the parameter value u
51         double operator ()(double u) const;

52         // determines whether the characteristic polynomial is an either odd or even function
53         bool isEvenOrOddFunction() const;

54         // clone function required by smart pointers based on the deep copy ownership policy
55         CharacteristicPolynomial* clone() const;
```


© Ágoston Róth, 2018

```
56      };

57      // overloaded output to stream operators
58      std::ostream& operator <<(std::ostream &lhs,
59                                const CharacteristicPolynomial::Zero &rhs);
60      std::ostream& operator <<(std::ostream &lhs, const CharacteristicPolynomial &rhs);
61  }

62  #endif // CHARACTERISTICPOLYNOMIALS_H
```

Listing 2.47. Characteristic polynomials (**EC/CharacteristicPolynomials.cpp**)

```
1   #include "CharacteristicPolynomials.h"
2   #include "../Core/Math/Constants.h"
3   #include <cmath>
4   #include <algorithm>

5   using namespace std;

6   namespace cagd
7   {
8       // special/default constructor
9       CharacteristicPolynomial::Zero::Zero(double real, double imaginary, int order):
10              real(real), imaginary(imaginary), order(order >= 1 ? order : 1)
11      {
12          assert("The multiplicity (or the order) of the given complex zero should be "
13                 "at least 1!" && order >= 1);
14      }

15      // binary logical operator for possible comparisons (conjugate roots are considered to be equivalent)
16      bool CharacteristicPolynomial::Zero::operator ==(
17              const CharacteristicPolynomial::Zero &rhs) const
18      {
19          return ((real == rhs.real) && (abs(imaginary) == abs(rhs.imaginary)));
20      }

21      // clone function required by smart pointers based on the deep copy ownership policy
22      CharacteristicPolynomial::Zero* CharacteristicPolynomial::Zero::clone() const
23      {
24          return new (nothrow) Zero(*this);
25      }

26      // default constructor
27      CharacteristicPolynomial::CharacteristicPolynomial():
28              _factorization(0), _factorization_changed(false)
29      {
30      }

31      // If during insertion the given zero (or its conjugate) already exists, the value of its order will be changed
32      // to the maximum of its formerly and newly given orders.
33      void CharacteristicPolynomial::insertZero(double real, double imaginary, int order)
34      {
35          Zero z(real, imaginary, order);

36          vector<Zero>::iterator i = find(_factorization.begin(), _factorization.end(), z);

37          if (i == _factorization.end())
38          {
39              _factorization.push_back(z);
40              _factorization_changed = true;
41          }
42          else
43          {
44              if (i->order < order)
45              {
46                  i->order = order;
47                  _factorization_changed = true;
48              }
49              else
50              {
51                  _factorization_changed = false;
52              }
```


© Ágoston Róth, 2018

```
53          }
54      }
55      void CharacteristicPolynomial::insertZero(const Zero &zero)
56      {
57          vector<Zero>::iterator i = find(_factorization.begin(), _factorization.end(),
58                                          zero);
59          if (i == _factorization.end())
60          {
61              _factorization.push_back(zero);
62              _factorization_changed = true;
63          }
64          else
65          {
66              if (i->order < zero.order)
67              {
68                  i->order = zero.order;
69                  _factorization_changed = true;
70              }
71              else
72              {
73                  _factorization_changed = false;
74              }
75          }
76      }
77      // If exists, deletes the given zero (or its conjugate) independently of its order.
78      bool CharacteristicPolynomial::deleteZero(double real, double imaginary)
79      {
80          Zero z(real, imaginary);
81          vector<Zero>::iterator i = find(_factorization.begin(), _factorization.end(), z);
82          if (i == _factorization.end())
83          {
84              _factorization_changed = false;
85              return false;
86          }
87          _factorization.erase(i);
88          _factorization_changed = true;
89          return true;
90      }
91      bool CharacteristicPolynomial::deleteZero(const Zero &zero)
92      {
93          vector<Zero>::iterator i = find(_factorization.begin(), _factorization.end(),
94                                          zero);
95          if (i == _factorization.end())
96          {
97              _factorization_changed = false;
98              return false;
99          }
100         _factorization.erase(i);
101         _factorization_changed = true;
102         return true;
103     }
104     // returns the state of the flag _factorization_changed
105     bool CharacteristicPolynomial::factorizationChanged() const
106     {
107         return _factorization_changed;
108     }
109     // flips the state of the flag _factorization_changed
```


© Ágoston Róth, 2018

```
110     void CharacteristicPolynomial::flipFactorizationChangedState()
111     {
112         _factorization_changed = !_factorization_changed;
113     }

114     // overloaded function operator that evaluates the characteristic polynomial (1.10/2) at the parameter value u
115     double CharacteristicPolynomial::operator ()(double u) const
116     {
117         double product = 1.0;
118         for (vector<Zero>::const_iterator z = _factorization.begin();
119              z != _factorization.end(); z++)
120         {
121             if (z->imaginary)
122             {
123                 product *= pow(z->real * z->real + z->imaginary * z->imaginary
124                                - 2.0 * z->real * u + u * u, z->order);
125             }
126             else
127             {
128                 product *= pow(u - z->real, z->order);
129             }
130         }

131         return product;
132     }

133     // determines whether the characteristic polynomial is an either odd or even function
134     bool CharacteristicPolynomial::isEvenOrOddFunction() const
135     {
136         if (_factorization.empty())
137         {
138             return false;
139         }

140         for (vector<Zero>::const_iterator z = _factorization.begin();
141              z != _factorization.end(); z++)
142         {
143             if (z->real != 0.0 && z->imaginary != 0.0)
144             {
145                 return false;
146             }

147             if (z->imaginary == 0.0)
148             {
149                 CharacteristicPolynomial::Zero symmetric(-z->real, 0.0);
150                 vector<Zero>::const_iterator s = find(_factorization.begin(),
151                                                       _factorization.end(), symmetric);

152                 if (s == _factorization.end())
153                 {
154                     return false;
155                 }
156                 else
157                 {
158                     if (z->order != s->order)
159                     {
160                         return false;
161                     }
162                 }

163             }
164         }
165         return true;
166     }

167     // clone function required by smart pointers based on the deep copy ownership policy
168     CharacteristicPolynomial* CharacteristicPolynomial::clone() const
169     {
170         return new (nothrow) CharacteristicPolynomial(*this);
171     }

172     // overloaded output to stream operators
173     ostream& operator <<(ostream &lhs, const CharacteristicPolynomial::Zero &rhs)
174     {
```


© Ágoston Róth, 2018

```
175         return lhs << rhs.real << " " << rhs.imaginary << " " << rhs.order;
176     }
177     ostream& operator <<(ostream &lhs, const CharacteristicPolynomial &rhs)
178     {
179         lhs << rhs._factorization.size() << endl;
180         for (vector<CharacteristicPolynomial::Zero>::const_iterator
181              i = rhs._factorization.begin(); i != rhs._factorization.end(); i++)
182         {
183             lhs << *i << endl;
184         }
185         return lhs;
186     }
187 }
```

2.16 EC spaces

EC spaces (that comprise the constants and can be identified with the solution spaces of differential equations of type (1.9/2) defined on intervals for which the corresponding spaces of derivatives are also EC) will be instances of the class ECSpace that:

- is able to generate and to update both the ordinary basis and the normalized B-basis of an EC vector space specified by the factorization of a characteristic polynomial of type (1.10/2);
- provides a function operator to evaluate the zeroth and higher order derivatives of both bases at any point of the definition domain;
- can also be used to generate the general basis transformation matrix formulated in Theorem 1.2/11 that maps the normalized B-basis to the ordinary basis of the underlying EC space;
- is able to decide whether the specified EC vector is reflection invariant;
- can list the LaTeX expressions of the ordinary basis functions;
- can also be used to generate the images of both the ordinary basis and the normalized B-basis functions.

The diagram of the class is illustrated in Fig. 2.25/217, while its definition and implementation can be found in Listings 2.48/216 and 2.49/220, respectively.

Listing 2.48. EC spaces (**EC/ECSpaces.h**)

```
1  #ifndef ECSPACES_H
2  #define ECSPACES_H
3  #include "CharacteristicPolynomials.h"
4  #include "../Core/Math/RealMatrices.h"
5  #include "../Core/SmartPointers/SpecializedSmartPointers.h"
6  #include "../Core/Geometry/Curves/GenericCurves3.h"
7  #include <string>
8  #include <vector>
9  namespace cagd
10 {
11     // The class ECSpace below represents an EC vector space S_n^{α,β} of functions that comprises the constants and can
12     // be identified with the solution space of the constant-coefficient homogeneous linear differential equation (1.9/2)
13     // defined on an interval [α, β] for which the space DS_n^{α,β} of derivatives is also EC.
14     // Note that the underlying EC space is spanned by those ordinary basis functions (1.1/1) that are generated by
15     // the (higher oder) zeros of the characteristic polynomial (1.10/2) associated with the differential equation (1.9/2).
16     // In order to ensure that the underlying EC space also contains the constant functions and also has normalizable
17     // bases, we have to assume that z = 0 is at least a first order zero of the characteristic polynomial (1.10/2).
```


© Ágoston Róth, 2018

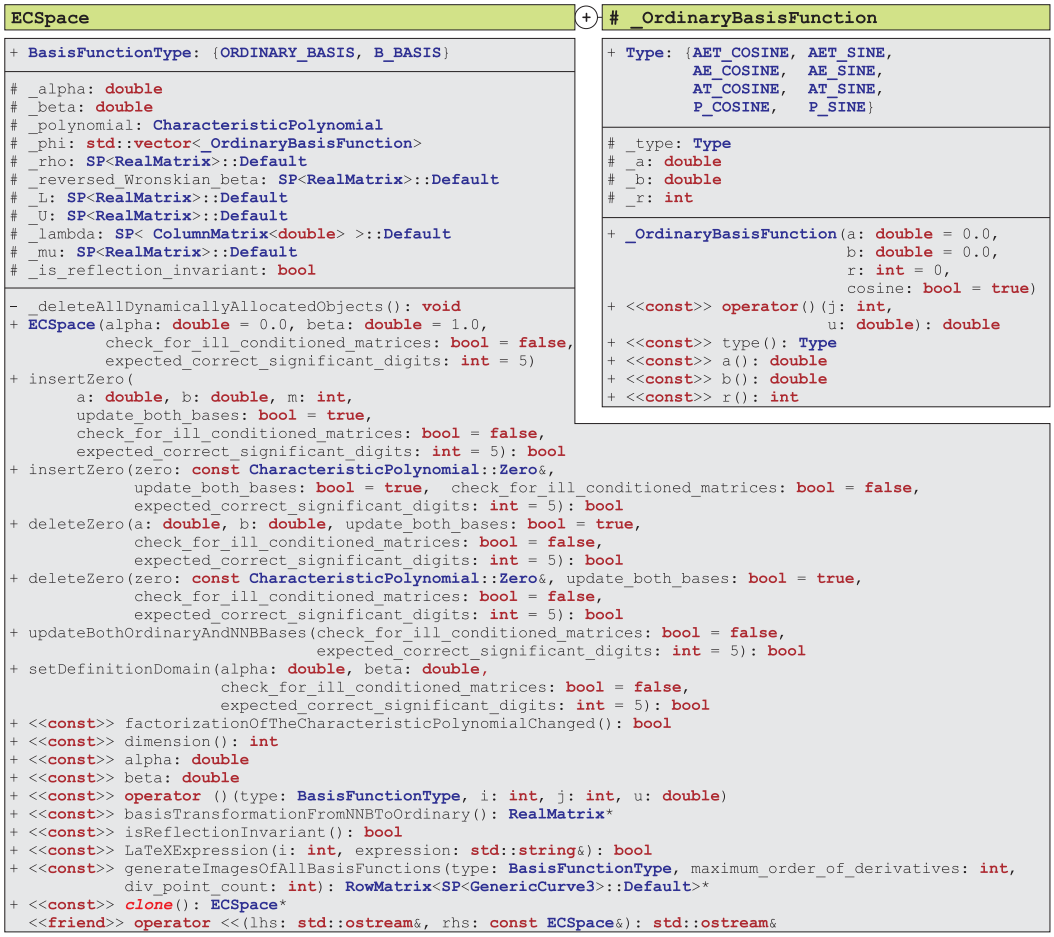


Fig. 2.25: Class diagram of constant-comprising translation invariant EC spaces that can be identified with the solution spaces of constant-coefficient homogeneous linear differential equations defined on intervals for which the corresponding spaces of derivatives are also EC

```
18      class ECSpace
19      {
20          // overloaded friend output to stream operator that lists the conjugately equivalent zeros of the
21          // characteristic polynomial
22          friend std::ostream& operator <<(std::ostream &lhs, const ECSpace &rhs);
23      public:
24          // The class is able to determine both the ordinary basis (1.1/1) and the normalized B-basis (1.4/2) of the
25          // underlying EC vector space of functions and it can also be used to compute their derivatives of arbitrary
26          // order. In order to specify the type of the basis functions that have to be differentiated at a given parameter
27          // value u ∈ [α, β], one has to use the named constants provided by the public enumaration BasisFunctionType.
28          enum BasisFunctionType{ORDINARY_BASIS, B_BASIS};
29      protected:
30          // The protected nested class _OrdinaryBasisFunction below represents an ordinary basis function generated
31          // by a possible higher order root of the given characterteristic polynomial (1.10/2).
32          // Based on the real and imaginary parts of a possible complex root z = a ± ib of order m, we will
33          // differentiate algebraic-exponential-trigonometric (AET), algebraic-exponential (AE), algebraic-
34          // trigonometric (AT) and pure polynomial (P) ordinary basis functions of types (1.21/5), (1.22/6), (1.23/6)
35          // and (1.24/6), respectively.
36          class _OrdinaryBasisFunction
37          {
```


© Ágoston Róth, 2018

```
38          public:
39              // Possible ordinary basis function types can be specified by the following named constants.
40              // Note that enumerators AE_SINE and P_SINE denote the constant zero function.
41              enum Type{AET_COSINE,  AET_SINE,
42                        AE_COSINE,   AE_SINE,
43                        AT_COSINE,   AT_SINE,
44                        P_COSINE,    P_SINE};

45          protected:
46              Type     _type;       // properties of the ordinary basis function u^r · [e^{au}] · {cos(bu)| sin(bu)},
47              double   _a, _b;      // where the arguments of parantheses [] and {} are optional and obligatory,
48              int      _r;          // respectively, and the bar | denotes a possible choice

49          public:
50              // default/special constructors based on a possible complex root z = a ± ib of order m ≥ 1, where
51              // r ∈ {0, ..., m − 1}; if the boolean parameter cosine is true, the generated ordinary basis function
52              // corresponds to u^r · [e^{au}] · cos(bu), otherwise to u^r · [e^{au}] · sin(bu)
53              _OrdinaryBasisFunction(double a = 0.0, double b = 0.0, int r = 0,
54                                     bool cosine = true);

55              // overloaded function operator that evaluates the jth order (j ≥ 0) derivative of the stored ordinary
56              // basis function at the parameter value u ∈ [α, β]
57              double operator ()(int j, double u) const;

58              // getters
59              Type     type() const;
60              double   a() const;
61              double   b() const;
62              int      r() const;
63          };

64          // user-specified endpoints of the definition domain [α, β], where 0 < β − α < ℓ′ (S_n^{α,β})
65          double                                  _alpha, _beta;

66          // The member variable _polynomial represents the characteristic polynomial (1.10/2) associated with the
67          // differential equation (1.9/2). During the maintenance of its factorization (i.e., in case of root insertion
68          // and deletion) conjugate zeros will be considered equivalent. Therefore the variable _polynomial only
69          // stores conjugately pairwise different zeros, and if one tries to insert an already existing root, only its
70          // order (i.e., multiplicity) will be updated to the maximum of its former and newly given values.
71          // An existing root can also be indetified by its conjugately equivalent variant and, if its requested,
72          // it will be deleted independently of its order.
73          CharacteristicPolynomial                _polynomial;

74          // represents the ordinary basis {φ_{n,i}(u) : u ∈ [α, β]}_{i=0}^n of type (1.1/1) that is generated by the (higher
75          // order) zeros of the given characteristic polynomial
76          std::vector<_OrdinaryBasisFunction>     _phi;

77          // By solving the linear system (1.13/3) for all indices i = 0, 1, ..., n, one can build the unique real square
78          // matrix [ρ_{i,k}]_{i=0,k=0}^{n,n} that transforms the ordinary basis functions {φ_{n,i} : u ∈ [α, β]}_{i=0}^n to the particular
79          // integrals (i.e., bicanonical basis) {v_{n,i}(u) : u ∈ [α, β]}_{i=0}^n described by equation (1.12/3), i.e.,
80          // v_{n,i}(u) = Σ_{k=0}^n ρ_{i,k} φ_{n,k}(u), ∀u ∈ [α, β], ∀i = 0, 1, ..., n. The address of the matrix [ρ_{i,k}]_{i=0,k=0}^{n,n} will
81          // be stored in the next smart pointer.
82          SP<RealMatrix>::Default                 _rho;

83          // a smart pointer to the Wronskian W_{[v_{n,n}, v_{n,n−1}, ..., v_{n,0}]}(β) of the reverse ordered system
84          // {v_{n,n−i}(u) : u ∈ [α, β]}_{i=0}^n at the parameter value u = β
85          SP<RealMatrix>::Default                 _reversed_v_Wronskian_beta;

86          // smart pointers to the lower and upper triangular matrices that appear in the Doolittle-type LU
87          // decomposition L · U = W_{[v_{n,n}, v_{n,n−1}, ..., v_{n,0}]}(β)
88          SP<RealMatrix>::Default                 _L, _U;

89          // a smart pointer to the first column of the inverse matrix L^{−1}
90          SP< ColumnMatrix<double> >::Default     _lambda;

91          // a smart pointer that stores – based on the reflection invariance of the underlying EC space – the
92          // address of a matrix that consists of either all or first half of the columns of the inverse matrix U^{−1}
93          SP<RealMatrix>::Default                 _mu;

94          // determines whether the underlying EC space is reflection invariant or not
95          bool                                    _is_reflection_invariant;
```


© Ágoston Róth, 2018

```
96      private:
97          // Our smart pointers automatically delete the dynamically allocated objects referenced by them when they
98          // go out of scope. However, it may happen that we need to manually delete the referenced objects and to
99          // re-initialize the smart pointers to their default null values in order to highlight that some mathematical
100         // operation could not be performed successfully. For such special cases we will use the next private method.
101         void _deleteAllDynamicallyAllocatedObjects();
102
103     public:
104         // Remark
105         //
106         // Several methods of this class expect a boolean flag named check_for_ill_conditioned_matrices and a
107         // non-negative integer named expected_correct_significant_digits.
108         //
109         // If the flag check_for_ill_conditioned_matrices is set to true, the method will calculate the condition number
110         // of each matrix that appears in the construction of the unique normalized B-basis of the underlying EC space.
111         // Using singular value decomposition, each condition number is determined as the ratio of the largest
112         // and smallest smallest singular values of the corresponding matrices.
113         //
114         // If at least one of the obtained condition numbers is too large, i.e., when the number of estimated correct
115         // significant digits is less than number of expected ones, the method will throw an exception that states
116         // that one of the systems of linear equations is ill-conditioned and therefore its solution may be not accurate.
117         //
118         // If the user catches such an exception, one can try:
119         //   1) to lower the number of expected correct significant digits;
120         //   2) to decrease the dimension of the underlying EC space;
121         //   3) to change the endpoints of the definition domain [α, β];
122         //   4) to run the code without testing for ill-conditioned matrices and hope for the best.
123         //
124         // Note that the standard condition number may lead to an overly pessimistic estimate for the overall error
125         // and, by activating this boolean flag, the run-time of these methods will increase. Several numerical tests
126         // show that ill-conditioned matrices appear either when one defines EC spaces with relatively big dimensions,
127         // or when the endpoints of the definition domain are poorly chosen. Considering that, in practice, curves and
128         // surfaces are mostly composed of smoothly joined lower order arcs and patches, by default we opted for speed,
129         // i.e., initially the flag check_for_ill_conditioned_matrices is set to false. Naturally, if one obtains mathematically
130         // or geometrically unexpected results, then one should (also) study the condition numbers mentioned above.
131
132         // default/special constructor
133         ECSpace(double alpha = 0.0, double beta = 1.0,
134                 bool check_for_ill_conditioned_matrices = false,
135                 int expected_correct_significant_digits = 5);
136
137         // Inserts an mth order complex root of the form z = a±ib into the factorization of the characteristic polynomial.
138         bool insertZero(
139                 double a, double b, int m,
140                 bool update_both_bases = true,
141                 bool check_for_ill_conditioned_matrices = false,
142                 int expected_correct_significant_digits = 5);
143
144         bool insertZero(
145                 const CharacteristicPolynomial::Zero &zero,
146                 bool update_both_bases = true,
147                 bool check_for_ill_conditioned_matrices = false,
148                 int expected_correct_significant_digits = 5);
149
150         // If exists, deletes a complex root of the form z = a ± ib independently of its order. Note that the
151         // root z = 0 will never be deleted since its existence its critical.
152         bool deleteZero(
153                 double a, double b,
154                 bool update_both_bases = true,
155                 bool check_for_ill_conditioned_matrices = false,
156                 int expected_correct_significant_digits = 5);
157
158         bool deleteZero(
159                 const CharacteristicPolynomial::Zero &zero,
160                 bool update_both_bases = true,
161                 bool check_for_ill_conditioned_matrices = false,
162                 int expected_correct_significant_digits = 5);
163
164         // Generates/updates all ordinary basis functions {φ_{n,i}(u) : u ∈ [α, β]}_{i=0}^n and also calculates all
165         // information necessary for the evaluation and differentiation of all normalized B-basis functions
166         // {b_{n,i}(u) : u ∈ [α, β]}_{i=0}^n.
167         bool updateBothBases(
168                 bool check_for_ill_conditioned_matrices = false,
```


© Ágoston Róth, 2018

```
                int expected_correct_significant_digits = 5);

        // Returns the dimension of the underlying EC vector space of functions.
        int dimension() const;

        // Returns the starting point of the definition domain.
        double alpha() const;

        // Returns the ending point of the definition domain.
        double beta() const;

        // Overloaded function operator that evaluates the jth order (j ≥ 0) derivative either of the ith ordinary basis
        // function or of the normalized B-basis function at the parameter value u ∈ [α, β].
        double operator ()(BasisFunctionType type, int i, int j, double u) const;

        // Calculates the entries of the matrix [t^n_{i,j}]^{n,n}_{i=0,j=0} that appears in the general basis transformation (1.45/11)
        // that maps the normalized B-basis {b_{n,i}(u) : u ∈ [α, β]}^n_{i=0} of the underlying EC vector space of functions
        // to its ordinary basis {φ_{n,i}(u) : u ∈ [α, β]}^n_{i=0}.
        RealMatrix* basisTransformationFromNBToOrdinary() const;

        // Determines whether the specified EC vector space is reflection invariant.
        bool isReflectionInvariant() const;

        // If possible, generates the LaTeX expression of the ith ordinary basis functions.
        bool LaTeXExpression(int i, std::string &expression) const;

        // Sets the endpoints of the definition domain (in case of modifications the normalized B-basis has to be
        // udated).
        bool setDefinitionDomain(double alpha, double beta,
                                 bool check_for_ill_conditioned_matrices = false,
                                 int expected_correct_significant_digits = 5);

        // Returns whether the ordninary basis and the normalized B-basis have to be updated.
        bool factorizationOfTheCharacteristicPolynomialChanged() const;

        // Generates images of all ordinary or normalized B-basis functions.
        RowMatrix<SP<GenericCurve3>::Default>* generateImagesOfAllBasisFunctions(
                BasisFunctionType type,
                int maximum_order_of_derivatives, int div_point_count) const;

        // clone function required by smart pointers based on the deep copy ownership policy
        virtual ECSpace* clone() const;

        // destructor
        virtual ~ECSpace();
    };

    // overloaded output to stream operator
    std::ostream& operator <<(std::ostream &lhs, const ECSpace &rhs);
}

#endif // ECSPACES_H
```

Listing 2.49. EC spaces (**EC/ECSpaces.cpp**)

```
#include "ECSpaces.h"

#include "../Core/Exceptions.h"
#include "../Core/Math/Constants.h"
#include "../Core/Math/Matrices.h"
#include "../Core/Math/RealMatrixDecompositions.h"

#include "../Core/Utilities.h"

#include <algorithm>
#include <cassert>
#include <cmath>
#include <cstdlib>
#include <new>

using namespace std;
```


© Ágoston Róth, 2018

```
13 namespace cagd
14 {
15     // default/special constructors based on a possible complex root z = a ± ib of order m ≥ 1, where
16     // r ∈ {0, ..., m − 1}; if the boolean parameter cosine is true, the generated ordinary basis function
17     // corresponds to u^r · [e^au] · cos(bu), otherwise to u^r · [e^au] · sin(bu)
18     ECSpace::_OrdinaryBasisFunction::_OrdinaryBasisFunction(
19             double a, double b, int r, bool cosine):
20             _type(cosine ? AET_COSINE : AET_SINE),
21             _a(a), _b(b),
22             _r(r)
23     {
24         if (_a != 0.0 && _b == 0.0)
25         {
26             if (cosine)
27             {
28                 _type = AE_COSINE;
29             }
30             else
31             {
32                 _type = AE_SINE; // Note that the enumerator AE_SINE denotes the constant zero function.
33             }
34         }

35         if (_a == 0.0 && _b != 0.0)
36         {
37             if (cosine)
38             {
39                 _type = AT_COSINE;
40             }
41             else
42             {
43                 _type = AT_SINE;
44             }
45         }

46         if (_a == 0.0 && _b == 0.0)
47         {
48             if (cosine)
49             {
50                 _type = P_COSINE;
51             }
52             else
53             {
54                 _type = P_SINE; // Note that the enumeratorr P_SINE denotes the constant zero function.
55             }
56         }
57     }

58     // overloaded function operator that evaluates the jth order (j ≥ 0) derivative of the stored ordinary
59     // basis function at the parameter value u ∈ [α, β]
60     double ECSpace::_OrdinaryBasisFunction::operator ()(int j, double u) const
61     {
62         double result = 0.0;

63         if (j == 0) // zeroth order derivatives of ordinary basis functions
64         {
65             switch (_type)
66             {
67             case AET_COSINE:
68                 result = pow(u, _r) * exp(_a * u) * cos(_b * u);
69                 break;

70             case AET_SINE:
71                 result = pow(u, _r) * exp(_a * u) * sin(_b * u);
72                 break;

73             case AE_COSINE:
74                 result = pow(u, _r) * exp(_a * u);
75                 break;

76             case AE_SINE:
77                 result = 0.0;
78                 break;
```


© Ágoston Róth, 2018

```
79              case AT_COSINE:
80                  result = pow(u, _r) * cos(_b * u);
81                  break;
82              case AT_SINE:
83                  result = pow(u, _r) * sin(_b * u);
84                  break;
85              case P_COSINE:
86                  result = pow(u, _r);
87                  break;
88              case P_SINE:
89                  result = 0.0;
90                  break;
91              }
92          }
93          else          // higher order derivatives of ordinary basis functions
94          {
95              switch (_type)
96              {
97              case AET_COSINE:
98                  {
99                      if (!_r)
100                     {
101                         // applying the Leibniz rule for the evaluation of the derivative d^j/du^j e^{au} cos(bu)
102                         double derivative = 0.0;
103
103                         double exp_a_u    = exp(_a * u);
104                         double a_power    = 1.0;
105                         double b_power    = pow(_b, j);
106                         int    b_exponent = j;
107                         double cb         = cos(_b * u);
108                         double sb         = sin(_b * u);
109                         for (int k = 0; k <= j; k++)
110                         {
111                             double aux = 1.0;
112                             switch (b_exponent % 4)
113                             {
114                             case 0: aux =  cb; break;
115                             case 1: aux = -sb; break;
116                             case 2: aux = -cb; break;
117                             case 3: aux =  sb; break;
118                             }
119                             derivative += BC(j, k) * a_power * b_power * aux;
120                             a_power    *= _a;
121                             b_power    /= _b;
122                             b_exponent --;
123                         }
124                         derivative *= exp_a_u;
125                         result = derivative;
126                     }
127                     else
128                     {
129                         // applying the Leibniz rule for the evaluation of the derivative d^j/du^j u^r e^{au} cos(bu)
130                         double derivative = 0.0;
131                         double u_coefficient = 1.0;
132                         double u_power       = pow(u, _r);
133                         int    u_exponent    = _r;
134                         double cb            = cos(_b * u);
135                         double sb            = sin(_b * u);
```


© Ágoston Róth, 2018

```
136                         for (int l = 0; l <= min(j, _r); l++)
137                         {
138                             // applying the Leibniz rule for the evaluation of the derivative d^{j-l}/du^{j-l} e^{au} cos(bu)
139                             double exp_au_cos_bu_derivative = 0.0;
140                             {
141                                 double exp_a_u     = exp(_a * u);
142                                 double a_power     = 1.0;
143                                 double b_power     = pow(_b, j - l);
144                                 int    b_exponent = j - l;
145                                 for (int k = 0; k <= j - l; k++)
146                                 {
147                                     double aux = 1.0;
148                                     switch (b_exponent % 4)
149                                     {
150                                     case 0: aux =  cb; break;
151                                     case 1: aux = -sb; break;
152                                     case 2: aux = -cb; break;
153                                     case 3: aux =  sb; break;
154                                     }
155                                     exp_au_cos_bu_derivative += BC(j - l, k) * a_power *
156                                                                 b_power * aux;
157                                     a_power    *= _a;
158                                     b_power    /= _b;
159                                     b_exponent --;
160                                 }
161                                 exp_au_cos_bu_derivative *= exp_a_u;
162                             }
163                             derivative += BC(j, l) * u_coefficient * u_power *
164                                           exp_au_cos_bu_derivative;
165                             u_coefficient *= u_exponent;
166                             u_exponent    --;
167                             if (u)
168                             {
169                                 u_power /= u;
170                             }
171                             else
172                             {
173                                 u_power = u_exponent ? 0.0 : 1.0;
174                             }
175                         }
176                         result = derivative;
177                     }
178                 }
179                 break;
180             case AET_SINE:
181                 {
182                     if (!_r)
183                     {
184                         // applying the Leibniz rule for the evaluation of the derivative d^{j}/du^{j} e^{au} sin(bu)
185                         double derivative = 0.0;
186                         double exp_a_u     = exp(_a * u);
187                         double a_power     = 1.0;
188                         double b_power     = pow(_b, j);
189                         int    b_exponent = j;
190                         double cb          = cos(_b * u);
191                         double sb          = sin(_b * u);
192                         for (int k = 0; k <= j; k++)
```


© Ágoston Róth, 2018

```
193                         {
194                             double aux = 1.0;
195                             switch (b_exponent % 4)
196                             {
197                             case 0: aux =  sb; break;
198                             case 1: aux =  cb; break;
199                             case 2: aux = -sb; break;
200                             case 3: aux = -cb; break;
201                             }
202                             derivative += BC(j, k) * a_power * b_power * aux;
203                             a_power    *= _a;
204                             b_power    /= _b;
205                             b_exponent --;
206                         }
207                         derivative *= exp_a_u;
208                         result = derivative;
209                     }
210                     else
211                     {
212                         // applying the Leibniz rule for the evaluation of the derivative d^j/du^j u^r e^au sin(bu)
213                         double derivative    = 0.0;
214                         double u_coefficient = 1.0;
215                         double u_power       = pow(u, _r);
216                         int    u_exponent    = _r;
217                         double cb            = cos(_b * u);
218                         double sb            = sin(_b * u);
219                         for (int l = 0; l <= min(j, _r); l++)
220                         {
221                             // applying the Leibniz rule for the evaluation of the derivative d^(j-l)/du^(j-l) e^au sin(bu)
222                             double exp_au_sin_bu_derivative = 0.0;
223                             {
224                                 double exp_a_u    = exp(_a * u);
225                                 double a_power    = 1.0;
226                                 double b_power    = pow(_b, j - l);
227                                 int    b_exponent = j - l;
228                                 for (int k = 0; k <= j - l; k++)
229                                 {
230                                     double aux = 1.0;
231                                     switch (b_exponent % 4)
232                                     {
233                                     case 0: aux =  sb; break;
234                                     case 1: aux =  cb; break;
235                                     case 2: aux = -sb; break;
236                                     case 3: aux = -cb; break;
237                                     }
238                                     exp_au_sin_bu_derivative += BC(j - l, k) * a_power *
239                                                                 b_power * aux;
240                                     a_power    *= _a;
241                                     b_power    /= _b;
242                                     b_exponent --;
243                                 }
244                                 exp_au_sin_bu_derivative *= exp_a_u;
245                             }
246                             derivative += BC(j, l) * u_coefficient * u_power *
247                                           exp_au_sin_bu_derivative;
248                             u_coefficient *= u_exponent;
```


© Ágoston Róth, 2018

```
249                             u_exponent    --;
250                             if (u)
251                             {
252                                 u_power /= u;
253                             }
254                             else
255                             {
256                                 u_power = u_exponent ? 0.0 : 1.0;
257                             }
258                         }
259                         result = derivative;
260                     }
261                 }
262                 break;
263             case AE_COSINE:
264                 {
265                     if (!_r)
266                     {
267                         return pow(_a, j) * exp(_a * u);
268                     }
269                     else
270                     {
271                         // applying the Leibniz rule for the evaluation of the derivative d^j/du^j u^r e^au
272                         double derivative    = 0.0;
273                         double u_coefficient = 1.0;
274                         int    u_exponent    = _r;
275                         double u_power       = pow(u, _r);
276                         double exp_a_u       = exp(_a * u);
277                         double a_power       = pow(_a, j);
278                         for (int k = 0; k <= min(j, _r); k++)
279                         {
280                             derivative    += BC(j, k) * u_coefficient * u_power *
281                                              a_power;
282                             u_coefficient *= u_exponent;
283                             u_exponent    --;
284                             if (u)
285                             {
286                                 u_power   /= u;
287                             }
288                             else
289                             {
290                                 u_power = u_exponent ? 0.0 : 1.0;
291                             }
292                             a_power       /= _a;
293                         }
294                         derivative *= exp_a_u;
295                         result = derivative;
296                     }
297                 }
298                 break;
299             case AE_SINE:
300                 result = 0.0;
301                 break;
302             case AT_COSINE:
303                 {
304                     if (!_r)
305                     {
306                         double sb    = (j > 0 ? sin(_b * u) : 0.0);
307                         double cb    = cos(_b * u);
308                         double power = pow(_b, j);
```


© Ágoston Róth, 2018

```
309                     switch (j % 4)
310                     {
311                     case 0: return  power * cb;
312                     case 1: return -power * sb;
313                     case 2: return -power * cb;
314                     case 3: return  power * sb;
315                     }
316                 }
317                 else
318                 {
319                     // applying the Leibniz rule for the evaluation of the derivative d^j/du^j u^r cos(bu)
320                     double derivative    = 0.0;
321                     double u_coefficient = 1.0;
322                     double u_power       = pow(u, _r);
323                     int    u_exponent    = _r;
324                     double b_power       = pow(_b, j);
325                     int    b_exponent    = j;
326                     double cb            = cos(_b * u);
327                     double sb            = (j > 0 ? sin(_b * u) : 0.0);
328                     for (int k = 0; k <= min(j, _r); k++)
329                     {
330                         double aux = 1.0;
331                         switch (b_exponent % 4)
332                         {
333                         case 0: aux =  cb; break;
334                         case 1: aux = -sb; break;
335                         case 2: aux = -cb; break;
336                         case 3: aux =  sb; break;
337                         }
338                         derivative    += BC(j, k) * u_coefficient * u_power *
339                                          b_power * aux;
340                         u_coefficient *= u_exponent;
341                         u_exponent    --;
342                         if (u)
343                         {
344                             u_power   /= u;
345                         }
346                         else
347                         {
348                             u_power = u_exponent ? 0.0 : 1.0;
349                         }
350                         b_power       /= _b;
351                         b_exponent    --;
352                     }
353                     result = derivative;
354                 }
355             }
356             break;
357         case AT_SINE:
358             {
359                 if (!_r)
360                 {
361                     double cb    = (j > 0 ? cos(_b * u) : 0.0);
362                     double sb    = sin(_b * u);
363                     double power = pow(_b, j);
364                     switch (j % 4)
365                     {
366                     case 0: return  power * sb;
367                     case 1: return  power * cb;
368                     case 2: return -power * sb;
369                     case 3: return -power * cb;
```


© Ágoston Róth, 2018

```
370                     }
371                 }
372                 else
373                 {
374                     // applying the Leibniz rule for the evaluation of the derivative d^j/du^j u^r cos(bu)
375                     double derivative    = 0.0;
376                     double u_coefficient = 1.0;
377                     double u_power       = pow(u, _r);
378                     int    u_exponent    = _r;
379                     double b_power       = pow(_b, j);
380                     int    b_exponent    = j;
381                     double cb            = (j > 0 ? cos(_b * u) : 0.0);
382                     double sb            = sin(_b * u);
383                     for (int k = 0; k <= min(j, _r); k++)
384                     {
385                         double aux = 1.0;
386                         switch (b_exponent % 4)
387                         {
388                         case 0: aux =  sb; break;
389                         case 1: aux =  cb; break;
390                         case 2: aux = -sb; break;
391                         case 3: aux = -cb; break;
392                         }
393                         derivative    += BC(j, k) * u_coefficient * u_power *
394                                          b_power * aux;
395                         u_coefficient *= u_exponent;
396                         u_exponent    --;
397                         if (u)
398                         {
399                             u_power   /= u;
400                         }
401                         else
402                         {
403                             u_power = u_exponent ? 0.0 : 1.0;
404                         }
405                         b_power       /= _b;
406                         b_exponent    --;
407                     }
408                     result = derivative;
409                 }
410             }
411             break;
412         case P_COSINE:
413             {
414                 if (j > _r)
415                 {
416                     return 0.0;
417                 }
418                 double u_coefficient = 1.0;
419                 int    u_exponent    = _r - j;
420                 for (int k = _r; k > u_exponent; k--)
421                 {
422                     u_coefficient *= k;
423                 }
424                 result = u_coefficient * pow(u, u_exponent);
425             }
426             break;
427         case P_SINE:
428             result = 0.0;
```


© Ágoston Róth, 2018

```
429                 break;
430             }
431         }
432         return result;
433     }
434     // getters
435     ECSpace::_OrdinaryBasisFunction::Type ECSpace::_OrdinaryBasisFunction::type() const
436     {
437         return _type;
438     }
439     double ECSpace::_OrdinaryBasisFunction::a() const
440     {
441         return _a;
442     }
443     double ECSpace::_OrdinaryBasisFunction::b() const
444     {
445         return _b;
446     }
447     int ECSpace::_OrdinaryBasisFunction::r() const
448     {
449         return _r;
450     }
451     // Our smart pointers automatically delete the dynamically allocated objects referenced by them when they
452     // go out of scope. However, it may happen that we need to manually delete the referenced objects and
453     // to re-initialize the smart pointers to their default null values in order to highlight that some
454     // mathematical operation could not be performed successfully. For such special cases we
455     // will use the next private method.
456     void ECSpace::_deleteAllDynamicallyAllocatedObjects()
457     {
458         _rho                          = SP<RealMatrix>::Default();
459         _reversed_v_Wronskian_beta = SP<RealMatrix>::Default();
460         _L                            = SP<RealMatrix>::Default();
461         _U                            = SP<RealMatrix>::Default();
462         _lambda                       = SP< ColumnMatrix<double> >::Default();
463         _mu                           = SP<RealMatrix>::Default();
464     }
465     // default/special constructor
466     ECSpace::ECSpace(
467             double alpha, double beta,
468             bool check_for_ill_conditioned_matrices,
469             int expected_correct_significant_digits):
470             _alpha(alpha), _beta(beta)
471     {
472         if (_alpha >= _beta)
473         {
474             throw Exception("The starting point of the definition domain should be "
475                             "less than its ending point!");
476         }
477         // we have to ensure that z = 0 is at least a first order root of the characteristic polynomial,
478         // otherwise the underlying EC vector space of functions would not contain the constants and
479         // consequently would not possess normalizable bases
480         _polynomial.insertZero(0.0, 0.0, 1);
481         updateBothBases(check_for_ill_conditioned_matrices,
482                         expected_correct_significant_digits);
483     }
484     // Inserts an mth order complex root of the form z = a ± ib into the factorization of the characteristic polynomial.
485     bool ECSpace::insertZero(
486             double a, double b, int m,
487             bool update_both_bases,
488             bool check_for_ill_conditioned_matrices,
489             int expected_correct_significant_digits)
490     {
491         _polynomial.insertZero(a, b, m);
```


© Ágoston Róth, 2018

```
492         if (update_both_bases)
493         {
494             return updateBothBases(check_for_ill_conditioned_matrices,
495                                    expected_correct_significant_digits);
496         }
497         return true;
498     }
499     bool ECSpace::insertZero(
500             const CharacteristicPolynomial::Zero &zero,
501             bool update_both_bases,
502             bool check_for_ill_conditioned_matrices,
503             int expected_correct_significant_digits)
504     {
505         _polynomial.insertZero(zero);
506         if (update_both_bases)
507         {
508             return updateBothBases(check_for_ill_conditioned_matrices,
509                                    expected_correct_significant_digits);
510         }
511         return true;
512     }
513     // If exists, deletes a complex root of the form z = a ± ib independently of its order. Note that the
514     // root z = 0 will never be deleted since its existence its critical.
515     bool ECSpace::deleteZero(
516             double a, double b,
517             bool update_both_bases,
518             bool check_for_ill_conditioned_matrices,
519             int expected_correct_significant_digits)
520     {
521         if (a == 0.0 && b == 0.0)
522         {
523             return false;
524         }
525         if (!_polynomial.deleteZero(a, b))
526         {
527             return false;
528         }
529         else
530         {
531             if (update_both_bases)
532             {
533                 return updateBothBases(check_for_ill_conditioned_matrices,
534                                        expected_correct_significant_digits);
535             }
536             return true;
537         }
538     }
539     bool ECSpace::deleteZero(
540             const CharacteristicPolynomial::Zero &zero,
541             bool update_both_bases,
542             bool check_for_ill_conditioned_matrices,
543             int expected_correct_significant_digits)
544     {
545         if (zero.real == 0.0 && zero.imaginary == 0.0)
546         {
547             return false;
548         }
549         if (!_polynomial.deleteZero(zero))
550         {
551             return false;
552         }
553         else
554         {
555             if (update_both_bases)
556             {
```


© Ágoston Róth, 2018

```
557                 return updateBothBases(check_for_ill_conditioned_matrices,
558                                        expected_correct_significant_digits);
559             }
560             return true;
561         }
562     }
563     // Generates/updates all ordinary basis functions {φ_{n,i}(u) : u ∈ [α, β]}_{i=0}^n and also calculates all
564     // information necessary for the evaluation and differentiation of all normalized B-basis functions
565     // {b_{n,i}(u) : u ∈ [α, β]}_{i=0}^n.
566     // The method's implementation is based on the construction process ((1.11/3)–(1.13/3), (1.25/6)–(1.26/6)).
567     bool ECSpace::updateBothBases(
568             bool check_for_ill_conditioned_matrices,
569             int expected_correct_significant_digits)
570     {
571         if (!_polynomial.factorizationChanged())
572         {
573             return true;
574         }
575         _is_reflection_invariant = _polynomial.isEvenOrOddFunction();
576         _phi.clear();
577         for (int z = 0; z < (int)_polynomial._factorization.size(); z++)
578         {
579             CharacteristicPolynomial::Zero &zero = _polynomial._factorization[z];
580             if (zero.order >= 1)
581             {
582                 for (int r = 0; r <= zero.order - 1; r++)
583                 {
584                     _phi.push_back(_OrdinaryBasisFunction(
585                             zero.real, zero.imaginary, r, true));
586                     if (zero.imaginary != 0.0)
587                     {
588                         _phi.push_back(_OrdinaryBasisFunction(
589                                 zero.real, zero.imaginary, r, false));
590                     }
591                 }
592             }
593         }
594         int dimension = (int)_phi.size();
595         int n         = dimension - 1;
596         if (!dimension)
597         {
598             _deleteAllDynamicallyAllocatedObjects();
599             return false;
600         }
601         _rho     = SP<RealMatrix>::Default(
602                 new (nothrow) RealMatrix(dimension, dimension));
603         _reversed_v_Wronskian_beta = SP<RealMatrix>::Default(
604                 new (nothrow) RealMatrix(dimension, dimension));
605         _L       = SP<RealMatrix>::Default(
606                 new (nothrow) RealMatrix(dimension, dimension));
607         _U       = SP<RealMatrix>::Default(
608                 new (nothrow) RealMatrix(dimension, dimension));
609         _lambda = SP< ColumnMatrix<double> >::Default(
610                 new (nothrow) ColumnMatrix<double>(dimension));
611         _mu      = SP<RealMatrix>::Default(
612                 new (nothrow) RealMatrix(
613                         dimension,
614                         n / (_is_reflection_invariant ? 2 : 1) +1));
615         if (!_rho || !_reversed_v_Wronskian_beta || !_L || !_U || !_lambda || !_mu)
616         {
617             _deleteAllDynamicallyAllocatedObjects();
618             return false;
619         }
```


© Ágoston Róth, 2018

```
620         if (dimension == 1)
621         {
622             (*_rho)(0, 0) = (*_reversed_v_Wronskian_beta)(0, 0) = (*_L)(0, 0)
623                           = (*_U)(0, 0) = (*_lambda)[0] = (*_mu)(0, 0) = 1.0;
624             if (_polynomial.factorizationChanged())
625             {
626                 _polynomial.flipFactorizationChangedState();
627             }
628             return true;
629         }
630         RealMatrix phi_Wronskian_alpha(dimension, dimension);
631         RealMatrix phi_Wronskian_beta(n, dimension);
632         #pragma omp parallel for
633         for (int j_k = 0; j_k < dimension * dimension; j_k++)
634         {
635             int j = j_k / dimension;
636             int k = j_k % dimension;
637             phi_Wronskian_alpha(j, k) = _phi[k](j, _alpha);
638             if (j < n)
639             {
640                 phi_Wronskian_beta(j, k)  = _phi[k](j, _beta);
641             }
642         }
643         vector<double> condition_numbers(
644                 check_for_ill_conditioned_matrices ? dimension : 0);
645         bool particular_integrals_aborted = false;
646         #pragma omp parallel for
647         for (int i = 0; i <= n; i++)
648         {
649             #pragma omp flush (particular_integrals_aborted)
650             if (!particular_integrals_aborted)
651             {
652                 RealMatrix Hermite(dimension, dimension);
653                 RowMatrix<double> initial_conditions(dimension);
654                 initial_conditions[i] = 1.0;
655                 for (int j = 0; j <= i; j++)
656                 {
657                     for (int k = 0; k <= n; k++)
658                     {
659                         Hermite(j, k) = phi_Wronskian_alpha(j, k);
660                     }
661                 }
662                 for (int j = 0; j <= n - 1 - i; j++)
663                 {
664                     for (int k = 0; k <= n; k++)
665                     {
666                         Hermite(i + 1 + j, k) = phi_Wronskian_beta(j, k);
667                     }
668                 }
669                 if (check_for_ill_conditioned_matrices)
670                 {
671                     SVDecomposition SVD(Hermite);
672                     if (!SVD.isCorrect())
673                     {
674                         particular_integrals_aborted = true;
675                         #pragma omp flush (particular_integrals_aborted)
676                     }
677                     else
678                     {
679                         condition_numbers[i] = SVD.conditionNumber();
```


© Ágoston Róth, 2018

```
680                     }
681                 }
682                 PLUDecomposition  PLUD(Hermite);
683                 RowMatrix<double> particular_integral_coefficients;
684                 if (!PLUD.solveLinearSystem(
685                         initial_conditions, particular_integral_coefficients, false))
686                 {
687                     particular_integrals_aborted = true;
688                     #pragma omp flush (particular_integrals_aborted)
689                 }
690                 else
691                 {
692                     _rho->setRow(i, particular_integral_coefficients);
693                 }
694             }
695         }
696         if (particular_integrals_aborted)
697         {
698             _deleteAllDynamicallyAllocatedObjects();
699             return false;
700         }
701         if (check_for_ill_conditioned_matrices)
702         {
703             double maximal_condition_number = -std::numeric_limits<double>::max();
704             for (int i = 0; i < dimension; i++)
705             {
706                 if (maximal_condition_number < condition_numbers[i])
707                 {
708                     maximal_condition_number = condition_numbers[i];
709                 }
710             }
711             double estimated_correct_significant_digits =
712                     -log10(2.0 * MACHINE_EPS * maximal_condition_number);
713             if (estimated_correct_significant_digits <
714                 expected_correct_significant_digits)
715             {
716                 _deleteAllDynamicallyAllocatedObjects();
717                 throw Exception("Based on one of the calculated condition numbers (" +
718                                 toString(maximal_condition_number) +
719                                 "), the estimated number (" +
720                                 toString(estimated_correct_significant_digits) +
721                                 ") of correct significant digits is less than the "
722                                 "number (" +
723                                 toString(expected_correct_significant_digits) +
724                                 ") of the expected ones! (When a condition number is "
725                                 "large, the corresponding system is considered ill-"
726                                 "conditioned and the solution may not be accurate.)\n"
727                                  "Try one of the followings:\n"
728                                  "   1) lower the number of expected correct significant "
729                                  "digits,\n"
730                                  "   2) decrease the dimension of the underlying EC "
731                                  "space,\n"
732                                  "   3) change the endpoints of the definition domain,\n"
733                                  "   4) run your code without testing for ill-"
734                                  "conditioned matrices and hope for the best (the "
735                                  "standard condition number may lead to an overly "
736                                  "pessimistic estimate for the overall error).");
737             }
738         }
739         #pragma omp parallel for
740         for (int j = 0; j <= n; j++)
741         {
742             for (int i = n; i >= 0; i--)
743             {
744                 for (int k = 0; k <= n; k++)
745                 {
746                     (*_reversed_v_Wronskian_beta)(j, n - i) += (*_rho)(i, k) *
```


© Ágoston Róth, 2018

```
747                                                         _phi[k](j, _beta);
748                 }
749             }
750         }

751         if (check_for_ill_conditioned_matrices)
752         {
753             SVDecomposition SVD(*_reversed_v_Wronskian_beta);

754             if (!SVD.isCorrect())
755             {
756                 _deleteAllDynamicallyAllocatedObjects();
757                 return false;
758             }

759             double condition_number = SVD.conditionNumber();

760             double estimated_correct_significant_digits =
761                     -log10(2.0 * MACHINE_EPS * condition_number);

762             if (estimated_correct_significant_digits <
763                 expected_correct_significant_digits)
764             {
765                 _deleteAllDynamicallyAllocatedObjects();
766                 throw Exception("Based on one of the calculated condition numbers (" +
767                                 toString(condition_number) +
768                                 "), the estimated number (" +
769                                 toString(estimated_correct_significant_digits) +
770                                 ") of correct significant digits is less than the "
771                                 "number (" +
772                                 toString(expected_correct_significant_digits) +
773                                 ") of the expected ones! (When a condition number is "
774                                 "large, the corresponding system is considered ill-"
775                                 "conditioned and the solution may not be accurate.)\n"
776                                 "Try one of the followings:\n"
777                                 "  1) lower the number of expected correct significant "
778                                 "digits,\n"
779                                 "  2) decrease the dimension of the underlying EC "
780                                 "space,\n"
781                                 "  3) change the endpoints of the definition domain,\n"
782                                 "  4) run your code without testing for ill-"
783                                 "conditioned matrices and hope for the best (the "
784                                 "standard condition number may lead to an overly "
785                                 "pessimistic estimate for the overall error).");
786             }
787         }

788         FactorizedUnpivotedLUDecomposition FLUD(*_reversed_v_Wronskian_beta);

789         if (!FLUD.isCorrect())
790         {
791             _deleteAllDynamicallyAllocatedObjects();
792             return false;
793         }

794         ColumnMatrix<double> e0(dimension);
795         e0[0] = 1.0;

796         Matrix<GLdouble> identity(dimension, n / (_is_reflection_invariant ? 2 : 1) + 1);

797         for (int i = 0; i < identity.columnCount(); i++)
798         {
799             identity(i, i) = 1.0;
800         }

801         if (!FLUD.solveLLinearSystem(e0, *_lambda) ||
802             !FLUD.solveULinearSystem(identity, *_mu))
803         {
804             _deleteAllDynamicallyAllocatedObjects();
805             return false;
806         }

807         _polynomial.flipFactorizationChangedState();
```


© Ágoston Róth, 2018

```
808         return true;
809     }
810     // Returns the dimension of the underlying EC vector space of functions.
811     int ECSpace::dimension() const
812     {
813         if (_polynomial.factorizationChanged())
814         {
815             throw Exception("Before checking the dimension both the ordinary basis and "
816                             "the normalized B-basis of the underlying EC space should "
817                             "be updated!");
818         }
819         return (int)_phi.size();
820     }
821     // Returns the starting point of the definition domain.
822     double ECSpace::alpha() const
823     {
824         return _alpha;
825     }
826     // Returns the ending point of the definition domain.
827     double ECSpace::beta() const
828     {
829         return _beta;
830     }
831     // Overloaded function operator that evaluates the jth order (j ≥ 0) derivative either of the ith ordinary basis
832     // function or of the normalized B-basis function at the parameter value u ∈ [α, β].
833     // The method's implementation is based on Corollary 1.1/7.
834     double ECSpace::operator ()(ECSpace::BasisFunctionType type,
835                                 int i, int j, double u) const
836     {
837         if (_polynomial.factorizationChanged())
838         {
839             throw Exception("Before differentiation both the ordinary basis and the "
840                             "normalized B-basis of the underlying EC space should be "
841                             "updated!");
842         }
843         int dimension = (int)_phi.size();
844         if (!dimension)
845         {
846             throw Exception("The dimension of the underlying EC space of functions "
847                             "should be at least 1!");
848         }
849         if (i < 0 || i > dimension)
850         {
851             throw Exception("The function index i should be a non-negative integer that "
852                             "is strictly less than the dimension of the underlying EC "
853                             "vector space of functions!");
854         }
855         if (j < 0)
856         {
857             throw Exception("The differentiation order j should be a non-negative "
858                             "integer!");
859         }
860         if (type == ORDINARY_BASIS)
861         {
862             return _phi[i](j, u);
863         }
864         else
865         {
866             if (!_rho || !_reversed_v_Wronskian_beta || !_L || !_U || !_lambda || !_mu)
867             {
868                 throw Exception("The normalized B-basis of the underlying EC vector "
869                                 "space of functions was not updated!");
870             }
```


© Ágoston Róth, 2018

```
871             int n         = dimension - 1;
872             int index     = n - i;
873             double result = 0.0;
874             if (!_is_reflection_invariant || (_is_reflection_invariant && (i > n / 2)))
875             {
876                 #pragma omp parallel for
877                 for (int r = 0; r <= index; r++)
878                 {
879                     double c_sum = 0.0;
880                     for (int c = 0; c <= n; c++)
881                     {
882                         c_sum += (*_rho)(n - r, c) * _phi[c](j, u);
883                     }
884                     #pragma omp atomic
885                     result += (*_mu)(r, index) * c_sum;
886                 }
887                 result *= (*_lambda)[index];
888             }
889             else
890             {
891                 if (i <= n / 2)
892                 {
893                     u = _alpha + _beta - u;
894                     #pragma omp parallel for
895                     for (int r = 0; r <= i; r++)
896                     {
897                         double c_sum = 0.0;
898                         for (int c = 0; c <= n; c++)
899                         {
900                             c_sum += (*_rho)(n - r, c) * _phi[c](j, u);
901                         }
902                         #pragma omp atomic
903                         result += (*_mu)(r, i) * c_sum;
904                     }
905                     result *= (*_lambda)[i];
906                     result *= ((j % 2) ? -1.0 : 1.0);
907                 }
908             }
909             return result;
910         }
911     }
912     // Calculates the entries of the matrix [t^n_{i,j}]^{n,n}_{i=0,j=0} that appears in the general basis transformation (1.45/11)
913     // that maps the normalized B-basis {b_{n,i}(u) : u ∈ [α,β]}^n_{i=0} of the underlying EC vector space of functions
914     // to its ordinary basis {φ_{n,i}(u) : u ∈ [α,β]}^n_{i=0}. The method's implementation is based on Theorem 1.2/11.
915     RealMatrix *ECSpace::basisTransformationFromNBToOrdinary() const
916     {
917         if (_polynomial.factorizationChanged())
918         {
919             throw Exception("Before the evaluation of the transformation matrix both "
920                             "the ordinary basis and the normalized B-basis of the "
921                             "underlying EC space should be updated!");
922         }
923         int dimension = (int)_phi.size();
924         int n         = dimension - 1;
925         RealMatrix *T = new (nothrow) RealMatrix(dimension, dimension);
926         if (!T)
927         {
928             return nullptr;
929         }
930         #pragma omp parallel for
931         for (int j = 0; j <= n; j++)
932         {
933             (*T)(0, j) = 1.0;
```


© Ágoston Róth, 2018

```
934         }
935         #pragma omp parallel for
936         for (int i = 1; i <= n; i++)
937         {
938             (*T)(i, 0) = _phi[i](0, _alpha);
939             (*T)(i, n) = _phi[i](0, _beta);
940         }
941         for (int j = 1; j <= n / 2; j++)
942         {
943             RowMatrix<double> d_j_b_0_to_j_at_alpha(j + 1);
944             #pragma omp parallel for
945             for (int k = 0; k <= j; k++)
946             {
947                 d_j_b_0_to_j_at_alpha[k] = operator ()(B_BASIS, k, j, _alpha);
948             }
949             #pragma omp parallel for
950             for (int i = 1; i <= n; i++)
951             {
952                 (*T)(i, j) =  _phi[i](j, _alpha);
953                 for (int k = 0; k <= j - 1; k++)
954                 {
955                     (*T)(i, j) -= (*T)(i, k) * d_j_b_0_to_j_at_alpha[k];
956                 }
957                 (*T)(i, j) /= d_j_b_0_to_j_at_alpha[j];
958             }
959         }
960         for (int j = 1; j <= (n - 1) / 2; j++)
961         {
962             RowMatrix<double> d_j_b_n_downto_n_minus_j_at_beta(j + 1);
963             #pragma omp parallel for
964             for (int k = 0; k <= j; k++)
965             {
966                 d_j_b_n_downto_n_minus_j_at_beta[k] = operator ()(B_BASIS,
967                                                                   n - k, j, _beta);
968             }
969             #pragma omp parallel for
970             for (int i = 1; i <= n; i++)
971             {
972                 (*T)(i, n - j) =  _phi[i](j, _beta);
973                 for (int k = 0; k <= j - 1; k++)
974                 {
975                     (*T)(i, n - j) -= (*T)(i, n - k) *
976                                       d_j_b_n_downto_n_minus_j_at_beta[k];
977                 }
978                 (*T)(i, n - j) /= d_j_b_n_downto_n_minus_j_at_beta[j];
979             }
980         }
981         return T;
982     }
983     // Determines whether the specified EC vector space is reflection invariant.
984     bool ECSpace::isReflectionInvariant() const
985     {
986         if (_polynomial.factorizationChanged())
987         {
988             throw Exception("Before checking the parity of the characteristic "
989                             "polynomial both the ordinary basis and the normalized "
990                             "B-basis of the underlying EC space should be updated!");
991         }
992         return _is_reflection_invariant;
993     }
```


© Ágoston Róth, 2018

```
994     // If possible, generates the LaTeX expression of the ith ordinary basis functions.
995     bool ECSpace::LaTeXExpression(int i, string &expression) const
996     {
997         if (_polynomial.factorizationChanged() || i >= (int)_phi.size())
998         {
999             expression.clear();
1000            return false;
1001        }

1002        expression.clear();

1003        switch (_phi[i].type())
1004        {
1005        case _OrdinaryBasisFunction::AET_COSINE:

1006            if (_phi[i].r())
1007            {
1008                if (_phi[i].r() == 1)
1009                {
1010                    expression += "u \\cdot ";
1011                }
1012                else
1013                {
1014                    expression += "u^{";
1015                    expression += toString(_phi[i].r());
1016                    expression += "} \\cdot ";
1017                }
1018            }

1019            expression += "e^{";
1020            if (abs(_phi[i].a()) != 1.0)
1021            {
1022                expression += toString(_phi[i].a());
1023                expression += " \\cdot u} \\cdot \\cos\\left(";
1024            }
1025            else
1026            {
1027                expression += (_phi[i].a() > 0.0 ? "" : "-");
1028                expression += "u} \\cdot \\cos\\left(";
1029            }

1030            if (abs(_phi[i].b()) != 1.0)
1031            {
1032                expression += toString(abs(_phi[i].b()));
1033                expression += " \\cdot u\\right)";
1034            }
1035            else
1036            {
1037                expression += "u\\right)";
1038            }

1039            break;

1040        case _OrdinaryBasisFunction::AET_SINE:

1041            if (_phi[i].b() < 0.0)
1042            {
1043                expression += "-";
1044            }

1045            if (_phi[i].r())
1046            {
1047                if (_phi[i].r() == 1)
1048                {
1049                    expression += "u \\cdot ";
1050                }
1051                else
1052                {
1053                    expression += "u^{";
1054                    expression += toString(_phi[i].r());
1055                    expression += "} \\cdot ";
1056                }
1057            }
```


© Ágoston Róth, 2018

```
1058            expression += "e^{";
1059            if (abs(_phi[i].a()) != 1.0)
1060            {
1061                expression += toString(_phi[i].a());
1062                expression += " \\cdot u} \\cdot \\sin\\left(";
1063            }
1064            else
1065            {
1066                expression += (_phi[i].a() > 0.0 ? "" : "-");
1067                expression += "u} \\cdot \\sin\\left(";
1068            }

1069            if (abs(_phi[i].b()) != 1.0)
1070            {
1071                expression += toString(abs(_phi[i].b()));
1072                expression += " \\cdot u\\right)";
1073            }
1074            else
1075            {
1076                expression += "u\\right)";
1077            }

1078            break;

1079        case _OrdinaryBasisFunction::AE_COSINE:

1080            if (_phi[i].r())
1081            {
1082                if (_phi[i].r() == 1)
1083                {
1084                    expression += "u \\cdot ";
1085                }
1086                else
1087                {
1088                    expression += "u^{";
1089                    expression += toString(_phi[i].r());
1090                    expression += "} \\cdot ";
1091                }
1092            }

1093            expression += "e^{";
1094            if (abs(_phi[i].a()) != 1.0)
1095            {
1096                expression += toString(_phi[i].a());
1097                expression += " \\cdot u}";
1098            }
1099            else
1100            {
1101                expression += (_phi[i].a() > 0.0 ? "" : "-");
1102                expression += "u}";
1103            }

1104            break;

1105        case _OrdinaryBasisFunction::AE_SINE:
1106            expression += "0";
1107            break;

1108        case _OrdinaryBasisFunction::AT_COSINE:

1109            if (_phi[i].r())
1110            {
1111                if (_phi[i].r() == 1)
1112                {
1113                    expression += "u \\cdot ";
1114                }
1115                else
1116                {
1117                    expression += "u^{";
1118                    expression += toString(_phi[i].r());
1119                    expression += "} \\cdot ";
1120                }
1121            }
```


© Ágoston Róth, 2018

```
1122            if (abs(_phi[i].b()) != 1.0)
1123            {
1124                expression += "\\cos\\left(";
1125                expression += toString(abs(_phi[i].b()));
1126                expression += " \\cdot u\\right)";
1127            }
1128            else
1129            {
1130                expression += "\\cos\\left(u\\right)";
1131            }
1132            break;
1133        case _OrdinaryBasisFunction::AT_SINE:
1134            if (_phi[i].b() < 0.0)
1135            {
1136                expression += "-";
1137            }
1138            if (_phi[i].r())
1139            {
1140                if (_phi[i].r() == 1)
1141                {
1142                    expression += "u \\cdot ";
1143                }
1144                else
1145                {
1146                    expression += "u^{";
1147                    expression += toString(_phi[i].r());
1148                    expression += "} \\cdot ";
1149                }
1150            }
1151            if (abs(_phi[i].b()) != 1.0)
1152            {
1153                expression += "\\sin\\left(";
1154                expression += toString(abs(_phi[i].b()));
1155                expression += " \\cdot u\\right)";
1156            }
1157            else
1158            {
1159                expression += "\\sin\\left(u\\right)";
1160            }
1161            break;
1162        case _OrdinaryBasisFunction::P_COSINE:
1163            if (_phi[i].r())
1164            {
1165                if (_phi[i].r() > 1)
1166                {
1167                    expression += "u^{";
1168                    expression += toString(_phi[i].r());
1169                    expression += "}";
1170                }
1171                else
1172                {
1173                    expression += "u";
1174                }
1175            }
1176            else
1177            {
1178                expression += "1";
1179            }
1180            break;
1181        case _OrdinaryBasisFunction::P_SINE:
1182            expression += "0";
```


© Ágoston Róth, 2018

```
1183            break;
1184        }
1185
1186        return true;
1187    }
1188
1189    // Sets the endpoints of the definition domain (in case of modifications the normalized B-basis has to be udated).
1190    bool ECSpace::setDefinitionDomain(
1191            double alpha, double beta,
1192            bool check_for_ill_conditioned_matrices,
1193            int expected_correct_significant_digits)
1194    {
1195        if ((_alpha != alpha || _beta != beta) && _alpha < beta)
1196        {
1197            _alpha = alpha;
1198            _beta  = beta;
1199            _polynomial.flipFactorizationChangedState();
1200        }
1201
1202        return updateBothBases(check_for_ill_conditioned_matrices,
1203                               expected_correct_significant_digits);
1204    }
1205
1206    // Returns whether the ordninary basis and the normalized B-basis have to be updated.
1207    bool ECSpace::factorizationOfTheCharacteristicPolynomialChanged() const
1208    {
1209        return _polynomial.factorizationChanged();
1210    }
1211
1212    // Generates images of all ordinary or normalized B-basis functions.
1213    RowMatrix<SP<GenericCurve3>::Default>* ECSpace::generateImagesOfAllBasisFunctions(
1214            BasisFunctionType type,
1215            int maximum_order_of_derivatives, int div_point_count) const
1216    {
1217        assert("The given maximum order of derivatives should be non-negative!" &&
1218               maximum_order_of_derivatives >= 0);
1219
1220        assert("The number of subdivision points should be greater than 1!" &&
1221               div_point_count > 1);
1222
1223        if (maximum_order_of_derivatives < 0 || div_point_count <= 1 ||
1224            _polynomial.factorizationChanged())
1225        {
1226            return nullptr;
1227        }
1228
1229        int dimension = (int)_phi.size();
1230        int n         = dimension - 1;
1231
1232        RowMatrix<SP<GenericCurve3>::Default> *result =
1233                new (nothrow) RowMatrix<SP<GenericCurve3>::Default>(dimension);
1234
1235        if (!result)
1236        {
1237            return nullptr;
1238        }
1239
1240        GLdouble step = (_beta - _alpha) / (div_point_count - 1);
1241
1242        RowMatrix<GLdouble> u(div_point_count);
1243
1244        #pragma omp parallel for
1245        for (int k = 0; k < div_point_count; k++)
1246        {
1247            u[k] = min(_alpha + k * step, _beta);
1248        }
1249
1250        for (int i = 0; i <= n; i++)
1251        {
1252            (*result)[i] = SP<GenericCurve3>::Default(
1253                    new (nothrow) GenericCurve3(maximum_order_of_derivatives,
1254                                                div_point_count));
1255
1256            if (!(*result)[i])
```


© Ágoston Róth, 2018

```
1242                {
1243                    delete result, result = nullptr;
1244                    return result;
1245                }
1246                GenericCurve3 &function_i = *(*result)[i];
1247                #pragma omp parallel for
1248                for (int j_k = 0; j_k < (maximum_order_of_derivatives + 1) * div_point_count;
1249                     j_k++)
1250                {
1251                    int j = j_k / div_point_count;
1252                    int k = j_k % div_point_count;
1253                    Cartesian3 &derivative = function_i(j, k);
1254                    switch (j) {
1255                    case 0:
1256                        derivative[0] = u[k];
1257                        break;
1258                    case 1:
1259                        derivative[0] = 1.0;
1260                        break;
1261                    default:
1262                        derivative[0] = 0.0;
1263                        break;
1264                    }
1265                    derivative[1] = operator ()(type, i, j, u[k]);
1266                }
1267            }
1268            return result;
1269        }
1270        // clone function required by smart pointers based on the deep copy ownership policy
1271        ECSpace* ECSpace::clone() const
1272        {
1273            return new ECSpace(*this);
1274        }
1275        // destructor
1276        ECSpace::~ECSpace()
1277        {
1278            _deleteAllDynamicallyAllocatedObjects();
1279        }
1280        // overloaded output to stream operator
1281        ostream& operator <<(ostream &lhs, const ECSpace &rhs)
1282        {
1283            return lhs << rhs._polynomial << endl;
1284        }
1285    }
```

2.17 B-curves

B-curves of type (1.19/4) are represented by the class BCurve3 that is derived from the abstract base class LinearCombination3 and is based on the results of Corollary 1.1, Lemma 1.1 and Theorems 1.1, 1.2–1.3. It can be used to perform general order elevation, subdivision and to describe exactly user-specified ordinary integral curves by means of convex combinations of control points and normalized B-basis functions. The diagram of the class is illustrated in Fig. 2.26/242, its definition and implementation are listed in Listings 2.50/241 and 2.51/244, respectively. Note that the class redeclares and defines those pure virtual methods that are inherited as interfaces from the abstract base class LinearCombination3.

Listing 2.50. B-curves (**EC/BCurves3.h**)

© Ágoston Róth, 2018

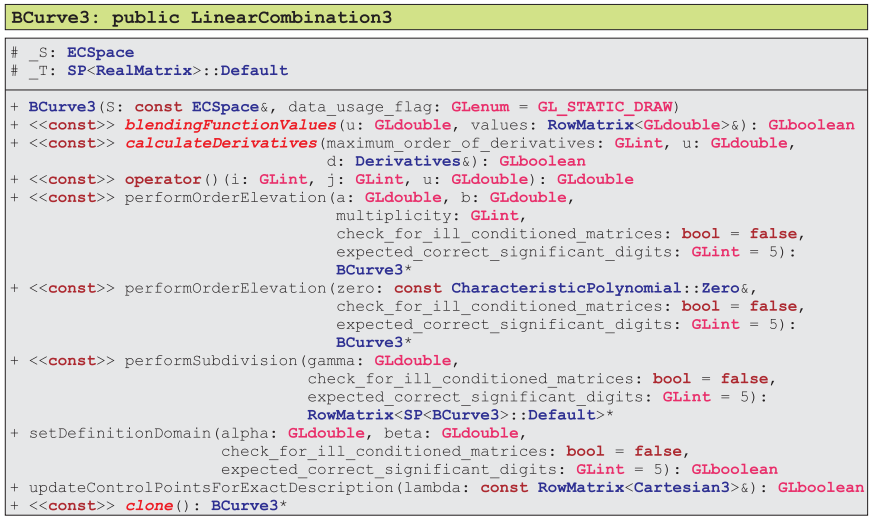


Fig. 2.26: Class diagram of general B-curves

```
#ifndef BCURVES3_H
#define BCURVES3_H

#include "../Core/SmartPointers/SpecializedSmartPointers.h"
#include "../Core/Geometry/Curves/LinearCombinations3.h"
#include "ECSpaces.h"

namespace cagd
{
    // The class BCurve3 below represents a B-curve of type (1.19/4). It is derived from the abstract base class
    // LinearCombination3 that provides two pure virtual methods which have to be redeclared and defined in
    // this derived class, otherwise one cannot instantiate curve objects from it. One of these methods is
    //
    //     GLboolean blendingFunctionValues(GLdouble u, RowMatrix<GLdouble> &values) const
    //
    // that has to evaluate all normalized B-basis functions of the underlying EC space represented by the variable _S.
    // This method is required by the base class LinearCombination3 when it tries to solve user-defined curve
    // interpolation problems by calling its method
    //
    //     virtual GLboolean updateDataForInterpolation(const ColumnMatrix<GLdouble>& knot_vector,
    //                                                  const ColumnMatrix<Cartesian3>& data_points_to_interpolate).
    //
    // The other pure virtual method that has to be redeclared and implemented is
    //
    //     GLboolean calculateDerivatives(GLuint max_order_of_derivatives, GLdouble u, Derivatives &d) const
    //
    // that is used by the base class in order to generate the image of the B-curve by invoking its method
    //
    //     virtual GenericCurve3* generateImage(GLuint max_order_of_derivatives, GLuint div_point_count,
    //                                          GLenum usage_flag = GL_STATIC_DRAW) const.
    //
    // Apart from the aforementioned two pure virtual methods, from image generation and solution of curve
    // interpolation problems, the base class LinearCombination3 also provides other fully implemented
    // functionalities that are responsible for the control point and vertex buffer object management of
    // the control polygon:
    //
    //     const Cartesian3&   operator [](const GLint &index) const; // get control point by constant reference
    //     Cartesian3&         operator [](const GLint &index);        // get control point by non-constant reference
    //
    //     virtual GLvoid deleteVertexBufferObjectsOfData();
    //     virtual GLboolean renderData(GLenum render_mode = GL_LINE_STRIP) const;
    //     virtual GLboolean updateVertexBufferObjectsOfData(GLenum usage_flag = GL_STATIC_DRAW).
    class BCurve3: public LinearCombination3
    {
```


© Ágoston Róth, 2018

```
43      protected:
44          // represents the EC space S_n^{α,β} = ⟨F_n^{α,β} := {φ_{n,i}(u) : u ∈ [α,β]}_{i=0}^n⟩ = ⟨B_n^{α,β} := {b_{n,i}(u) : u ∈ [α,β]}_{i=0}^n⟩,
45          // where 0 < β − α < ℓ'(S_n^{α,β})
46          ECSpace _S;
47          // a smart pointer to the transformation matrix [t_{i,j}^n]_{i=0,j=0}^{n,n} that maps the normalized B-basis B_n^{α,β} to the
48          // ordinary basis F_n^{α,β}
49          SP<RealMatrix>::Default _T;
50      public:
51          // special constructor
52          BCurve3(const ECSpace &S, GLenum data_usage_flag = GL_STATIC_DRAW);
53          // inherited pure virtual methods that have to be redeclared and defined
54          GLboolean blendingFunctionValues(GLdouble u, RowMatrix<GLdouble> &values) const;
55          GLboolean calculateDerivatives(GLint maximum_order_of_derivatives, GLdouble u,
56                                         Derivatives &d) const;
57          // Overloaded function operator that evaluates the derivative b_{n,i}^{(j)}(u), where u ∈ [α,β], i = 0,1,...,n
58          // and j ∈ ℕ.
59          GLdouble operator ()(GLint i, GLint j, GLdouble u) const;
60          // Overloaded member functions for general order elevation. In order to generate a higher dimensional EC
61          // space S_{n+q}^{α,β} that fulfills the condition S_n^{α,β} ⊂ S_{n+q}^{α,β}, where q ≥ 1 and 0 < β − α < min{ℓ'(S_n^{α,β}), ℓ'(S_{n+q}^{α,β})},
62          // one has either to define a new complex root z = a ± ib of multiplicity m ≥ 1, or to specify an existing
63          // root with a multiplicity that is greater than its former value m' ≥ 1. If z ∈ ℂ \ ℝ, then q = 2(m − m'),
64          // otherwise q = m − m', where m' = 0 whenever z denotes a nonexisting former root.
65          // The implementation of the next two methods are based on Lemma 1.1/8.
66          BCurve3* performOrderElevation(
67                  GLdouble a, GLdouble b, GLint multiplicity,
68                  bool check_for_ill_conditioned_matrices = false,
69                  GLint expected_correct_significant_digits = 5) const;
70          BCurve3* performOrderElevation(
71                  const CharacteristicPolynomial::Zero &zero,
72                  bool check_for_ill_conditioned_matrices = false,
73                  GLint expected_correct_significant_digits = 5) const;
74          // Using the general B-algorithm presented in Theorem 1.1/10, the method subdivides the given B-curve
75          // into two arcs of the same order at the parameter value γ ∈ (α,β). In case of success, the output is a
76          // nonzero raw pointer to a 2-element row matrix that stores 2 smart pointers (based on the deep copy
77          // ownership policy) that point to the left and right arcs of the subdivided curve.
78          virtual RowMatrix<SP<BCurve3>::Default>* performSubdivision(
79                  GLdouble gamma,
80                  bool check_for_ill_conditioned_matrices = false,
81                  GLint expected_correct_significant_digits = 5) const;
82          // Inherited virtual method that has to be redeclared and redefined, since if one alters the endpoints of
83          // the definition domain [α,β] both bases F_n^{α,β} and B_n^{α,β} of S_n^{α,β} have to be updated. Note that, the new
84          // interval length should also satisfy the condition 0 < β − α < ℓ'(S_n^{α,β}).
85          GLboolean setDefinitionDomain(GLdouble alpha, GLdouble beta,
86                                        bool check_for_ill_conditioned_matrices = false,
87                                        GLint expected_correct_significant_digits = 5);
88          // Given an ordinary integral curve c(u) = Σ_{i=0}^n λ_i φ_{n,i}(u), u ∈ [α,β], 0 < β − α < ℓ'(S_n^{α,β}), λ_i ∈ ℝ^3,
89          // the method determines the coefficients [p_j]_{j=0}^n of the normalized B-basis functions {b_{n,j}(u) : u ∈ [α,β]}_{j=0}^n
90          // such that c(u) ≡ Σ_{j=0}^n p_j b_{n,j}(u), ∀u ∈ [α,β]. Note that, the control points [p_j]_{j=0}^n that have to be
91          // updated are stored in the inherited data structure ColumnMatrix<Cartesian3> LinearCombination3::_data.
92          // The method's implementation is based on Theorems 1.2/11 and 1.3/11.
93          GLboolean updateControlPointsForExactDescription(
94                  const RowMatrix<Cartesian3> &lambda);
95          // clone function required by smart pointers based on the deep copy ownership policy
96          virtual BCurve3* clone() const;
97      };
98  }
99  #endif // BCURVES3_H
```


© Ágoston Róth, 2018

Listing 2.51. B-curves (**EC/BCurves3.cpp**)

```
#include "BCurves3.h"

#include "../Core/Exceptions.h"

#include <cmath>
#include <iostream>
#include <algorithm>
using namespace std;

namespace cagd
{
    // special constructor
    BCurve3::BCurve3(const ECSpace &S, GLenum data_usage_flag):
            LinearCombination3(S.alpha(), S.beta(), S.dimension(), data_usage_flag),
            _S(S),
            _T(_S.basisTransformationFromNBToOrdinary())
    {
        if (!_T)
        {
            throw Exception("The transformation matrix that maps the normalized "
                            "B-basis of the given EC space to its ordinary basis "
                            "could not be created!");
        }
    }

    // Calculates all normalized B-basis functions at the parameter value u ∈ [α, β].
    GLboolean BCurve3::blendingFunctionValues(
            GLdouble u, RowMatrix<GLdouble> &values) const
    {
        if (u < _S.alpha() || u > _S.beta())
        {
            values.resizeColumns(0);
            return GL_FALSE;
        }

        GLint dimension = _S.dimension();

        values.resizeColumns(dimension);

        #pragma omp parallel for
        for (GLint i = 0; i < dimension; i++)
        {
            values[i] = _S(ECSpace::B_BASIS, i, 0, u);
        }

        return GL_TRUE;
    }

    // Differentiates the given B-curve up to a maximal order at the parameter value u ∈ [α, β]. The zeroth and
    // higher order derivatives will be stored in the rows of the column matrix referenced by the variable d.
    GLboolean BCurve3::calculateDerivatives(
            GLint maximum_order_of_derivatives, GLdouble u, Derivatives &d) const
    {
        assert("The given maximum order of derivatives should be non-negative!" &&
               maximum_order_of_derivatives >= 0);

        if (maximum_order_of_derivatives < 0 || u < _S.alpha() || u > _S.beta())
        {
            d.resizeRows(0);
            return GL_FALSE;
        }

        GLint dimension = _S.dimension();

        d.resizeRows(maximum_order_of_derivatives + 1);
        d.loadNullVectors();

        // implementation of the formula c_n^(j)(u) = Σ_{i=0}^{n} p_i b_{n,i}^(j)(u), where the control points [p_i]_{i=0}^{n} are stored
        // in the inherited column matrix _data
        #pragma omp parallel for
        for (GLint j = 0; j <= maximum_order_of_derivatives; j++)
```


© Ágoston Róth, 2018

```
59          {
60              for (GLint i = 0; i < dimension; i++)
61              {
62                  d[j] += _data[i] * _S(ECSpace::B_BASIS, i, j, u);
63              }
64          }
65          return GL_TRUE;
66      }
67      // Overloaded function operator that evaluates the derivative b_{n,i}^{(j)}(u), where u ∈ [α, β], i = 0, 1, ..., n, j ∈ ℕ.
68      GLdouble BCurve3::operator ()(GLint i, GLint j, GLdouble u) const
69      {
70          assert("The given B-basis function index is out of bounds!" &&
71                 (i >= 0 && i < _S.dimension()));
72          assert("The given differentiation order should be non-negative!" && j >= 0);
73          return _S(ECSpace::B_BASIS, i, j, u);
74      }
75      // Overloaded member functions for general order elevation. In order to generate a higher dimensional EC
76      // space 𝕊_{n+q}^{α,β} that fulfills the condition 𝕊_n^{α,β} ⊂ 𝕊_{n+q}^{α,β}, where q ≥ 1 and 0 < β − α < min{ℓ'(𝕊_n^{α,β}), ℓ'(𝕊_{n+q}^{α,β})},
77      // one has either to define a new complex root z = a ± ib of multiplicity m ≥ 1, or to specify an existing
78      // root with a multiplicity that is greater than its former value m' ≥ 1. If z ∈ ℂ \ ℝ, then q = 2(m − m'),
79      // otherwise q = m − m', where m' = 0 whenever z denotes a nonexisting former root.
80      // The implementation of the next two methods are based on Lemma 1.1/8.
81      BCurve3* BCurve3::performOrderElevation(
82              GLdouble a, GLdouble b, GLint multiplicity,
83              bool check_for_ill_conditioned_matrices,
84              GLint expected_correct_significant_digits) const
85      {
86          return performOrderElevation(CharacteristicPolynomial::Zero(a, b, multiplicity),
87                                       check_for_ill_conditioned_matrices,
88                                       expected_correct_significant_digits);
89      }
90      BCurve3* BCurve3::performOrderElevation(
91              const CharacteristicPolynomial::Zero &zero,
92              bool check_for_ill_conditioned_matrices,
93              GLint expected_correct_significant_digits) const
94      {
95          assert("The number of expected correct significant digits should be "
96                 "non-negative!" && expected_correct_significant_digits >= 0);
97          if (expected_correct_significant_digits < 0)
98          {
99              return nullptr;
100         }
101         //At first, we try to clone our curve object together with is embedded EC space.
102         BCurve3* result = clone();
103         if (!result)
104         {
105             return nullptr;
106         }
107         try
108         {
109             // Then, we try to update the factorization of the characteristic polynomial (1.10/2) associated with the
110             // differential equation (1.9/2). If the factorization is changed, the ordinary basis and the normalized
111             // B-basis stored in the clone will be automatically updated.
112             if (!result->_S.insertZero(zero,
113                                        true,
114                                        check_for_ill_conditioned_matrices,
115                                        expected_correct_significant_digits))
116             {
117                 delete result, result = nullptr;
118                 return nullptr;
119             }
120         }
121         catch (Exception &e)
```


© Ágoston Róth, 2018

```
122         {
123             delete result, result = nullptr;
124             throw e;
125         }
126
127         // In what follows, we compare the dimensions of the original and new EC spaces.
128         GLint old_dimension      = _S.dimension();
129         GLint elevated_dimension = result->_S.dimension();
130
131         // If there is no difference, we return the pointer that points to the cloned object.
132         if (old_dimension == elevated_dimension)
133         {
134             return result;
135         }
136
137         GLdouble alpha = _S.alpha(), beta = _S.beta();
138
139         result->_data.resizeRows(elevated_dimension);
140
141         // If the difference is 1, we apply the order elevation formulas of Lemma 1.1/8:
142         if (elevated_dimension - old_dimension == 1)
143         {
144             GLint n = old_dimension - 1;
145
146             (*result)[0]             = _data[0];
147             (*result)[old_dimension] = _data[n];
148
149             // apply formula (1.31/8), i.e., p_{1,i} = (1 - b^{(i)}_{n,i}(α) / b^{(i)}_{n+1,i}(α)) p_{i-1} + b^{(i)}_{n,i}(α) / b^{(i)}_{n+1,i}(α) p_i, i = 1, ..., ⌊n/2⌋;
150             #pragma omp parallel for
151             for (GLint i = 1; i <= n / 2; i++)
152             {
153                 GLdouble ratio = (*this)(i, i, alpha) / (*result)(i, i, alpha);
154                 (*result)[i]   = (1 - ratio) * _data[i - 1] + ratio * _data[i];
155             }
156
157             // apply formula (1.32/8), i.e., p_{1,n+1-i} = b^{(i)}_{n,n-i}(β) / b^{(i)}_{n+1,n+1-i}(β) p_{n-i} + (1 - b^{(i)}_{n,n-i}(β) / b^{(i)}_{n+1,n+1-i}(β)) p_{n+1-i},
158             // i = 1, ..., ⌊(n+1)/2⌋.
159             #pragma omp parallel for
160             for (GLint i = 1; i <= old_dimension / 2; i++)
161             {
162                 GLdouble ratio                       = (*this)(n - i, i, beta) /
163                                                        (*result)(old_dimension - i, i, beta);
164                 (*result)[old_dimension - i] = ratio * _data[n - i] +
165                                                        (1 - ratio) * _data[old_dimension - i];
166             }
167
168             return result;
169         }
170
171         // If the difference between the dimensions is strictly greater than 1, we reduce the order elevation to a curve
172         // interpolation problem.
173         ColumnMatrix<GLdouble>     knot_vector(elevated_dimension);
174         ColumnMatrix<Cartesian3>   sample(elevated_dimension);
175
176         GLboolean sampling_aborted = GL_FALSE;
177         GLdouble  step = (beta - alpha) / (elevated_dimension - 1);
178
179         #pragma omp parallel for
180         for (GLint i = 0; i < elevated_dimension; i++)
181         {
182             #pragma omp flush (sampling_aborted)
183             if (!sampling_aborted)
184             {
185                 // uniform subdivision points of the interval [α, β]
186                 knot_vector[i] = min(alpha + i * step, beta);
187
188                 Derivatives d;
189                 if (!calculateDerivatives(0, knot_vector[i], d))
190                 {
191                     sampling_aborted = GL_TRUE;
```


© Ágoston Róth, 2018

```
179                     #pragma omp flush (sampling_aborted)
180                 }
181                 else
182                 {
183                     sample[i] = d[0]; // sample point of the original curve
184                 }
185             }
186         }
187         // If the sampling was aborted or the curve interpolation problem cannot be solved, the clone will be
188         // deleted and a null pointer will be returned.
189         if (sampling_aborted || !result->updateDataForInterpolation(knot_vector, sample))
190         {
191             delete result, result = nullptr;
192         }
193         return result;
194     }
195     // Using the general B-algorithm presented in Theorem 1.1/10, the method subdivides the given B-curve
196     // into two arcs of the same order at the parameter value γ ∈ (α, β). In case of success, the output is a
197     // nonzero raw pointer to a 2-element row matrix that stores 2 smart pointers (based on the deep copy
198     // ownership policy) that point to the left and right arcs of the subdivided curve.
199     RowMatrix<SP<BCurve3>::Default>* BCurve3::performSubdivision(
200             GLdouble gamma,
201             bool check_for_ill_conditioned_matrices,
202             GLint expected_correct_significant_digits) const
203     {
204         assert("The number of expected correct significant digits should be "
205                "non-negative!" && expected_correct_significant_digits >= 0);
206         GLdouble alpha  = _S.alpha(), beta = _S.beta();
207         // check whether γ ∈ (α, β)
208         if (gamma <= alpha || gamma >= beta || expected_correct_significant_digits < 0)
209         {
210             return nullptr;
211         }
212         // try to dynamically allocate a 2-element row matrix that stores smart pointers to the left and right arcs
213         RowMatrix<SP<BCurve3>::Default> *result =
214                 new (nothrow) RowMatrix<SP<BCurve3>::Default>(2);
215         if (!result)
216         {
217             return nullptr;
218         }
219         // Using the same EC space _S, the left and right arcs of the subdivided curve are defined over the
220         // subintervals [α, γ] and [γ, β], respectively.
221         (*result)[0] = SP<BCurve3>::Default(new (nothrow) BCurve3(_S, _data_usage_flag));
222         (*result)[1] = SP<BCurve3>::Default(new (nothrow) BCurve3(_S, _data_usage_flag));
223         if (!(*result)[0] || !(*result)[1])
224         {
225             delete result, result = nullptr;
226             return nullptr;
227         }
228         // In order to simplify the lines below and to ease the comparison to the proposed formulas, we will use
229         // references to the left and right arcs.
230         BCurve3 &lambda         = *(*result)[0], &rho           = *(*result)[1];
231         BCurve3 &b_alpha_gamma = lambda,         &b_gamma_beta = rho;
232         try
233         {
234             // Set the definition domains of the arcs, i.e., try to generate on both sections the corresponding
235             // ordinary bases, the normalized B-bases and the transformation matrices as well.
236             if (!lambda.setDefinitionDomain(alpha, gamma,
237                                             check_for_ill_conditioned_matrices,
238                                             expected_correct_significant_digits) ||
239                 !rho.setDefinitionDomain(gamma, beta,
240                                          check_for_ill_conditioned_matrices,
241                                          expected_correct_significant_digits))
```


© Ágoston Róth, 2018

```
242             {
243                 delete result, result = nullptr;
244                 return nullptr;
245             }
246         }
247         catch (Exception &e)
248         {
249             delete result, result = nullptr;
250             throw e;
251         }
252         GLint dimension = _S.dimension();
253         GLint n         = dimension - 1;
254         // Try to calculate the derivatives {c_n^(i)(α), c_n^(i)(γ), c_n^(i)(β)}_{i=0}^{⌊n/2⌋}.
255         Derivatives c_alpha, c_gamma, c_beta;
256         Derivatives *c_ptr[3] = {&c_alpha, &c_gamma, &c_beta};
257         GLdouble     u[3] = {alpha, gamma, beta};
258         GLboolean c_alpha_gamma_beta_derivatives_aborted = GL_FALSE;
259         #pragma omp parallel for
260         for (GLint i = 0; i < 3; i++)
261         {
262             #pragma omp flush (c_alpha_gamma_beta_derivatives_aborted)
263             if (!c_alpha_gamma_beta_derivatives_aborted)
264             {
265                 if (!calculateDerivatives(n / 2, u[i], *c_ptr[i]))
266                 {
267                     c_alpha_gamma_beta_derivatives_aborted = GL_TRUE;
268                     #pragma omp flush (c_alpha_gamma_beta_derivatives_aborted)
269                 }
270             }
271         }
272         if (c_alpha_gamma_beta_derivatives_aborted)
273         {
274             delete result, result = nullptr;
275             return nullptr;
276         }
277         // Set the initial conditions of recursive formulas (1.42/10) and (1.43/10).
278         lambda[n] = c_gamma[0], lambda[0]   = _data[0];
279         rho[0]    = c_gamma[0], rho[n]      = _data[n];
280         // Apply recursive formulas (1.42/10) and (1.43/10), i.e.,
281         // λ_{n-i}(γ) = 1/b^(i)_{n,n-i}(γ;α,γ) (c_n^(i)(γ) - Σ_{j=0}^{i-1} λ_{n-j}(γ) · b^(i)_{n,n-j}(γ;α,γ)), i = 1,...,⌊n/2⌋,
282         //    ϱ_i(γ) = 1/b^(i)_{n,i}(γ;γ,β) (c_n^(i)(γ) - Σ_{j=0}^{i-1} ϱ_j(γ) · b^(i)_{n,j}(γ;γ,β)), i = 1,...,⌊n/2⌋.
283         for (GLint i = 1; i <= n / 2; i++)
284         {
285             Cartesian3 &lambda_n_minus_i = lambda[n - i];
286             Cartesian3 &rho_i            = rho    [i];
287             lambda_n_minus_i = rho_i = c_gamma[i];
288             #pragma omp parallel for
289             for (GLint j = 0; j < i; j++)
290             {
291                 Cartesian3 backward = lambda[n - j] * b_alpha_gamma(n - j, i, gamma);
292                 Cartesian3 forward  = rho    [j]    * b_gamma_beta (    j, i, gamma);
293                 #pragma omp critical (inner_points)
294                 lambda_n_minus_i -= backward;
295                 rho_i            -= forward;
296             }
297             lambda_n_minus_i /= b_alpha_gamma(n - i, i, gamma);
298             rho_i            /= b_gamma_beta (    i, i, gamma);
299         }
```


© Ágoston Róth, 2018

```
300         // Apply recursive formulas (1.41/10) and (1.44/10), i.e.,
301         //    λ_i(γ) = 1/b^(i)_{n,i}(α;α,γ) ( c_n^(i)(α) − Σ_{j=0}^{i−1} λ_j(γ) · b^(i)_{n,j}(α;α,γ) ),  i = 1,...,⌊(n−1)/2⌋,
302         // ϱ_{n−i}(γ) = 1/b^(i)_{n,n−i}(β;γ,β) ( c_n^(i)(β) − Σ_{j=0}^{i−1} ϱ_{n−j}(γ) · b^(i)_{n,n−j}(β;γ,β) ),  i = 1,...,⌊(n−1)/2⌋.
303         for (GLint i = 1; i <= (n - 1) / 2; i++)
304         {
305             Cartesian3 &lambda_i         = lambda[i];
306             Cartesian3 &rho_n_minus_i    = rho[n - i];
307
307             lambda_i         = c_alpha[i];
308             rho_n_minus_i    = c_beta [i];
309
309             #pragma omp parallel for
310             for (GLint j = 0; j < i; j++)
311             {
312                 Cartesian3 backward = rho[n - j] * b_gamma_beta (n - j, i, beta);
313                 Cartesian3 forward  = lambda[j] * b_alpha_gamma(    j, i, alpha);
314
314                 #pragma omp critical (inner_points)
315                 lambda_i      -= forward;
316                 rho_n_minus_i -= backward;
317             }
318
318             lambda_i        /= b_alpha_gamma(    i, i, alpha);
319             rho_n_minus_i   /= b_gamma_beta (n - i, i, beta);
320         }
321
321         return result;
322     }
323
323     // A redeclared and redefined inherited virtual method. If one alters the endpoints of the definition domain [α,β]
324     // both bases F_n^{α,β} and B_n^{α,β} of S_n^{α,β} have to be updated. Note that, the new interval length should also satisfy
325     // the condition 0 < β − α < ℓ'(S_n^{α,β}). The transformation matrix [t^n_{i,j}]^{n,n}_{i=0,j=0} that maps B_n^{α,β} to F_n^{α,β} is also
326     // updated.
327     GLboolean BCurve3::setDefinitionDomain(
328             GLdouble alpha, GLdouble beta,
329             bool check_for_ill_conditioned_matrices,
330             GLint expected_correct_significant_digits)
331     {
332         assert("The number of expected correct significant digits should be "
333                "non-negative!" && expected_correct_significant_digits >= 0);
334
334         if (expected_correct_significant_digits < 0 ||
335             !LinearCombination3::setDefinitionDomain(alpha, beta) ||
336             !_S.setDefinitionDomain(alpha, beta,
337                                     check_for_ill_conditioned_matrices,
338                                     expected_correct_significant_digits))
339         {
340             return GL_FALSE;
341         }
342
342         _T = SP<RealMatrix>::Default(_S.basisTransformationFromNBToOrdinary());
343
343         return (_T ? GL_TRUE : GL_FALSE);
344     }
345
345     // Given an ordinary integral curve c(u) = Σ_{i=0}^n λ_i φ_{n,i}(u), u ∈ [α,β], 0 < β − α < ℓ'(S_n^{α,β}), λ_i ∈ R^3,
346     // the method determines the coefficients [p_j]^n_{j=0} of the normalized B-basis functions {b_{n,j}(u) : u ∈ [α,β]}^n_{j=0}
347     // such that c(u) ≡ Σ_{j=0}^n p_j b_{n,j}(u), ∀u ∈ [α,β]. Note that, the control points [p_j]^n_{j=0} that have to be
348     // updated are stored in the inherited data structure ColumnMatrix<Cartesian3> LinearCombination3::_data.
349     // The method's implementation is based on Theorems 1.2/11 and 1.3/11.
350     GLboolean BCurve3::updateControlPointsForExactDescription(
351             const RowMatrix<Cartesian3> &lambda)
352     {
353         if (!_T || _S.factorizationOfTheCharacteristicPolynomialChanged())
354         {
355             return GL_FALSE;
356         }
357
357         GLint dimension = _S.dimension();
```


© Ágoston Róth, 2018

```
358         if (_T->rowCount() != dimension || lambda.columnCount() != dimension )
359         {
360             return GL_FALSE;
361         }

362         // Unknown control points [p_j]_{j=0}^n are calculated by means of [Róth, 2015a, Corollary 2.1, p. 43], i.e.,
363         // p_j = sum_{i=0}^n lambda_i t_{i,j}^n, j = 0,1,...,n. (Consider also Theorem 1.3/11.)
364         #pragma omp parallel for
365         for (GLint j = 0; j < dimension; j++)
366         {
367             Cartesian3 &p_j = _data[j];

368             p_j[0] = p_j[1] = p_j[2] = 0.0;

369             for (GLint i = 0; i < dimension; i++)
370             {
371                 p_j += lambda[i] * (*_T)(i, j);
372             }
373         }

374         return GL_TRUE;
375     }

376     // clone function required by smart pointers based on the deep copy ownership policy
377     BCurve3* BCurve3::clone() const
378     {
379         return new (nothrow) BCurve3(*this);
380     }
381 }
```

2.18 B-surfaces

B-surfaces of type (1.20/5) are represented by the class BSurface3 that is derived from the abstract base class TensorProductSurface3. It is based on Theorem 1.4/11 and on the natural extensions of Corollary 1.1/7, Lemma 1.1/8 and Theorem 1.1/10. It can be used to perform general order elevation, subdivision and to describe exactly a large family of ordinary integral surfaces of type (1.50/12). Its diagram is presented in Fig. 2.27/251, while its definition and implementation can be found in Listings 2.52/250 and 2.53/253. Note that the class redeclares and defines those pure virtual methods that are inherited from the abstract base class TensorProductSurface3.

Listing 2.52. B-surfaces (**EC/BSurfaces3.h**)

```
1  #ifndef BSURFACES3_H
2  #define BSURFACES3_H

3  #include "../Core/SmartPointers/SpecializedSmartPointers.h"
4  #include "../Core/Geometry/Surfaces/TensorProductSurfaces3.h"
5  #include "../Core/Math/Constants.h"
6  #include "../Core/Math/Matrices.h"
7  #include "ECSpaces.h"
8  #include <iostream>
9  #include <vector>
10 #include <utility>

11 using namespace std;

12 namespace cagd
13 {
14     // Let F_{n_r}^{alpha_r,beta_r} = {varphi_{n_r,i_r}(u_r) : u_r in [alpha_r,beta_r]}_{i_r=0}^{n_r}, varphi_{n_r,0} ≡ 1, 0 < beta_r - alpha_r < ell'(S_{n_r}^{alpha_r,beta_r}) be the ordinary basis
15     // and B_{n_r}^{alpha_r,beta_r} = {b_{n_r,j_r}(u_r) : u_r in [alpha_r,beta_r]}_{j_r=0}^{n_r} be the normalized B-basis of some EC vector space S_{n_r}^{alpha_r,beta_r} of
16     // functions and denote by [t_{i_r,j_r}^{n_r}]_{i_r=0,j_r=0}^{n_r,n_r} the regular square matrix that transforms B_{n_r}^{alpha_r,beta_r} to F_{n_r}^{alpha_r,beta_r}, where
17     // r = 0,1. Consider also the ordinary integral surface
18     // s(u_0,u_1) = [ s^0(u_0,u_1)  s^1(u_0,u_1)  s^2(u_0,u_1) ]^T in R^3, (u_0,u_1) in [alpha_0,beta_0] x [alpha_1,beta_1],
```


© Ágoston Róth, 2018

OrdinarySurfaceCoefficients

```
# _lambda: std::vector< std::vector< pair< RowMatrix<GLdouble>, RowMatrix<GLdouble> > > >
```

```
+ OrdinarySurfaceCoefficients(u_dimension: GLint, v_dimension: GLint, sigma: const std::vector<GLint>&)
+ <<const>> operator()(k: GLint, zeta: GLint, type: variable::Type, index: GLint): GLdouble
+ operator()(k: GLint, zeta: GLint, type: variable::Type, index: GLint): GLdouble&
+ <<const>> sigma(k: GLint): GLint
+ <<const>> dimension(type: variable::Type): GLint
+ <<const>> clone(): OrdinarySurfaceCoefficients*
```

BSurface3: public TensorProductSurface3

```
# _S: ECSpace[2]
# _T: SP<RealSquareMatrix>::Default[2]
```

```
+ BSurface3(u_space: const ECSpace&, v_space: const ECSpace&)
+ setInterval(type: variable::Type, alpha: GLdouble, beta: GLdouble,
              check_for_ill_conditioned_matrices: bool = false,
              expected_correct_significant_digits: GLint = 5): GLboolean
+ <<const>> blendingFunctionValues(type: variable::Type, parameter_value: GLdouble,
                                    values: RowMatrix<GLdouble>&): GLboolean
+ <<const>> calculateAllPartialDerivatives(maximum_order_of_derivatives: GLint,
                                           u: GLdouble, v: GLdouble, pd: PartialDerivatives&): GLboolean
+ <<const>> calculateDirectionalDerivatives(direction: variable::Type,
                                            maximum_order_of__derivatives: GLint,
                                            u: GLdouble, v: GLdouble v,
                                            d: DirectionalDerivatives&): GLboolean
+ <<const>> operator()(type: variable::Type, i: GLint, j: GLint, parameter_value: GLdouble): GLdouble
+ <<const>> performOrderElevation(type: variable::Type, a: GLdouble, b: GLdouble, order: GLint,
                                  check_for_ill_conditioned_matrices: bool = false,
                                  expected_correct_significant_digits: GLint = 5): BSurface3*
+ <<const>> performOrderElevation(type: variable::Type, zero: const CharacteristicPolynomial::Zero&,
                                  check_for_ill_conditioned_matrices: bool = false,
                                  expected_correct_significant_digits: GLint = 5): BSurface3*
+ <<const>> performSubdivision(type: variable::Type, gamma: GLdouble,
                               check_for_ill_conditioned_matrices: bool = false,
                               expected_correct_significant_digits: GLint = 5):
                               RowMatrix<SP<BSurface3>::Default>*
+ updateControlPointsForExactDescription(lambda: const OrdinarySurfaceCoefficients&): GLboolean
+ <<const>> clone(): BSurface3*
```

Fig. 2.27: Class diagrams of general B-surfaces and of ordinary surface coefficients

```
19  // where s^k(u_0,u_1) = \sum_{\zeta=0}^{\sigma_k-1} \prod_{r=0}^{1} \left( \sum_{i_r=0}^{n_r} \lambda^{k,\zeta}_{n_r,i_r} \varphi_{n_r,i_r}(u_r) \right), \sigma_k \geq 1, k = 0,1,2.

20  // The class OrdinarySurfaceCoefficients stores the pairs \left\{ \left( \left[\lambda^{k,\zeta}_{n_0,i_0}\right]_{i_0=0}^{n_0}, \left[\lambda^{k,\zeta}_{n_1,i_1}\right]_{i_1=0}^{n_1} \right) \right\}_{\zeta=0,\,k=0}^{\sigma_k-1,\,2} of row matrices
21  // that consist of those coefficients which appear in the multiplied linear combinations of the coordinate functions.
22  class OrdinarySurfaceCoefficients
23  {
24  protected:
25      //          k                 \zeta
26      std::vector< std::vector< pair<RowMatrix<GLdouble>,
27                                     RowMatrix<GLdouble> > > > _lambda;

28  public:
29      // special constructor                          n_0 + 1,                n_1 + 1
30      OrdinarySurfaceCoefficients(GLint u_dimension, GLint v_dimension,
31                                  const std::vector<GLint> &sigma); // [\sigma_k]_{k=0}^{2}

32      // based on the given variable type, returns the constant reference of either \lambda^{k,\zeta}_{n_0,i_0} or \lambda^{k,\zeta}_{n_1,i_1}, where the
33      // input variable index denotes either i_0 or i_1
34      const GLdouble& operator()(GLint k, GLint zeta, variable::Type type,
35                                 GLint index) const;

36      // based on the given variable type, returns non-constant references to either \lambda^{k,\zeta}_{n_0,i_0} or \lambda^{k,\zeta}_{n_1,i_1}, where the
37      // input variable index denotes either i_0 or i_1
38      GLdouble& operator()(GLint k, GLint zeta, variable::Type type, GLint index);

39      // returns the number of multiplied linear combination pairs appearing in the coordinate function s^k
40      GLint sigma(GLint k) const;

41      // based on the given variable type, returns the value either of n_0 + 1 or n_1 + 1
```


© Ágoston Róth, 2018

```
42          GLint dimension(variable::Type type) const;

43          // clone function required by smart pointers based on the deep copy ownership policy
44          virtual OrdinarySurfaceCoefficients* clone() const;
45      };

46      // The class BSurface3 represents tensor product of surfaces of type (1.20/5), i.e.,
47      // s_{n0,n1}(u0,u1) = sum_{i0=0}^{n0} sum_{i1=0}^{n1} p_{i0,i1} b_{n0,i0}(u0;a0,b0) b_{n1,i1}(u1;a1,b1), p_{i0,i1} = [p^k_{i0,i1}]_{k=0}^{2} in R^3
48      class BSurface3: public TensorProductSurface3
49      {
50      protected:
51          ECSpace                    _S[2]; // EC spaces S^{ar,br}_{nr}, r = 0,1
52          SP<RealMatrix>::Default    _T[2]; // transformation matrices [t^{nr}_{ir,jr}]_{ir=0,..,nr; jr=0,..,nr}, r = 0,1

53      public:
54          // special constructor
55          BSurface3(const ECSpace &u_space, const ECSpace &v_space);

56          // Inherited virtual method that has to be redeclared and redefined, since if one alters the endpoints of
57          // the definition domain [ar,br] both bases F^{ar,br}_{nr} and B^{ar,br}_{nr} of S^{ar,br}_{nr} have to be updated. Note that,
58          // the new interval length should also satisfy the condition 0 < br - ar < l'(S^{ar,br}_{nr}).
59          GLboolean setInterval(
60                  variable::Type type, GLdouble alpha, GLdouble beta,
61                  bool check_for_ill_conditioned_matrices = false,
62                  GLint expected_correct_significant_digits = 5);

63          // inherited pure virtual methods that have to be redeclared and defined
64          GLboolean blendingFunctionValues(
65                  variable::Type type, GLdouble parameter_value,
66                  RowMatrix<GLdouble> &values) const;

67          GLboolean calculateAllPartialDerivatives(
68                  GLint maximum_order_of_partial_derivatives,
69                  GLdouble u, GLdouble v, PartialDerivatives& pd) const;

70          GLboolean calculateDirectionalDerivatives(
71                  variable::Type direction,
72                  GLint maximum_order_of_directional_derivatives, GLdouble u, GLdouble v,
73                  DirectionalDerivatives &d) const;

74          // const and non-const indexing operators of the control points
75          using TensorProductSurface3::operator ();

76          // Overloaded function operator that evaluates the derivative b^{(j)}_{nr,ir}(ur), where ur in [ar,br], ir = 0,1,...,nr,
77          // and j in N.
78          GLdouble operator ()(
79                  variable::Type type, GLint function_index,
80                  GLint differentiation_order, GLdouble parameter_value) const;

81          // Overloaded member functions for general order elevation. In order to generate a higher dimensional EC
82          // space S^{ar,br}_{nr+q} that fulfills the condition S^{ar,br}_{nr} subset S^{ar,br}_{nr+q} where q >= 1 and 0 < br - ar < min{l'(S^{ar,br}_{nr}),
83          // l'(S^{ar,br}_{nr+q})} one has either to define a new complex root z = a +- ib of multiplicity m >= 1, or to specify
84          // an existing root with a multiplicity that is greater than its former value m' >= 1. If z in C \ R, then
85          // q = 2(m - m'), otherwise q = m - m', where m' = 0 whenever z denotes a nonexisting former root.
86          // Their implementations are based on the extension of Lemma 1.1/8.
87          BSurface3* performOrderElevation(
88                  variable::Type type,
89                  GLdouble a, GLdouble b, GLint multiplicity,
90                  bool check_for_ill_conditioned_matrices = false,
91                  GLint expected_correct_significant_digits = 5) const;

92          BSurface3* performOrderElevation(
93                  variable::Type type,
94                  const CharacteristicPolynomial::Zero zero,
95                  bool check_for_ill_conditioned_matrices = false,
96                  GLint expected_correct_significant_digits = 5) const;

97          // Using the extension of the general B-algorithm formulated in Theorem 1.1/10, the method subdivides the
98          // given B-surface along the specified direction into two patches of the same order at the parameter value
```


© Ágoston Róth, 2018

```
99          // γ ∈ (α_r, β_r). In case of success, the output is a nonzero raw pointer to a 2-element row matrix that stores
100         // 2 smart pointers (based on the deep copy ownership policy) that point to the left and right patches of the
101         // subdivided surface.
102         RowMatrix<SP<BSurface3>::Default>* performSubdivision(
103                 variable::Type type,
104                 GLdouble gamma,
105                 bool check_for_ill_conditioned_matrices = false,
106                 GLint expected_correct_significant_digits = 5) const;
107         // Given an ordinary integral surface of type (1.50/12) the method updates the control points [p_{i0,i1}]_{i0=0,i1=0}^{n0,n1}
108         // of the B-surface (1.20/5) in order to ensure control point based exact description. The ordinary surface
109         // (1.50/12) is described by an instance of the class OrdinarySurfaceCoefficients, while the control points
110         // [p_{i0,i1}]_{i0=0,i1=0}^{n0,n1} that have to be updated are stored in the inherited data structure
111         // Matrix<Cartesian3> TensorProductSurface3::_data. The method is based on formulas of Theorem 1.4/11.
112         GLboolean updateControlPointsForExactDescription(
113                 const OrdinarySurfaceCoefficients &lambda);
114         // redeclared clone function required by smart pointers based on the deep copy ownership policy
115         virtual BSurface3* clone() const;
116     };
117 }
118 #endif // BSURFACES3_H
```

Listing 2.53. B-surfaces (**EC/BSurfaces3.cpp**)

```
1  #include "BSurfaces3.h"
2  #include "../Core/Exceptions.h"
3  #include "BCurves3.h"
4  #include <algorithm>
5  #include <cmath>
6  #include <new>
7  using namespace std;
8  namespace cagd
9  {
10     // special constructor
11     OrdinarySurfaceCoefficients::OrdinarySurfaceCoefficients(
12             GLint u_dimension, GLint v_dimension, const std::vector<GLint> &sigma)
13     {
14         if (sigma.size() != 3)
15         {
16             throw Exception("The vector sigma should contain 3 elements!");
17         }
18         for (GLint k = 0; k < 3; k++)
19         {
20             if (sigma[k] <= 0)
21             {
22                 throw Exception("Each element of the vector should be positive "
23                                 "integers!");
24             }
25         }
26         if (!u_dimension || !v_dimension)
27         {
28             throw Exception("Dimensions of underlying EC spaces should be at least 1 in "
29                             "all directions!");
30         }
31         _lambda.resize(3);
32         for (GLint k = 0; k < 3; k++)
33         {
34             for (GLint zeta = 0; zeta < sigma[k]; zeta++)
35             {
36                 _lambda[k].push_back(pair< RowMatrix<GLdouble>, RowMatrix<GLdouble> >(
37                     RowMatrix<GLdouble>(u_dimension), RowMatrix<GLdouble>(v_dimension)));
```


© Ágoston Róth, 2018

```
38              }
39          }
40      }

41      // based on the given variable type, returns the constant reference of either λ^{k,ζ}_{n0,i0} or λ^{k,ζ}_{n1,i1}, where the
42      // input variable index denotes either i0 or i1
43      const GLdouble& OrdinarySurfaceCoefficients::operator()(
44              GLint k, GLint zeta, variable::Type type, GLint index) const
45      {
46          assert("The given coordinate function index is out of bounds!" &&
47                  (k >= 0 && k < (GLint)_lambda.size()));

48          assert("The given summation term index is out of bounds!" &&
49                  (zeta >= 0 && zeta < (GLint)_lambda[k].size()));

50          assert("The given coefficient index is out of bounds!" &&
51                  (index >= 0 && (type == variable::U ?
52                                  index < _lambda[k][zeta].first.columnCount() :
53                                  index < _lambda[k][zeta].second.columnCount())));

54          return (type == variable::U ? _lambda[k][zeta].first[index] :
55                                        _lambda[k][zeta].second[index]);
56      }

57      // based on the given variable type, returns non-constant references to either λ^{k,ζ}_{n0,i0} or λ^{k,ζ}_{n1,i1}, where the
58      // input variable index denotes either i0 or i1
59      GLdouble& OrdinarySurfaceCoefficients::operator()(
60              GLint k, GLint zeta, variable::Type type, GLint index)
61      {
62          assert("The given coordinate function index is out of bounds!" &&
63                  (k >= 0 && k < (GLint)_lambda.size()));

64          assert("The given summation term index is out of bounds!" &&
65                  (zeta >= 0 && zeta < (GLint)_lambda[k].size()));

66          assert("The given coefficient index is out of bounds!" &&
67                  (index >= 0 && (type == variable::U ?
68                                  index < _lambda[k][zeta].first.columnCount() :
69                                  index < _lambda[k][zeta].second.columnCount())));

70          return (type == variable::U ? _lambda[k][zeta].first[index] :
71                                        _lambda[k][zeta].second[index]);
72      }

73      // returns the number of multiplied linear combination pairs appearing in the coordinate function s^k
74      GLint OrdinarySurfaceCoefficients::sigma(GLint k) const
75      {
76          assert("The given coordinate function index is out of bounds!" &&
77                  (k >= 0 && k < (GLint)_lambda.size()));

78          return (GLint)_lambda[k].size();
79      }

80      // based on the given variable type, returns the value either of n0 + 1 or n1 + 1
81      GLint OrdinarySurfaceCoefficients::dimension(variable::Type type) const
82      {
83          return (type == variable::U ? _lambda[0][0].first.columnCount() :
84                                        _lambda[0][0].second.columnCount());
85      }

86      // clone function required by smart pointers based on the deep copy ownership policy
87      OrdinarySurfaceCoefficients* OrdinarySurfaceCoefficients::clone() const
88      {
89          return new (nothrow) OrdinarySurfaceCoefficients(*this);
90      }

91      // special constructor
92      BSurface3::BSurface3(const ECSpace &u_space, const ECSpace &v_space):
93          TensorProductSurface3(
94                  u_space.alpha(), u_space.beta(),
95                  v_space.alpha(), v_space.beta(),
96                  u_space.dimension(), v_space.dimension())
97      {
```


© Ágoston Róth, 2018

```
98          _S[variable::U] = u_space;
99          _S[variable::V] = v_space;
100
101         for (GLint i = variable::U; i <= variable::V; i++)
102         {
103             _T[i] = SP<RealMatrix>::Default(
104                     _S[i].basisTransformationFromNBToOrdinary());
105
106             if (!_T[i])
107             {
108                 throw Exception("One of the transformation matrices that maps the "
109                                 "normalized B-basis of an EC space to its ordinary "
110                                 "basis could not be created!");
111             }
112         }
113     }
114
115     // Inherited virtual method that has to be redeclared and redefined, since if one alters the endpoints of
116     // the definition domain [α_r, β_r] both bases F_{n_r}^{α_r,β_r} and B_{n_r}^{α_r,β_r} of S_{n_r}^{α_r,β_r} have to be updated. Note that,
117     // the new interval length should also satisfy the condition 0 < β_r − α_r < ℓ'(S_{n_r}^{α_r,β_r}). The corresponding
118     // basis transformation matrix [t_{i_r,j_r}^{n_r}]_{i_r=0,j_r=0}^{n_r,n_r} that maps B_{n_r}^{α_r,β_r} to F_{n_r}^{α_r,β_r} will also be updated.
119     GLboolean BSurface3::setInterval(
120             variable::Type type,
121             GLdouble alpha, GLdouble beta,
122             bool check_for_ill_conditioned_matrices,
123             GLint expected_correct_significant_digits)
124     {
125         assert("The number of expected correct significant digits should be "
126                "non-negative!" && expected_correct_significant_digits >= 0);
127
128         if (expected_correct_significant_digits < 0 ||
129             !TensorProductSurface3::setInterval(type, alpha, beta) ||
130             !_S[type].setDefinitionDomain(alpha, beta,
131                                           check_for_ill_conditioned_matrices,
132                                           expected_correct_significant_digits))
133         {
134             return GL_FALSE;
135         }
136
137         _T[type] = SP<RealMatrix>::Default(
138                 _S[type].basisTransformationFromNBToOrdinary());
139
140         return (_T[type] ? GL_TRUE : GL_FALSE);
141     }
142
143     // The next tree member functions are inherited pure virtual methods that have to be redeclared and defined.
144
145     // Evaluates all normalized B-basis function values {b_{n_r,i_r}(u_r)}_{i_r=0}^{n_r}, where u_r ∈ [α_r, β_r] is specified by the
146     // variable 'parameter_value' and, based on the variable 'type', r is either 0, or 1. In case of success, the obtained
147     // values will be stored in the real row matrix referenced by the variable 'values'.
148     GLboolean BSurface3::blendingFunctionValues(
149             variable::Type type, GLdouble parameter_value,
150             RowMatrix<GLdouble> &values) const
151     {
152         if (parameter_value < _S[type].alpha() || parameter_value > _S[type].beta())
153         {
154             values.resizeColumns(0);
155             return GL_FALSE;
156         }
157
158         GLint dimension = _S[type].dimension();
159
160         values.resizeColumns(dimension);
161
162         #pragma omp parallel for
163         for (GLint i = 0; i < dimension; i++)
164         {
165             values[i] = _S[type](ECSpace::B_BASIS, i, 0, parameter_value);
166         }
167
168         return GL_TRUE;
169     }
```


© Ágoston Róth, 2018

```
158 // Calculates the partial derivatives [∂^r/(∂u_0^{r-j} ∂u_1^j) s_{n0,n1}(u0,u1)]_{j=0}^{r} for all r = 0,...,ρ, where ρ ∈ ℕ denotes
159 // the maximum order of partial derivatives. Parameter values u0 ∈ [α0,β0] and u1 ∈ [α1,β1] correspond to
160 // the input variables 'u' and 'v', respectively, while, in case of success, the obtained partial derivatives will
161 // be stored in a triangular matrix referenced by the variable 'd'.
162 GLboolean BSurface3::calculateAllPartialDerivatives(
163         GLint maximum_order_of_partial_derivatives,
164         GLdouble u, GLdouble v, PartialDerivatives& d) const
165 {
166     assert("The given maximum order of partial derivatives should be "
167            "non-negative!" && maximum_order_of_partial_derivatives >= 0);
168     if (maximum_order_of_partial_derivatives < 0 ||
169         u < _S[variable::U].alpha() || u > _S[variable::U].beta() ||
170         v < _S[variable::V].alpha() || v > _S[variable::V].beta())
171     {
172         d.resizeRows(0);
173         return GL_FALSE;
174     }
175     d.resizeRows(maximum_order_of_partial_derivatives + 1);
176     d.loadNullVectors();
177     GLint u_dimension = _S[variable::U].dimension();
178     GLint v_dimension = _S[variable::V].dimension();
179     #pragma omp parallel for
180     for (GLint order = 0; order <= maximum_order_of_partial_derivatives; order++)
181     {
182         #pragma omp parallel for
183         for (GLint j_u = order; j_u >= 0; j_u--)
184         {
185             GLint j_v = order - j_u;
186             Cartesian3 &derivative = d(order, j_v);
187             #pragma omp parallel for
188             for (GLint i_u = 0; i_u < u_dimension; i_u++)
189             {
190                 Cartesian3 aux;
191                 for (GLint i_v = 0; i_v < v_dimension; i_v++)
192                 {
193                     aux += _data(i_u, i_v) *
194                            _S[variable::V](ECSpace::B_BASIS, i_v, j_v, v);
195                 }
196                 derivative += aux * _S[variable::U](ECSpace::B_BASIS, i_u, j_u, u);
197             }
198         }
199     }
200     return GL_TRUE;
201 }
202 // Computes the partial derivatives [∂^j/∂u_r^j s_{n0,n1}(u0,u1)]_{j=0}^{ρ}, where ρ ∈ ℕ denotes the maximum order of
203 // partial derivatives. Parameter values u0 ∈ [α0,β0] and u1 ∈ [α1,β1] correspond to the input variables 'u'
204 // and 'v', respectively. The index r is determined by the variable 'direction'. In case of success, the obtained
205 // partial derivatives will be stored in a column matrix referenced by the variable 'd'.
206 GLboolean BSurface3::calculateDirectionalDerivatives(
207         variable::Type direction, GLint maximum_order_of_directional_derivatives,
208         GLdouble u, GLdouble v, DirectionalDerivatives &d) const
209 {
210     assert("The given maximum order of directional derivatives should be "
211            "non-negative!" && maximum_order_of_directional_derivatives >= 0);
212     if (maximum_order_of_directional_derivatives < 0)
213     {
214         d.resizeRows(0);
215         return GL_FALSE;
216     }
```


© Ágoston Róth, 2018

```
217         if (direction == variable::U)
218         {
219             if (u < _S[variable::U].alpha() || u > _S[variable::U].beta())
220             {
221                 d.resizeRows(0);
222                 return GL_FALSE;
223             }
224             d.resizeRows(maximum_order_of_directional_derivatives + 1);
225             d.loadNullVectors();
226             GLint u_dimension = _S[variable::U].dimension();
227             GLint v_dimension = _S[variable::V].dimension();
228             #pragma omp parallel for
229             for (GLint order = 0; order <= maximum_order_of_directional_derivatives;
230                  order++)
231             {
232                 Cartesian3 &u_derivative = d[order];
233                 #pragma omp parallel for
234                 for (GLint i = 0; i < u_dimension; i++)
235                 {
236                     Cartesian3 aux;
237                     for (GLint j = 0; j < v_dimension; j++)
238                     {
239                         aux += _data(i, j) * _S[variable::V](ECSpace::B_BASIS, j, 0, v);
240                     }
241                     u_derivative += aux * _S[variable::U](ECSpace::B_BASIS, i, order, u);
242                 }
243             }
244             return GL_TRUE;
245         }
246         else
247         {
248             if (v < _S[variable::V].alpha() || v > _S[variable::U].beta())
249             {
250                 d.resizeRows(0);
251                 return GL_FALSE;
252             }
253             d.resizeRows(maximum_order_of_directional_derivatives + 1);
254             d.loadNullVectors();
255             GLint u_dimension = _S[variable::U].dimension();
256             GLint v_dimension = _S[variable::V].dimension();
257             #pragma omp parallel for
258             for (GLint order = 0; order <= maximum_order_of_directional_derivatives;
259                  order++)
260             {
261                 Cartesian3 &v_derivative = d[order];
262                 #pragma omp parallel for
263                 for (GLint i = 0; i < u_dimension; i++)
264                 {
265                     Cartesian3 aux;
266                     for (GLint j = 0; j < v_dimension; j++)
267                     {
268                         aux += _data(i, j) *
269                                _S[variable::V](ECSpace::B_BASIS, j, order, v);
270                     }
271                     v_derivative += aux * _S[variable::U](ECSpace::B_BASIS, i, 0, u);
272                 }
273             }
274             return GL_TRUE;
275         }
```


© Ágoston Róth, 2018

```
276     }
277     // Overloaded function operator that evaluates the derivative b^(j)_{n_r,i_r}(u_r), where u_r ∈ [α_r, β_r], i_r = 0, 1, ..., n_r,
278     // and j ∈ ℕ.
279     GLdouble BSurface3::operator ()(
280             variable::Type type, GLint function_index,
281             GLint differentiation_order, GLdouble parameter_value) const
282     {
283         assert("The given B-basis function index is out of bounds!" &&
284                (function_index >= 0 && function_index < _S[type].dimension()));
285         assert("The given differentiation order should be non-negative!" &&
286                differentiation_order >= 0);
287         return _S[type](ECSpace::B_BASIS, function_index,
288                         differentiation_order, parameter_value);
289     }
290     // Overloaded member functions for general order elevation. In order to generate a higher dimensional EC
291     // space S^{α_r,β_r}_{n_r+q} that fulfills the condition S^{α_r,β_r}_{n_r} ⊂ S^{α_r,β_r}_{n_r+q} where q ≥ 1 and 0 < β_r − α_r < min{ℓ'(S^{α_r,β_r}_{n_r})},
292     // ℓ'(S^{α_r,β_r}_{n_r+q})} one has either to define a new complex root z = a ± ib of multiplicity m ≥ 1, or to specify
293     // an existing root with a multiplicity that is greater than its former value m' ≥ 1. If z ∈ ℂ \ ℝ, then
294     // q = 2(m − m'), otherwise q = m − m', where m' = 0 whenever z denotes a nonexisting former root.
295     // Their implementations are based on the extension of Lemma 1.1/8.
296     BSurface3* BSurface3::performOrderElevation(
297             variable::Type type,
298             GLdouble a, GLdouble b, GLint multiplicity,
299             bool check_for_ill_conditioned_matrices,
300             GLint expected_correct_significant_digits) const
301     {
302         assert("The given order of multiplicity should be non-negative!" &&
303                multiplicity >= 0);
304         assert("The number of expected correct significant digits should be "
305                "non-negative!" && expected_correct_significant_digits >= 0);
306         return performOrderElevation(type,
307                                      CharacteristicPolynomial::Zero(a, b, multiplicity),
308                                      check_for_ill_conditioned_matrices,
309                                      expected_correct_significant_digits);
310     }
311     BSurface3* BSurface3::performOrderElevation(
312             variable::Type type,
313             const CharacteristicPolynomial::Zero zero,
314             bool check_for_ill_conditioned_matrices,
315             GLint expected_correct_significant_digits) const
316     {
317         assert("The number of expected correct significant digits should be "
318                "non-negative!" && expected_correct_significant_digits >= 0);
319         if (expected_correct_significant_digits < 0)
320         {
321             return nullptr;
322         }
323         BSurface3* result = clone();
324         if (!result)
325         {
326             return nullptr;
327         }
328         if (type == variable::U)
329         {
330             try
331             {
332                 if (!result->_S[variable::U].insertZero(
333                         zero,
334                         true,
335                         check_for_ill_conditioned_matrices,
336                         expected_correct_significant_digits))
337                 {
```


© Ágoston Róth, 2018

```
338                     delete result, result = nullptr;
339                     return nullptr;
340                 }
341             }
342             catch (Exception &e)
343             {
344                 delete result, result = nullptr;
345                 throw e;
346             }

347             GLint u_old_dimension      = _S[variable::U].dimension();
348             GLint u_elevated_dimension = result->_S[variable::U].dimension();

349             if (u_old_dimension == u_elevated_dimension)
350             {
351                 return result;
352             }

353             GLdouble u_alpha = _S[variable::U].alpha(), u_beta = _S[variable::U].beta();

354             result->_data.resizeRows(u_elevated_dimension);

355             if (u_elevated_dimension - u_old_dimension == 1)
356             {
357                 GLint n = u_old_dimension - 1;

358                 #pragma omp parallel for
359                 for (GLint column = 0; column < _S[variable::V].dimension(); column++)
360                 {
361                     (*result)(0, column)                = _data(0, column);
362                     (*result)(u_old_dimension, column) = _data(n, column);

363                     #pragma omp parallel for
364                     for (GLint i = 1; i <= n / 2; i++)
365                     {
366                         GLdouble ratio       = (*this)(variable::U, i, i, u_alpha) /
367                                                (*result)(variable::U, i, i, u_alpha);
368                         (*result)(i, column) = (1 - ratio) * _data(i - 1, column) +
369                                                ratio * _data(i, column);
370                     }

371                     #pragma omp parallel for
372                     for (GLint i = 1; i <= u_old_dimension / 2; i++)
373                     {
374                         GLdouble ratio                              =
375                                 (*this)(variable::U, n - i, i, u_beta) /
376                                 (*result)(variable::U, u_old_dimension - i, i, u_beta);
377                         (*result)(u_old_dimension - i, column) =
378                                 ratio * _data(n - i, column) +
379                                 (1 - ratio) * _data(u_old_dimension - i, column);
380                     }
381                 }

382                 return result;
383             }

384             GLint v_dimension = _S[variable::V].dimension();
385             GLdouble v_alpha  = _S[variable::V].alpha(), v_beta = _S[variable::V].beta();

386             RowMatrix<GLdouble>      u_knot_vector(u_elevated_dimension);
387             ColumnMatrix<GLdouble>   v_knot_vector(v_dimension);

388             Matrix<Cartesian3>       sample(u_elevated_dimension, v_dimension);

389             GLdouble u_step = (u_beta - u_alpha) / (u_elevated_dimension - 1);
390             GLdouble v_step = (v_beta - v_alpha) / (v_dimension - 1);

391             GLboolean sampling_aborted = GL_FALSE;

392             #pragma omp parallel for
393             for (GLint i_j = 0; i_j < u_elevated_dimension * v_dimension; i_j++)
394             {
395                 #pragma omp flush (sampling_aborted)
396                 if (!sampling_aborted)
```


© Ágoston Róth, 2018

```
397                 {
398                     GLint i = i_j / v_dimension;
399                     GLint j = i_j % v_dimension;
400                     u_knot_vector[i] = min(u_alpha + i * u_step, u_beta);
401                     v_knot_vector[j] = min(v_alpha + j * v_step, v_beta);
402                     PartialDerivatives d;
403                     if (!calculateAllPartialDerivatives(
404                             0, u_knot_vector[i], v_knot_vector[j], d))
405                     {
406                         sampling_aborted = GL_TRUE;
407                         #pragma omp flush (sampling_aborted)
408                     }
409                     sample(i, j) = d(0, 0);
410                 }
411             }
412             if (sampling_aborted ||
413                 !result->updateDataForInterpolation(
414                         u_knot_vector, v_knot_vector, sample))
415             {
416                 delete result, result = nullptr;
417             }
418             return result;
419         }
420         else
421         {
422             try
423             {
424                 if (!result->_S[variable::V].insertZero(
425                         zero,
426                         true,
427                         check_for_ill_conditioned_matrices,
428                         expected_correct_significant_digits))
429                 {
430                     delete result, result = nullptr;
431                     return nullptr;
432                 }
433             }
434             catch(Exception &e)
435             {
436                 delete result, result = nullptr;
437                 throw e;
438             }
439             GLint v_old_dimension      = _S[variable::V].dimension();
440             GLint v_elevated_dimension = result->_S[variable::V].dimension();
441             if (v_old_dimension == v_elevated_dimension)
442             {
443                 return result;
444             }
445             GLdouble v_alpha = _S[variable::V].alpha(), v_beta = _S[variable::V].beta();
446             result->_data.resizeColumns(v_elevated_dimension);
447             if (v_elevated_dimension - v_old_dimension == 1)
448             {
449                 GLint n = v_old_dimension - 1;
450                 #pragma omp parallel for
451                 for (GLint row = 0; row < _S[variable::U].dimension(); row++)
452                 {
453                     (*result)(row, 0)               = _data(row, 0);
454                     (*result)(row, v_old_dimension) = _data(row, n);
455                     #pragma omp parallel for
456                     for (GLint i = 1; i <= n / 2; i++)
457                     {
458                         GLdouble ratio    = (*this)(variable::V, i, i, v_alpha) /
```


© Ágoston Róth, 2018

```
459                                         (*result)(variable::V, i, i, v_alpha);
460                         (*result)(row, i) = (1 - ratio) * _data(row, i - 1) +
461                                             ratio * _data(row, i);
462                     }
463                     #pragma omp parallel for
464                     for (GLint i = 1; i <= v_old_dimension / 2; i++)
465                     {
466                         GLdouble ratio                          =
467                                 (*this)(variable::V, n - i, i, v_beta) /
468                                 (*result)(variable::V, v_old_dimension - i, i, v_beta);
469                         (*result)(row, v_old_dimension - i) =
470                                 ratio * _data(row, n - i) +
471                                 (1 - ratio) * _data(row, v_old_dimension - i);
472                     }
473                 }
474                 return result;
475             }
476             GLint u_dimension  = _S[variable::U].dimension();
477             GLdouble u_alpha  = _S[variable::U].alpha(), u_beta = _S[variable::U].beta();
478             RowMatrix<GLdouble>      u_knot_vector(u_dimension);
479             ColumnMatrix<GLdouble>   v_knot_vector(v_elevated_dimension);
480             Matrix<Cartesian3>       sample(u_dimension, v_elevated_dimension);
481             GLdouble u_step = (u_beta - u_alpha) / (u_dimension - 1);
482             GLdouble v_step = (v_beta - v_alpha) / (v_elevated_dimension - 1);
483             GLboolean sampling_aborted = GL_FALSE;
484             #pragma omp parallel for
485             for (GLint ij = 0; ij < u_dimension * v_elevated_dimension; ij++)
486             {
487                 #pragma omp flush (sampling_aborted)
488                 if (!sampling_aborted)
489                 {
490                     GLint i = ij / v_elevated_dimension;
491                     GLint j = ij % v_elevated_dimension;
492                     u_knot_vector[i] = min(u_alpha + i * u_step, u_beta);
493                     v_knot_vector[j] = min(v_alpha + j * v_step, v_beta);
494                     PartialDerivatives d;
495                     if (!calculateAllPartialDerivatives(
496                             0, u_knot_vector[i], v_knot_vector[j], d))
497                     {
498                         sampling_aborted = GL_TRUE;
499                         #pragma omp flush (sampling_aborted)
500                     }
501                     else
502                     {
503                         sample(i, j) = d(0, 0);
504                     }
505                 }
506             }
507             if (sampling_aborted ||
508                 !result->updateDataForInterpolation(
509                         u_knot_vector, v_knot_vector, sample))
510             {
511                 delete result, result = nullptr;
512             }
513             return result;
514         }
515     }
516     // Using the extension of the general B-algorithm formulated in Theorem 1.1/10, the method subdivides the
517     // given B-surface along the specified direction into two patches of the same order at the parameter value
518     // γ ∈ (α_r, β_r). In case of success, the output is a nonzero raw pointer to a 2-element row matrix that stores
519     // 2 smart pointers (based on the deep copy ownership policy) that point to the left and right patches of the
```


© Ágoston Róth, 2018

```
520     // subdivided surface.
521     RowMatrix<SP<BSurface3>::Default>* BSurface3::performSubdivision(
522             variable::Type type,
523             GLdouble gamma,
524             bool check_for_ill_conditioned_matrices,
525             GLint expected_correct_significant_digits) const
526     {
527         assert("The number of expected correct significant digits should be "
528                "non-negative!" && expected_correct_significant_digits >= 0);
529         if (expected_correct_significant_digits < 0)
530         {
531             return nullptr;
532         }
533         if (type == variable::U)
534         {
535             GLdouble u_alpha = _S[variable::U].alpha(), u_beta = _S[variable::U].beta();
536             if (gamma <= u_alpha || gamma >= u_beta)
537             {
538                 return nullptr;
539             }
540             RowMatrix<SP<BSurface3>::Default>* result =
541                     new (nothrow) RowMatrix<SP<BSurface3>::Default>(2);
542             if (!result)
543             {
544                 return nullptr;
545             }
546             ECSpace lambda_space(_S[variable::U]), rho_space(_S[variable::U]);
547             try
548             {
549                 if (!lambda_space.setDefinitionDomain(
550                         u_alpha, gamma,
551                         check_for_ill_conditioned_matrices,
552                         expected_correct_significant_digits) ||
553                     !rho_space.setDefinitionDomain(
554                             gamma, u_beta,
555                             check_for_ill_conditioned_matrices,
556                             expected_correct_significant_digits))
557                 {
558                     delete result, result = nullptr;
559                     return nullptr;
560                 }
561             }
562             catch (Exception &e)
563             {
564                 delete result, result = nullptr;
565                 throw e;
566             }
567             (*result)[0] = SP<BSurface3>::Default(
568                     new (nothrow) BSurface3(lambda_space, _S[variable::V]));
569             (*result)[1] = SP<BSurface3>::Default(
570                     new (nothrow) BSurface3(rho_space, _S[variable::V]));
571             if (!(*result)[0] || !(*result)[1])
572             {
573                 delete result, result = nullptr;
574                 return nullptr;
575             }
576             SP<BSurface3>::Default &L = (*result)[0];
577             SP<BSurface3>::Default &R = (*result)[1];
578             GLint u_dimension = _S[variable::U].dimension();
579             GLint u_n         = u_dimension - 1;
580             GLint half_u_n    = u_n / 2;
```


© Ágoston Róth, 2018

```
581             GLint  v_dimension = _S[variable::V].dimension();
582             GLint v_n          = v_dimension - 1;
583             for (GLint l = 0; l <= v_n; l++)
584             {
585                 ColumnMatrix<Cartesian3> c_alpha(half_u_n + 1),
586                                          c_gamma(half_u_n + 1),
587                                          c_beta(half_u_n + 1);
588                 for (GLint j = 0; j <= half_u_n; j++)
589                 {
590                     #pragma omp parallel for
591                     for (GLint i = 0; i < u_dimension; i++)
592                     {
593                         const Cartesian3 &cp = _data(i, l);
594                         GLdouble b_i_j_u_alpha = _S[variable::U](ECSpace::B_BASIS,
595                                                                  i, j, u_alpha);
596                         GLdouble b_i_j_gamma   = _S[variable::U](ECSpace::B_BASIS,
597                                                                  i, j, gamma);
598                         GLdouble b_i_j_u_beta  = _S[variable::U](ECSpace::B_BASIS,
599                                                                  i, j, u_beta);
600                         #pragma omp critical (c_derivatives_alpha_gamma_beta)
601                         c_alpha[j] += cp * b_i_j_u_alpha;
602                         c_gamma[j] += cp * b_i_j_gamma;
603                         c_beta [j] += cp * b_i_j_u_beta;
604                     }
605                 }
606                 L->_data(u_n, l) = c_gamma[0], L->_data(0, l)   = _data(0,   l);
607                 R->_data(0, l)   = c_gamma[0], R->_data(u_n, l) = _data(u_n, l);
608                 for (GLint i = 1; i <= half_u_n; i++)
609                 {
610                     Cartesian3 &lambda_u_n_minus_i = L->_data(u_n - i, l);
611                     Cartesian3 &rho_i              = R->_data(i, l);
612                     lambda_u_n_minus_i = rho_i = c_gamma[i];
613                     #pragma omp parallel for
614                     for (GLint j = 0; j < i; j++)
615                     {
616                         Cartesian3 backward = L->_data(u_n - j, l);
617                         Cartesian3 forward  = R->_data(j, l);
618                         backward *= lambda_space(ECSpace::B_BASIS, u_n - j, i,
619                                                  gamma);
620                         forward  *= rho_space   (ECSpace::B_BASIS,       j, i,
621                                                  gamma);
622                         #pragma omp critical (subdivision_points_I)
623                         lambda_u_n_minus_i -= backward;
624                         rho_i              -= forward;
625                     }
626                     #pragma omp critical (last_division)
627                     lambda_u_n_minus_i /= lambda_space(ECSpace::B_BASIS, u_n - i, i,
628                                                        gamma);
629                     rho_i              /= rho_space   (ECSpace::B_BASIS,       i, i,
630                                                        gamma);
631                 }
632                 for (GLint i = 1; i <= (u_n - 1) / 2; i++)
633                 {
634                     Cartesian3 &lambda_i         = L->_data(i, l);
635                     Cartesian3 &rho_u_n_minus_i  = R->_data(u_n - i, l);
636                     lambda_i         = c_alpha[i];
637                     rho_u_n_minus_i  = c_beta [i];
638                     #pragma omp parallel for
639                     for (GLint j = 0; j < i; j++)
640                     {
```


© Ágoston Róth, 2018

```
641                         Cartesian3 backward = R->_data(u_n - j, l);
642                         Cartesian3 forward  = L->_data(j, l);
643                         backward *= rho_space   (ECSpace::B_BASIS, u_n - j, i, u_beta);
644                         forward  *= lambda_space(ECSpace::B_BASIS,       j, i, u_alpha);
645                         #pragma omp critical (subdivision_points_II)
646                         lambda_i       -= forward;
647                         rho_u_n_minus_i -= backward;
648                     }
649                     #pragma omp critical (last_division)
650                     lambda_i         /= lambda_space(ECSpace::B_BASIS,       i, i,
651                                                      u_alpha);
652                     rho_u_n_minus_i  /= rho_space   (ECSpace::B_BASIS, u_n - i, i,
653                                                      u_beta);
654                 }
655             }
656             return result;
657         }
658         else
659         {
660             GLdouble v_alpha = _S[variable::V].alpha(), v_beta = _S[variable::V].beta();
661             if (gamma <= v_alpha || gamma >= v_beta)
662             {
663                 return nullptr;
664             }
665             RowMatrix<SP<BSurface3>::Default>* result =
666                     new (nothrow) RowMatrix<SP<BSurface3>::Default>(2);
667             if (!result)
668             {
669                 return nullptr;
670             }
671             ECSpace lambda_space(_S[variable::V]), rho_space(_S[variable::V]);
672             try
673             {
674                 if (!lambda_space.setDefinitionDomain(
675                         v_alpha, gamma,
676                         check_for_ill_conditioned_matrices,
677                         expected_correct_significant_digits) ||
678                     !rho_space.setDefinitionDomain(
679                             gamma, v_beta,
680                             check_for_ill_conditioned_matrices,
681                             expected_correct_significant_digits))
682                 {
683                     delete result, result = nullptr;
684                     return nullptr;
685                 }
686             }
687             catch (Exception &e)
688             {
689                 delete result, result = nullptr;
690                 throw e;
691             }
692             (*result)[0] = SP<BSurface3>::Default(
693                     new (nothrow) BSurface3(_S[variable::U], lambda_space));
694             (*result)[1] = SP<BSurface3>::Default(
695                     new (nothrow) BSurface3(_S[variable::U], rho_space));
696             if (!(*result)[0] || !(*result)[1])
697             {
698                 delete result, result = nullptr;
699                 return nullptr;
700             }
```


© Ágoston Róth, 2018

```
701             SP<BSurface3>::Default &L = (*result)[0];
702             SP<BSurface3>::Default &R = (*result)[1];


703             GLint v_dimension = _S[variable::V].dimension();
704             GLint v_n         = v_dimension - 1;
705             GLint half_v_n    = v_n / 2;

706             GLint u_dimension = _S[variable::U].dimension();
707             GLint u_n         = u_dimension - 1;

708             for (GLint k = 0; k <= u_n; k++)
709             {
710                 ColumnMatrix<Cartesian3> c_alpha(half_v_n + 1),
711                                          c_gamma(half_v_n + 1),
712                                          c_beta(half_v_n + 1);

713                 for (GLint j = 0; j <= half_v_n; j++)
714                 {
715                     #pragma omp parallel for
716                     for (GLint i = 0; i < v_dimension; i++)
717                     {
718                         const Cartesian3 &cp    = _data(k, i);
719                         GLdouble b_i_j_v_alpha = _S[variable::V](ECSpace::B_BASIS, i, j,
720                                                                  v_alpha);
721                         GLdouble b_i_j_gamma   = _S[variable::V](ECSpace::B_BASIS, i, j,
722                                                                  gamma);
723                         GLdouble b_i_j_v_beta  = _S[variable::V](ECSpace::B_BASIS, i, j,
724                                                                  v_beta);

725                         #pragma omp critical
726                         c_alpha[j] += cp * b_i_j_v_alpha;
727                         c_gamma[j] += cp * b_i_j_gamma;
728                         c_beta [j] += cp * b_i_j_v_beta;
729                     }
730                 }

731                 L->_data(k, v_n) = c_gamma[0], L->_data(k, 0)   = _data(k, 0);
732                 R->_data(k, 0)   = c_gamma[0], R->_data(k, v_n) = _data(k, v_n);

733                 for (GLint i = 1; i <= half_v_n; i++)
734                 {
735                     Cartesian3 &lambda_v_n_minus_i = L->_data(k, v_n - i);
736                     Cartesian3 &rho_i              = R->_data(k, i);

737                     lambda_v_n_minus_i = rho_i = c_gamma[i];

738                     #pragma omp parallel for
739                     for (GLint j = 0; j < i; j++)
740                     {
741                         Cartesian3 backward = L->_data(k, v_n - j);
742                         Cartesian3 forward  = R->_data(k, j);

743                         backward *= lambda_space(ECSpace::B_BASIS, v_n - j, i, gamma);
744                         forward  *= rho_space   (ECSpace::B_BASIS,       j, i, gamma);

745                         #pragma omp critical (subdivision_points_I)
746                         lambda_v_n_minus_i -= backward;
747                         rho_i              -= forward;
748                     }

749                     #pragma omp critical (last_division)
750                     lambda_v_n_minus_i /= lambda_space(ECSpace::B_BASIS, v_n - i, i,
751                                                        gamma);
752                     rho_i              /= rho_space   (ECSpace::B_BASIS,       i, i,
753                                                        gamma);
754                 }

755                 for (GLint i = 1; i <= (v_n - 1) / 2; i++)
756                 {
757                     Cartesian3 &lambda_i         = L->_data(k, i);
758                     Cartesian3 &rho_v_n_minus_i  = R->_data(k, v_n - i);

759                     lambda_i         = c_alpha[i];
```


© Ágoston Róth, 2018

```
760                     rho_v_n_minus_i  = c_beta [i];

761                     #pragma omp parallel for
762                     for (GLint j = 0; j < i; j++)
763                     {
764                         Cartesian3 backward = R->_data(k, v_n - j);
765                         Cartesian3 forward  = L->_data(k, j);

766                         backward *= rho_space    (ECSpace::B_BASIS, v_n - j, i, v_beta);
767                         forward  *= lambda_space(ECSpace::B_BASIS,       j, i, v_alpha);

768                         #pragma omp critical (subdivision_points_II)
769                         lambda_i        -= forward;
770                         rho_v_n_minus_i -= backward;
771                     }

772                     #pragma omp critical (last_division)
773                     lambda_i         /= lambda_space(ECSpace::B_BASIS,       i, i,
774                                                      v_alpha);
775                     rho_v_n_minus_i  /= rho_space   (ECSpace::B_BASIS, v_n - i, i,
776                                                      v_beta);
777                 }
778             }

779             return result;
780         }
781     }

782     // Given an ordinary integral surface of type (1.50/12) the method updates the control points [p_{i0,i1}]_{i0=0,i1=0}^{n0,n1}
783     // of the B-surface (1.20/5) in order to ensure control point based exact description. The ordinary surface
784     // (1.50/12) is described by an instance of the class OrdinarySurfaceCoefficients, while the control points
785     // [p_{i0,i1}]_{i0=0,i1=0}^{n0,n1} that have to be updated are stored in the inherited data structure
786     // Matrix<Cartesian3> TensorProductSurface3::_data. The method is based on formulas of Theorem 1.4/11.
787     GLboolean BSurface3::updateControlPointsForExactDescription(
788             const OrdinarySurfaceCoefficients &lambda)
789     {
790         if (!_T[variable::U] || !_T[variable::V] ||
791             _S[variable::U].dimension() != lambda.dimension(variable::U) ||
792             _S[variable::V].dimension() != lambda.dimension(variable::V))
793         {
794             return GL_FALSE;
795         }

796         #pragma omp parallel for
797         for (GLint j_0_j_1 = 0;
798               j_0_j_1 < _S[variable::U].dimension() * _S[variable::V].dimension();
799               j_0_j_1++)
800         {
801             GLint j_0 = j_0_j_1 / _S[variable::V].dimension();
802             GLint j_1 = j_0_j_1 % _S[variable::V].dimension();

803             Cartesian3 &p = _data(j_0, j_1);

804             p[0] = p[1] = p[2] = 0.0;

805             for (GLint k = 0; k < 3; k++)
806             {
807                 for (GLint zeta = 0; zeta < lambda.sigma(k); zeta++)
808                 {
809                     GLdouble u_sum = 0.0;

810                     for (GLint i_0 = 0; i_0 < _S[variable::U].dimension(); i_0++)
811                     {
812                         u_sum += lambda(k, zeta, variable::U, i_0) *
813                                  (*_T[variable::U])(i_0, j_0);
814                     }

815                     GLdouble v_sum = 0.0;

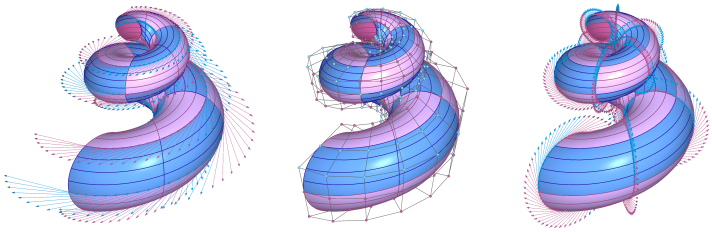
816                     for (GLint i_1 = 0; i_1 < _S[variable::V].dimension(); i_1++)
817                     {
818                         v_sum += lambda(k, zeta, variable::V, i_1) *
819                                  (*_T[variable::V])(i_1, j_1);
```


© Ágoston Róth, 2018

```
820                         }
821                         p[k] += u_sum * v_sum;
822                     }
823                 }
824             }
825             return GL_TRUE;
826         }
827         // redefined clone function required by smart pointers based on the deep copy ownership policy
828         BSurface3* BSurface3::clone() const
829         {
830             return new (std::nothrow) BSurface3(*this);
831         }
832 }
```


© Ágoston Róth, 2018

© Ágoston Róth, 2018



Defining and using specialized EC spaces • Differentiating and rendering ordinary and normalized B-basis functions • Creating, manipulating and rendering B-curves and B-surfaces • Generating B-representations of ordinary integral curves and surfaces

3 Usage examples

3.1 Defining and using specialized EC spaces

Deriving from the base class ECSpace, Listings 3.1/270 and 3.2/273 define and implement several classes that represent the following polynomial, trigonometric, hyperbolic, algebraic-trigonometric and two algebraic-exponential-trigonometric EC spaces

$$\mathbb{P}_n^{\alpha,\beta} = \langle \{1, u, \dots, u^n : u \in [\alpha, \beta]\} \rangle, \ \dim \mathbb{P}_n^{\alpha,\beta} = n+1, \ -\infty < \alpha < \beta < +\infty, \tag{3.1}$$

$$\begin{aligned} \mathbb{T}_{2n}^{\alpha,\beta} &= \langle \{1, \cos(u), \sin(u), \dots, \cos(nu), \sin(nu) : u \in [\alpha, \beta]\} \rangle, \\ &\dim \mathbb{T}_n^{\alpha,\beta} = 2n+1, \ \beta - \alpha \in (0, \pi), \end{aligned} \tag{3.2}$$

$$\begin{aligned} \mathbb{H}_{2n}^{\alpha,\beta} &= \langle \{1, e^u, e^{-u}, \dots, e^{nu}, e^{-nu} : u \in [\alpha, \beta]\} \rangle, \\ &\dim \mathbb{H}_n^{\alpha,\beta} = 2n+1, \ 0 < \beta - \alpha < +\infty, \end{aligned} \tag{3.3}$$

$$\begin{aligned} \mathbb{AT}_{n(n+2)}^{\alpha,\beta} &= \left\langle \left\{ 1, u, \dots, u^n, \{u^r \cos(ku), u^r \sin(ku)\}_{r=0,\,k=1}^{n-k,\,n} : u \in [\alpha, \beta] \right\} \right\rangle, \\ &\dim \mathbb{AT}_{n(n+2)}^{\alpha,\beta} = (n+1)^2, \end{aligned} \tag{3.4}$$

$$\begin{aligned} \mathbb{AET}_{n(2n+3)}^{\alpha,\beta} &= \left\langle \left\{ \{u^r\}_{r=0}^{n}, \{u^r e^{ku} \cos(ku), u^r e^{ku} \sin(ku)\}_{r=0,\,k=1}^{n-k,\,n}, \right.\right. \\ &\qquad \left.\left. \{u^r e^{-ku} \cos(ku), u^r e^{-ku} \sin(ku)\}_{r=0,\,k=1}^{n-k,\,n} : u \in [\alpha, \beta] \right\} \right\rangle, \\ &\dim \mathbb{AT}_{n(2n+3)}^{\alpha,\beta} = 2n^2 + 3n + 1, \end{aligned} \tag{3.5}$$

and

$$\begin{aligned} \mathbb{M}_{n+4,a,b}^{\alpha,\beta} = \langle \{1, u, \dots, u^n, \cosh(au)\cos(bu), \cosh(au)\sin(bu), \end{aligned} \tag{3.6}$$

$$\begin{aligned} &\sinh(au)\cos(bu), \sinh(au)\sin(bu) : u \in [\alpha, \beta]\} \rangle, \ n \geq 0, \ a, b > 0, \\ &\dim \mathbb{M}_{n+4,a,b}^{\alpha,\beta} = n+5, \end{aligned} \tag{3.7}$$

respectively.

Remark 3.1 *It is known that:*

© Ágoston Róth, 2018

- *the normalized B-basis of* $\mathbb{P}_n^{\alpha,\beta}$ *is formed by the Bernstein polynomials [Carnicer and Peña, 1993] of degree* n *defined over any non-empty compact interval* $[\alpha,\beta]\subset\mathbb{R}$*;*
- *the unique normalized B-basis of* $\mathbb{T}_{2n}^{\alpha,\beta}$ *exists whenever* $\beta-\alpha\in(0,\pi)$ *and it was constructed in closed form in [Sánchez-Reyes, 1998];*
- *the explicit form of the unique normalized B-basis of the EC space* $\mathbb{H}_{2n}^{\alpha,\beta}$ *was derived in [Shen and Wang, 2005] and, theoretically, the interval length* $\beta-\alpha$ *can be any positive number (concerning numerical instabilities, the only limitation lies in the usage of potentially too big exponentials);*
- *for arbitrary values of the order* $n\geq 2$*, the critical lengths* $\ell'\left(\mathbb{AT}_{n(n+2)}^{\alpha,\beta}\right)$ *and* $\ell'\left(\mathbb{AET}_{n(2n+3)}^{\alpha,\beta}\right)$ *for design and the explicit forms of the unique normalized B-bases of the spaces* $\mathbb{AT}_{n(n+2)}^{\alpha,\beta}$ *and* $\mathbb{AET}_{n(2n+3)}^{\alpha,\beta}$*, respectively, were not studied in the literature; however, normalized B-bases of some special subspaces of these parent spaces were investigated, e.g., in [Carnicer et al., 2004, 2007, 2014; Mainar and Peña, 2004, 2010] and references therein;*
- *the mixed algebraic-hyperbolic-trigonometric EC space* $\mathbb{M}_{n+4,a,b}^{\alpha,\beta}$ *was investigated in [Brilleaud and Mazure, 2012], where, for* $n=0$*, it was shown that* $\ell'\left(\mathbb{M}_{4,a,b}^{\alpha,\beta}\right)$ *coincides with the only solution of the equation* $b\tanh(au)=a\tan(bu)$*, where* $u\in\left(\frac{\pi}{b},\frac{3\pi}{2b}\right)$*, i.e., in general, the critical length for design is not necessarily a free parameter with respect to the parameters resulting from the differential equation (1.9); for example, in case of parameters* $n=0$*,* $a=1$ *and* $b=0.2$ *one obtains that* $\ell'\left(\mathbb{M}_{4,1,0.2}^{\alpha,\beta}\right)\approx 16.694941067922716$*, i.e., the space* $\mathbb{M}_{4,1,0.2}^{\alpha,\beta}$ *possesses a unique normalized B-basis provided that* $\beta-\alpha\in(0,16.694941067922716)$*; moreover,* $\ell'\left(\mathbb{M}_{n+4,a,b}^{\alpha,\beta}\right)\geq\ell'\left(\mathbb{M}_{4,a,b}^{\alpha,\beta}\right)$*,* $\forall n\geq 1$*,* $\forall a,b>0$*.*

Following the provided examples, the user can also implement other special EC spaces that comprise the constant functions and can be identified with the solution spaces of constant-coefficient homogeneous linear differential equations of type (1.9/2). Note that in Listing 3.2/273, apart from some natural exception handling, the constructors of the aforementioned derived classes only specify the factorization of the characteristic polynomial (1.10/2) of the differential equation (1.9/2), and by doing this both well-known and new types of normalized B-bases can easily be constructed (on intervals having appropriate lengths), by writing only a few lines of code.

Listing 3.1. Specialized EC spaces (**SpecializedECSpaces.h**)

```
#pragma once

#include <EC/ECSpaces.h>

namespace cagd
{
    // Represents the EC space P_n^{α,β} = <{φ_{n,k}(u) = u^k : u ∈ [α,β]}_{k=0}^n> of polynomials of degree at most n,
    // where -∞ < α < β < ∞. It was shown in [Carnicer and Peña, 1993] that its normalized B-basis is the system
    // {b_{n,i}(u) = C(n,i) ((u-α)/(β-α))^i ((β-u)/(β-α))^{n-i} : u ∈ [α,β]}_{i=0}^n of Bernstein-polynomials of degree n.
    // Note that P_n^{α,β} can be identified with the solution space of the constant-coefficient homogeneous linear
    // differential equation determined by the characteristic polynomial p_{n+1}(z) = z^{n+1}, z ∈ C.
    class PolynomialECSpace: public ECSpace
    {
    protected:
        int _degree; // n

    public:
```


© Ágoston Róth, 2018

```
15          // special constructor
16          PolynomialECSpace(double alpha, double beta, int degree,
17                            bool check_for_ill_conditioned_matrices = false,
18                            int expected_correct_significant_digits = 5);
19          // clone function required by smart pointers based on the deep copy ownership policy
20          PolynomialECSpace* clone() const;
21      };
```

// Represents the EC space $\mathbb{T}_{2n}^{\alpha,\beta} = \langle \{\varphi_{2n,0} \equiv 1, \{\varphi_{2n,2k-1}(u) = \cos(ku), \varphi_{2n,2k}(u) = \sin(ku)\}_{k=1}^{n} : u \in [\alpha,\beta]\}\rangle$

// of trigonometric polynomials of order at most n (i.e., of degree at most $2n$), where $0 < \beta - \alpha < \ell'\left(\mathbb{T}_{2n}^{\alpha,\beta}\right) = \pi$,

// $\forall n \geq 1$. Based on [Sánchez-Reyes, 1998], it can be shown that its normalized B-basis is the system

// $\left\{b_{n,i}(u) = c_{2n,i}\sin^{2n-i}\left(\frac{\beta-u}{2}\right)\sin^{i}\left(\frac{u-\alpha}{2}\right) : u \in [\alpha,\beta]\right\}_{i=0}^{2n}$, where the normalizing coefficients $\{c_{2n,i}\}_{i=0}^{2n}$

// fulfill the symmetry $c_{2n,i} = c_{2n,2n-i} = \frac{1}{\sin^{2n}\left(\frac{\beta-\alpha}{2}\right)}\sum_{r=0}^{\left\lfloor\frac{i}{2}\right\rfloor}\binom{n}{i-r}\binom{i-r}{r}\left(2\cos\left(\frac{\beta-\alpha}{2}\right)\right)^{i-2r}$, $i = 0, 1, \ldots, n$.

// Observe that $\mathbb{T}_{2n}^{\alpha,\beta}$ coincides with the solution space of the constant-coefficient homogeneous linear differential

// equation determined by the characteristic polynomial $p_{2n+1}(z) = z\prod_{k=1}^{n}\left(z^2 + k^2\right)$, $z \in \mathbb{C}$.

```
29      class TrigonometricECSpace: public ECSpace
30      {
31      protected:
32          int _order; // n
33      public:
34          // special constructor
35          TrigonometricECSpace(double alpha, double beta, int order,
36                               bool check_for_ill_conditioned_matrices = false,
37                               int expected_correct_significant_digits = 5);
38          // clone function required by smart pointers based on the deep copy ownership policy
39          TrigonometricECSpace* clone() const;
40      };
```

// Represents the EC space $\mathbb{H}_{2n}^{\alpha,\beta} = \left\langle \left\{\varphi_{2n,0} \equiv 1, \left\{\varphi_{2n,2k-1}(u) = e^{ku}, \varphi_{2n,2k}(u) = e^{-ku}\right\}_{k=1}^{n} : u \in [\alpha,\beta]\right\}\right\rangle$ of

// hyperbolic polynomials of order at most n (i.e., of degree at most $2n$), where $0 < \beta - \alpha < \infty$.

// Based on [Shen and Wang, 2005], it can be shown that its normalized B-basis is the system

// $\left\{b_{n,i}(u) = c_{2n,i}\sinh^{2n-i}\left(\frac{\beta-u}{2}\right)\sinh^{i}\left(\frac{u-\alpha}{2}\right) : u \in [\alpha,\beta]\right\}_{i=0}^{2n}$, where the normalizing coefficients $\{c_{2n,i}\}_{i=0}^{2n}$

// fulfill the symmetry $c_{2n,i} = c_{2n,2n-i} = \frac{1}{\sinh^{2n}\left(\frac{\beta-\alpha}{2}\right)}\sum_{r=0}^{\left\lfloor\frac{i}{2}\right\rfloor}\binom{n}{i-r}\binom{i-r}{r}\left(2\cosh\left(\frac{\beta-\alpha}{2}\right)\right)^{i-2r}$, $i = 0, 1, \ldots, n$.

// (In [Shen and Wang, 2005], these unique normalizing coefficients were expressed in a different way.)

// Note that $\mathbb{H}_{2n}^{\alpha,\beta}$ is in fact the solution space of the constant-coefficient homogeneous linear differential

// equation determined by the characteristic polynomial $p_{2n+1}(z) = z\prod_{k=1}^{n}\left(z^2 - k^2\right)$, $z \in \mathbb{C}$.

```
49      class HyperbolicECSpace: public ECSpace
50      {
51      protected:
52          int _order; // n
53      public:
54          // special constructor
55          HyperbolicECSpace(double alpha, double beta, int order,
56                            bool check_for_ill_conditioned_matrices = false,
57                            int expected_correct_significant_digits = 5);
58          // clone function required by smart pointers based on the deep copy ownership policy
59          HyperbolicECSpace* clone() const;
60      };
```

// An algebraic-trigonometric EC space $\mathbb{AT}_{n(n+2)}^{\alpha,\beta} = \langle \{1, u, \ldots, u^n, \{u^r\cos(ku), u^r\sin(ku)\}_{r=0,\,k=1}^{n-k,\,n} : u \in [\alpha,\beta]\}\rangle$

// of order n and dimension $(n+1)^2$. Its critical length $\ell'\left(\mathbb{AT}_{n(n+2)}^{\alpha,\beta}\right)$ for design and the closed form of its

// normalized B-basis, in general, were not investigated in the literature. However, some special cycloidal

// subspaces of this space were studied e.g. in [Carnicer et al., 2004, 2014] and references therein.

// Note that $\mathbb{AT}_{n(n+2)}^{\alpha,\beta}$ can be identified with the solution space of the constant-coefficient homogeneous linear

// differential equation defined by the characteristic polynomial $p_{(n+1)^2}(z) = z^{n+1}\prod_{k=1}^{n}\left(z^2 + k^2\right)^{n+1-k}$, $z \in \mathbb{C}$.

```
67      class ATECSpace: public ECSpace
68      {
69      protected:
70          int _order; // n
```


© Ágoston Róth, 2018

```
71      public:
72          // special constructor
73          ATECSpace(double alpha, double beta, int order,
74                    bool check_for_ill_conditioned_matrices = false,
75                    int expected_correct_significant_digits = 5);
76          // clone function required by smart pointers based on the deep copy ownership policy
77          ATECSpace* clone() const;
78      };
79      // An algebraic-exponential-trigonometric EC space
80      // AET^{α,β}_{n(2n+3)} = ⟨ { {u^r}_{r=0}^n , { u^r e^{ku} cos(ku) , u^r e^{ku} sin(ku) }_{r=0, k=1}^{n-k, n} ,
81      //                         { u^r e^{-ku} cos(ku) , u^r e^{-ku} sin(ku) }_{r=0, k=1}^{n-k, n} : u ∈ [α, β] } ⟩
82      // of order n and dimension 2n^2 + 3n + 1. Its critical length ℓ'(AET^{α,β}_{n(2n+3)}) for design and the explicit form of
83      // its normalized B-basis, in general, was not investigated in the literature. Note that AET^{α,β}_{n(2n+3)} coincides with
84      // the solution space of the constant-coefficient homogeneous linear differential equation determined by the
85      // characteristic polynomial p_{2n^2+3n+1}(z) = z^{n+1} ∏_{k=1}^n [(z^2 - 2kz + 2k^2) · (z^2 + 2kz + 2k^2)]^{n+1-k}, z ∈ ℂ.
86      class AETECSpace: public ECSpace
87      {
88      protected:
89          int _order; // n
90      public:
91          // special constructor
92          AETECSpace(double alpha, double beta, int order,
93                     bool check_for_ill_conditioned_matrices = false,
94                     int expected_correct_significant_digits = 5);
95          // clone function required by smart pointers based on the deep copy ownership policy
96          AETECSpace* clone() const;
97      };
98      // The following two special EC spaces will be used for the B-representation of the ordinary exponential-
99      // trigonometric integral surface s(u,v) = [ s^0(u,v) ; s^1(u,v) ; s^2(u,v) ] = [ (1 - e^{ω0 u}) cos(u) (5/4 + cos(v)) ;
100     // where (u,v) ∈ [7π/2, 49π/8] × [-π/3, 5π/3], ω0 = 1/(6π) and ω1 = 1/(3π).
101     // The critical length for design and the explicit form of the normalized B-basis of the EC space
102     // ET_6^{α,β} = ⟨{1, cos(u), sin(u), e^{ω0 u}, e^{ω1 u}, e^{ω0 u} cos(u), e^{ω0 u} sin(u) : u ∈ [α, β]}⟩ were not
103     // studied in the literature. Observe that ET_6^{α,β} is in fact the solution space of the constant-coefficient
104     // homogeneous linear differential equation determined by the characteristic polynomial
105     // p_7(z) = z(z - i)(z + i)(z - ω0)(z - ω1)(z - (ω0 - i))(z - (ω0 + i)), z ∈ ℂ.
106     class SnailUECSpace: public ECSpace
107     {
108     public:
109         // special constructor
110         SnailUECSpace(double alpha, double beta,
111                       bool check_for_ill_conditioned_matrices = false,
112                       int expected_correct_significant_digits = 5);
113         // clone function required by smart pointers based on the deep copy ownership policy
114         SnailUECSpace* clone() const;
115     };
116     // This is only the first order special case of the pure trigonometric EC space represented by
117     // the class TrigonometricECSpace.
118     class SnailVECSpace: public TrigonometricECSpace
119     {
120     public:
121         // special constructor
122         SnailVECSpace(double alpha, double beta,
123                       bool check_for_ill_conditioned_matrices = false,
124                       int expected_correct_significant_digits = 5);
125         // clone function required by smart pointers based on the deep copy ownership policy
126         SnailVECSpace* clone() const;
```


© Ágoston Róth, 2018

```
127     };

128     // The following algebraic-hyperbolic-trigonometric EC space M^{α,β}_{n+4,a,b} = ⟨{1, u, ..., u^n, cosh(au)cos(bu), cosh(au)·
129     // ·sin(bu), sinh(au)cos(bu), sinh(au)sin(bu) : u ∈ [α, β]}⟩ was investigated in [Brilleaud and Mazure, 2012],
130     // where, for n = 0, it was shown that ℓ'(M^{α,β}_{4,a,b}) coincides with the only solution of the equation
131     // b tanh(au) = a tan(bu), where u ∈ (π/b, 3π/2b). For example, in case of parameters n = 0, a = 1 and b = 0.2 one
132     // obtains that ℓ'(M^{α,β}_{4,1,0.2}) ≈ 16.694941067922716, i.e., the space M^{α,β}_{4,1,0.2} possesses a unique normalized B-basis
133     // provided that β − α ∈ (0, 16.694941067922716). Moreover, ℓ'(M^{α,β}_{n+4,a,b}) ≥ ℓ'(M^{α,β}_{4,a,b}), ∀n ≥ 1, ∀a, b > 0.
134     // Note that M^{α,β}_{n+4,a,b} coincides with the solution space of a constant-coefficient homogeneous linear differential
135     // equation determined by the characteristic polynomial
136     // p_{n+5}(z) = z^{n+1}(z − (a − ib))(z − (a + ib))(z − (−a − ib))(z − (−a + ib)), z ∈ ℂ.
137     class BrilleaudMazureSpace: public ECSpace
138     {
139     protected:
140         int      _order; // n ∈ ℕ
141         double   _a, _b; // a, b > 0

142     public:
143         // special constructor
144         BrilleaudMazureSpace(double alpha, double beta, int order, double a, double b,
145                              bool check_for_ill_conditioned_matrices = false,
146                              int expected_correct_significant_digits = 5);

147         // clone function required by smart pointers based on the deep copy ownership policy
148         BrilleaudMazureSpace* clone() const;
149     };
150 }
```

Listing 3.2. Specialized EC spaces (**SpecializedECSpaces.cpp**)

```
1  #include "SpecializedECSpaces.h"

2  #include <Core/Exceptions.h>
3  #include <Core/Math/Constants.h>

4  namespace cagd
5  {
6      // Constructs the EC space P_n^{α,β} = ⟨{φ_{n,k}(u) = u^k : u ∈ [α, β]}_{k=0}^n⟩ of polynomials of degree at most n,
7      // where −∞ < α < β < ∞. It was shown in [Carnicer and Peña, 1993] that its normalized B-basis is the system
8      // {b_{n,i}(u) = C(n,i) ((u−α)/(β−α))^i ((β−u)/(β−α))^{n−i} : u ∈ [α, β]}_{i=0}^n of Bernstein-polynomials of degree n.
9      // Note that P_n^{α,β} can be identified with the solution space of the constant-coefficient homogeneous linear
10     // differential equation determined by the characteristic polynomial p_{n+1}(z) = z^{n+1}, z ∈ ℂ.
11     PolynomialECSpace::PolynomialECSpace(
12         double alpha, double beta, int degree,
13         bool check_for_ill_conditioned_matrices,
14         int expected_correct_significant_digits):
15         ECSpace(alpha, beta,
16                 check_for_ill_conditioned_matrices, expected_correct_significant_digits),
17         _degree(degree)
18     {
19         if (_degree < 0)
20         {
21             throw Exception("The degree of the polynomial EC space should be a "
22                             "non-negative integer!");
23         }

24         // we store the single zero z = 0 of order n + 1 of the characteristic polynomial p_{n+1}(z) = z^{n+1}, z ∈ ℂ
25         if (!insertZero(0.0, 0.0, _degree + 1, true,
26                         check_for_ill_conditioned_matrices,
27                         expected_correct_significant_digits))
28         {
29             throw Exception("The ordinary basis and the normalized B-basis of the "
30                             "vector space of polynomials of finite degree could "
31                             "not be updated!");
32         }
```


© Ágoston Róth, 2018

```
33      }
34      // clone function required by smart pointers based on the deep copy ownership policy
35      PolynomialECSpace* PolynomialECSpace::clone() const
36      {
37          return new (std::nothrow) PolynomialECSpace(*this);
38      }
39      // Builds the EC space T_{2n}^{α,β} = ⟨{φ_{2n,0} ≡ 1, {φ_{2n,2k-1}(u) = cos(ku), φ_{2n,2k}(u) = sin(ku)}_{k=1}^n : u ∈ [α,β]}⟩
40      // of trigonometric polynomials of order at most n (i.e., of degree at most 2n), where 0 < β − α < ℓ'(T_{2n}^{α,β}) = π,
41      // ∀n ≥ 1. Based on [Sánchez-Reyes, 1998], it can be shown that its normalized B-basis is the system
42      // {b_{n,i}(u) = c_{2n,i} sin^{2n-i}((β−u)/2) sin^i((u−α)/2) : u ∈ [α,β]}_{i=0}^{2n}, where the normalizing coefficients {c_{2n,i}}_{i=0}^{2n}
43      // fulfill the symmetry c_{2n,i} = c_{2n,2n-i} = 1/sin^{2n}((β−α)/2) Σ_{r=0}^{⌊i/2⌋} C(n,i-r) C(i-r,r) (2cos((β−α)/2))^{i-2r}, i = 0,1,...,n.
44      // Observe that T_{2n}^{α,β} coincides with the solution space of the constant-coefficient homogeneous linear differential
45      // equation determined by the characteristic polynomial p_{2n+1}(z) = z Π_{k=1}^n (z^2 + k^2), z ∈ ℂ.
46      TrigonometricECSpace::TrigonometricECSpace(
47          double alpha, double beta, int order,
48          bool check_for_ill_conditioned_matrices,
49          int expected_correct_significant_digits):
50          ECSpace(alpha, beta,
51                  check_for_ill_conditioned_matrices, expected_correct_significant_digits),
52          _order(order)
53      {
54          if (_order < 0)
55          {
56              throw Exception("The order of the vector space of pure trigonometric "
57                              "polynomials should be a non-negative integer!");
58          }
59          if (beta - alpha >= PI)
60          {
61              throw Exception("The vector space of trigonometric polynomials of finite "
62                              "order is EC provided that the length of its definition "
63                              "domain is strictly less than pi!");
64          }
65          // we store the first order zeros z = ±ik of the characteristic polynomial p_{2n+1}(z) = z Π_{k=1}^n (z^2 + k^2), z ∈ ℂ
66          // note that, in case of complex roots it is sufficient to store one of the conjugately equivalent zeros and the
67          // first order real zero z = 0 was already stored by the constructor of the base class ECSpace
68          for (int k = 1; k <= _order; k++)
69          {
70              insertZero(0.0, k, 1, false);
71          }
72          if (!updateBothBases(check_for_ill_conditioned_matrices,
73                               expected_correct_significant_digits))
74          {
75              throw Exception("The ordinary basis and the normalized B-basis of the "
76                              "vector space of trigonometric polynomials of finite "
77                              "order could not be updated!");
78          }
79      }
80      // clone function required by smart pointers based on the deep copy ownership policy
81      TrigonometricECSpace* TrigonometricECSpace::clone() const
82      {
83          return new (std::nothrow) TrigonometricECSpace(*this);
84      }
85      // Constructs the EC space H_{2n}^{α,β} = ⟨{φ_{2n,0} ≡ 1, {φ_{2n,2k-1}(u) = e^{ku}, φ_{2n,2k}(u) = e^{-ku}}_{k=1}^n : u ∈ [α,β]}⟩ of
86      // hyperbolic polynomials of order at most n (i.e., of degree at most 2n), where 0 < β − α < ∞.
87      // Based on [Shen and Wang, 2005], it can be shown that its normalized B-basis is the system
88      // {b_{n,i}(u) = c_{2n,i} sinh^{2n-i}((β−u)/2) sinh^i((u−α)/2) : u ∈ [α,β]}_{i=0}^{2n}, where the normalizing coefficients {c_{2n,i}}_{i=0}^{2n}
89      // fulfill the symmetry c_{2n,i} = c_{2n,2n-i} = 1/sinh^{2n}((β−α)/2) Σ_{r=0}^{⌊i/2⌋} C(n,i-r) C(i-r,r) (2cosh((β−α)/2))^{i-2r}, i = 0,1,...,n.
90      // (In [Shen and Wang, 2005], these unique normalizing coefficients were expressed in a different way.)
91      // Note that H_{2n}^{α,β} is in fact the solution space of the constant-coefficient homogeneous linear differential
```


© Ágoston Róth, 2018

```
92      // equation determined by the characteristic polynomial p_{2n+1}(z) = z∏_{k=1}^{n}(z^2 − k^2), z ∈ ℂ.
93      HyperbolicECSpace::HyperbolicECSpace(
94          double alpha, double beta, int order,
95          bool check_for_ill_conditioned_matrices,
96          int expected_correct_significant_digits):
97          ECSpace(alpha, beta,
98                  check_for_ill_conditioned_matrices, expected_correct_significant_digits),
99          _order(order)
100     {
101         if (_order < 0)
102         {
103             throw Exception("The order of the vector space of pure hyperbolic "
104                             "polynomials should be a non-negative integer!");
105         }
106
107         // we store the first order real zeros z = ±k of the characteristic polynomial p_{2n+1}(z) = z∏_{k=1}^{n}(z^2 − k^2)
108         // note that, first order real zero z = 0 was already stored by the constructor of the base class ECSpace
109         for (int k = 1; k <= _order; k++)
110         {
111             insertZero( k, 0.0, 1, false);
112             insertZero(-k, 0.0, 1, false);
113         }
114
115         if (!updateBothBases(check_for_ill_conditioned_matrices,
116                              expected_correct_significant_digits))
117         {
118             throw Exception("The ordinary basis and the normalized B-basis of the "
119                             "vector space of hyperbolic polynomials of finite order "
120                             "could not be updated!");
121         }
122     }
123
124     // clone function required by smart pointers based on the deep copy ownership policy
125     HyperbolicECSpace* HyperbolicECSpace::clone() const
126     {
127         return new (std::nothrow) HyperbolicECSpace(*this);
128     }
129
130     // Constructs the algebraic-trigonometric EC space
131     // 𝔸𝕋^{α,β}_{n(n+2)} = ⟨{1, u, ..., u^n, {u^r cos(ku), u^r sin(ku)}_{r=0, k=1}^{n−k, n} : u ∈ [α, β]}⟩
132     // of order n and dimension (n+1)^2. Its critical length ℓ'(𝔸𝕋^{α,β}_{n(n+2)}) for design and the closed form of its
133     // normalized B-basis, in general, were not investigated in the literature. However, some special cycloidal
134     // subspaces of this space were studied e.g. in [Carnicer et al., 2004, 2014] and references therein.
135     // Note that 𝔸𝕋^{α,β}_{n(n+2)} can be identified with the solution space of the constant-coefficient homogeneous linear
136     // differential equation defined by the characteristic polynomial p_{(n+1)^2}(z) = z^{n+1}∏_{k=1}^{n}(z^2 + k^2)^{n+1−k}, z ∈ ℂ.
137     ATECSpace::ATECSpace(
138         double alpha, double beta, int order,
139         bool check_for_ill_conditioned_matrices,
140         int expected_correct_significant_digits):
141         ECSpace(alpha, beta,
142                 check_for_ill_conditioned_matrices, expected_correct_significant_digits),
143         _order(order)
144     {
145         if (_order < 0)
146         {
147             throw Exception("The order of the mixed algebraic-trigonometric vector "
148                             "space should be a non-negative integer!");
149         }
150
151         // we store the (higher order) zeros of the characteristic polynomial p_{(n+1)^2}(z) = z^{n+1}∏_{k=1}^{n}(z^2 + k^2)^{n+1−k}
152         insertZero(0.0, 0.0, _order + 1, false);
153
154         // note that, in case of complex roots it is sufficient to store one of the conjugately equivalent zeros
155         for (int k = 1; k <= _order; k++)
156         {
157             insertZero(0.0, (double)k, _order + 1 - k, false);
158         }
159
160         if (!updateBothBases(check_for_ill_conditioned_matrices,
161                              expected_correct_significant_digits))
162         {
```


© Ágoston Róth, 2018

```
156             throw Exception("The ordinary basis and the normalized B-basis of the "
157                             "algebraic-trigonometric vector space could not be updated!");
158         }
159     }

160     // clone function required by smart pointers based on the deep copy ownership policy
161     ATECSpace* ATECSpace::clone() const
162     {
163         return new (std::nothrow) ATECSpace(*this);
164     }

165     // Constructs the algebraic-exponential-trigonometric EC space
166     // AET^{α,β}_{n(2n+3)} = ⟨ { {u^r}_{r=0}^n , { u^r e^{ku} cos(ku) , u^r e^{ku} sin(ku) }_{r=0,k=1}^{n-k,n} ,
167     //                         { u^r e^{-ku} cos(ku) , u^r e^{-ku} sin(ku) }_{r=0,k=1}^{n-k,n} : u ∈ [α,β] } ⟩
168     // of order n and dimension 2n^2 + 3n + 1. Its critical length ℓ'(AET^{α,β}_{n(2n+3)}) for design and the explicit form of
169     // its normalized B-basis, in general, was not investigated in the literature. Note that AET^{α,β}_{n(2n+3)} coincides with
170     // the solution space of the constant-coefficient homogeneous linear differential equation determined by the
171     // characteristic polynomial p_{2n^2+3n+1}(z) = z^{n+1} ∏_{k=1}^n [(z^2 - 2kz + 2k^2) · (z^2 + 2kz + 2k^2)]^{n+1-k}, z ∈ ℂ.
172     AETECSpace::AETECSpace(
173         double alpha, double beta, int order,
174         bool check_for_ill_conditioned_matrices,
175         int expected_correct_significant_digits):
176         ECSpace(alpha, beta,
177                 check_for_ill_conditioned_matrices, expected_correct_significant_digits),
178         _order(order)
179     {
180         if (_order < 0)
181         {
182             throw Exception("The order of the mixed algebraic-exponential-trigonometric "
183                             "vector space should be a non-negative integer!");
184         }

185         // we store the (higher order) zeros of the characteristic polynomial
186         // p_{2n^2+3n+1}(z) = z^{n+1} ∏_{k=1}^n [(z^2 - 2kz + 2k^2) · (z^2 + 2kz + 2k^2)]^{n+1-k}, z ∈ ℂ
187         insertZero(0.0, 0.0, _order + 1, false);

188         // note that, it is sufficient to store one of the conjugately equivalent zeros z = k ± ik and z = -k ± ik
189         // of order n + 1 - k
190         for (int k = 1; k <= _order; k++)
191         {
192             insertZero( k, k, _order + 1 - k, false);
193             insertZero(-k, k, _order + 1 - k, false);
194         }

195         if (!updateBothBases(check_for_ill_conditioned_matrices,
196                              expected_correct_significant_digits))
197         {
198             throw Exception("The ordinary basis and the normalized B-basis of the "
199                             "algebraic-exponential-trigonometric vector space could "
200                             "not be updated!");
201         }
202     }

203     // clone function required by smart pointers based on the deep copy ownership policy
204     AETECSpace* AETECSpace::clone() const
205     {
206         return new (std::nothrow) AETECSpace(*this);
207     }

208     // The following two special EC spaces will be used for the B-representation of the ordinary exponential-
209     // trigonometric integral surface s(u,v) = [ s^0(u,v) ; s^1(u,v) ; s^2(u,v) ] = [ (1 - e^{ω0 u}) cos(u) (5/4 + cos(v)) ;
209     //                                                                              (e^{ω0 u} - 1) sin(u) (5/4 + cos(v)) ;
209     //                                                                              7 - e^{ω1 u} - sin(v) + e^{ω0 u} sin(v) ],
210     // where (u,v) ∈ [7π/2, 49π/8] × [-π/3, 5π/3], ω0 = 1/(6π) and ω1 = 1/(3π).

211     // The critical length for design and the explicit form of the normalized B-basis of the EC space
212     // ET^{α,β}_6 = ⟨ {1, cos(u), sin(u), e^{ω0 u}, e^{ω1 u}, e^{ω0 u} cos(u), e^{ω0 u} sin(u) : u ∈ [α,β] } ⟩ were not
```


© Ágoston Róth, 2018

```
    // studied in the literature. Observe that ET_6^{α,β} is in fact the solution space of the constant-coefficient
    // homogeneous linear differential equation determined by the characteristic polynomial
    // p_7(z) = z(z − i)(z + i)(z − ω_0)(z − ω_1)(z − (ω_0 − i))(z − (ω_0 + i)), z ∈ C.

    // special constructor
    SnailUECSpace::SnailUECSpace(
        double alpha, double beta,
        bool check_for_ill_conditioned_matrices,
        int expected_correct_significant_digits):
        ECSpace(alpha, beta,
                check_for_ill_conditioned_matrices, expected_correct_significant_digits)
    {
        // we store one of the conjugately equivalent first order zeros of the characteristic polynomial
        // p_7(z) = z(z − i)(z + i)(z − ω_0)(z − ω_1)(z − (ω_0 − i))(z − (ω_0 + i)), z ∈ C
        insertZero(0.0,              1.0, 1, false);
        insertZero(1.0 / 6.0 / PI, 0.0, 1, false);
        insertZero(1.0 / 3.0 / PI, 0.0, 1, false);
        insertZero(1.0 / 6.0 / PI, 1.0, 1, false);

        if (!updateBothOrdinaryAndNNBBases(
                check_for_ill_conditioned_matrices, expected_correct_significant_digits))
        {
            throw Exception("The ordinary basis and the normalized B-basis of the "
                            "SnailUECSpace could not be updated!");
        }
    }

    // clone function required by smart pointers based on the deep copy ownership policy
    SnailUECSpace* SnailUECSpace::clone() const
    {
        return new (std::nothrow) SnailUECSpace(*this);
    }

    // Constructs the first order special case of the pure trigonometric EC space represented by
    // the class TrigonometricECSpace.
    SnailVECSpace::SnailVECSpace(
        double alpha, double beta,
        bool check_for_ill_conditioned_matrices,
        int expected_correct_significant_digits):
        TrigonometricECSpace(alpha, beta, 1,
                             check_for_ill_conditioned_matrices,
                             expected_correct_significant_digits)
    {
    }

    // clone function required by smart pointers based on the deep copy ownership policy
    SnailVECSpace* SnailVECSpace::clone() const
    {
        return new (std::nothrow) SnailVECSpace(*this);
    }

    // The following algebraic-hyperbolic-trigonometric EC space M^{α,β}_{n+4,a,b} = ⟨{1, u, ..., u^n, cosh(au)cos(bu), cosh(au)·
    // ·sin(bu), sinh(au)cos(bu), sinh(au)sin(bu) : u ∈ [α, β]}⟩ was investigated in [Brilleaud and Mazure, 2012],
    // where, for n = 0, it was shown that ℓ'(M^{α,β}_{4,a,b}) coincides with the only solution of the equation
    // b tanh(au) = a tan(bu), where u ∈ (π/b, 3π/2b). For example, in case of parameters n = 0, a = 1 and b = 0.2 one
    // obtains that ℓ'(M^{α,β}_{4,1,0.2}) ≈ 16.694941067922716, i.e., the space M^{α,β}_{4,1,0.2} possesses a unique normalized B-basis
    // provided that β − α ∈ (0, 16.694941067922716). Moreover, ℓ'(M^{α,β}_{n+4,a,b}) ≥ ℓ'(M^{α,β}_{4,a,b}), ∀n ≥ 1, ∀a, b > 0.
    // Note that M^{α,β}_{n+4,a,b} coincides with the solution space of a constant-coefficient homogeneous linear differential
    // equation determined by the characteristic polynomial
    // p_{n+5}(z) = z^{n+1}(z − (a − ib))(z − (a + ib))(z − (−a − ib))(z − (−a + ib)), z ∈ C.

    // special constructor
    BrilleaudMazureSpace::BrilleaudMazureSpace(
        double alpha, double beta, int order, double a, double b,
        bool check_for_ill_conditioned_matrices,
        int expected_correct_significant_digits):
        ECSpace(alpha, beta,
                check_for_ill_conditioned_matrices, expected_correct_significant_digits),
        _order(order),
        _a(a), _b(b)
    {
```


© Ágoston Róth, 2018

```
277         if (_order < 0)
278         {
279             throw Exception("The order of the mixed algebraic-hyperbolic-trigonometric "
280                             "vector space should be a non-negative integer!");
281         }
282         if (_a <= 0.0 || _b <= 0.0)
283         {
284             throw Exception("Parameters a and b should be positive reals!");
285         }
286         // we store the (first and higher order) zeros of the characteristic polynomial
287         // p_{n+5}(z) = z^{n+1}(z-(a-ib))(z-(a+ib))(z-(-a-ib))(z-(-a+ib)), a,b>0, z in C
288         insertZero(0.0, 0.0, _order + 1, false);
289         insertZero( a, b, 1, false);
290         insertZero(-a, b, 1, false);
291         if (!updateBothBases(check_for_ill_conditioned_matrices,
292                              expected_correct_significant_digits))
293         {
294             throw Exception("The ordinary basis and the normalized B-basis of the "
295                             "BrilleaudMazureSpace could not be updated!");
296         }
297     }
298     // clone function required by smart pointers based on the deep copy ownership policy
299     BrilleaudMazureSpace* BrilleaudMazureSpace::clone() const
300     {
301         return new (std::nothrow) BrilleaudMazureSpace(*this);
302     }
303 }
```

3.2 Using our shader programs

Classes GenericCurve3, TriangleMesh3, BCurve3: public LinearCombination3 and BSurface3: public TensorProductSurface3 provide several rendering methods. Each of them have two variants, the first of these overloaded member functions relies on instances of our class ShaderProgram, while the second one assumes that users handle their shader programs in a different way by providing at the same time attribute (like position, color, normal and texture coordinate) locations of proper types as input parameters for this second variant of our rendering methods. The latter variants will return a false value whenever one of the followings happens: the users did not activate their custom shader programs; the given attribute locations cannot be found in the list of active attribute locations, or they exist but are associated with attributes that have incorrect types (positions, colors, normals and texture coordinate attribute variables have to be of types vec3, vec4, vec3 and vec4, respectively); the rendering mode or other input parameters are invalid. For more details consider the lines 34/158–59/159, 49/172–71/172, 56/187–75/187 and 145/197–168/197 of Listings 2.37/157, 2.40/170, 2.42/185 and 2.44/194, respectively.

For convenience we have also provided shader programs for simple (flat) color shading, for two-sided per pixel lighting that is able to handle user-defined directional, point and spotlights with uniform front and back materials, and another one for reflection lines that are combined with two-sided per pixel lighting. All figures of this user manual were rendered by using these shader programs. If these would not be sufficient, more experienced users are invited to implement and use their custom shader programs and to append the provided function library with new special effects like in an open source project.

3.2.1 Simple (flat) color vertex and fragment shaders

Vertex buffer objects that store curve points, first and higher order derivatives, control polygons or nets do not provide coordinates for normal vectors. Therefore, in order to render these geometric

© Ágoston Róth, 2018

objects, we advise to use the simple (flat) color vertex and fragment shaders that are presented in Listings 3.3/279 and 3.4/279, respectively.

Listing 3.3. Simple color vertex shader

```
1 #version 130
2 uniform mat4 PVM;         // product of projection, view and model matrices
3 in      vec3 position;    // attribute
4 void main()
5 {
6     // convert the given position to clip coordinates and pass along
7     gl_Position = PVM * vec4(position, 1.0);
8 }
```

The uniform variable PVM of type mat4 declared in line 2/279 of Listing 3.3/279 denotes the product of the (either perspective or orthogonal) projection, of the view (i.e., camera) and of the model transformation (i.e., rotation, translation and scaling) matrices that were provided in Listing 2.30/119 as special OpenGL transformations. At the CPU-side, multiplications of these special transformation matrices can be performed by using the arithmetical multiplication operators inherited from their common generic base class GLTransformation that is defined and implemented in Listings 2.28/109 and 2.29/111, respectively. The constant address of the memory-continuous constant float array that stores the obtained 4×4 column-major ordered transformation matrix can be queried by the method GLTransformation::address() that is declared in line 91/111 of Listing 2.28/109. By passing this pointer as the fourth input parameter to the method GLboolean ShaderProgram::setUniformMatrix4fv(const std::string &name, GLsizei count, GLboolean transpose, const GLfloat *values) declared in line 316/134 of Listing 2.35/129, one can initialize the uniform matrix variable PVM.

Note that, our ShaderProgram objects assume by default that the name of the position attribute is "position", while its type is vec3. If required, this attribute name can be changed by using the method GLvoid ShaderProgram::setPositionAttributeName(const std::string &name) declared in line 256/133 of Listing 2.35/129.

Listing 3.4. Simple color fragment shader

```
1 #version 130
2 uniform vec4 color;          // uniform color variable that can be modified by the user
3 out     vec4 fragment_color; // output of the current fragment shader
4 void main()
5 {
6     fragment_color = color;
7 }
```

The uniform variable color of type vec4 declared in line 2/279 of Listing 3.4/279 can be initialized by using one of the methods GLboolean ShaderProgram::setUniformValue4f(const std::string &name, GLfloat value_0, GLfloat value_1, GLfloat value_2, GLfloat value_3) , GLboolean ShaderProgram::setUniformValue4fv(const std::string &name, GLsizei count, const GLfloat *values) or GLboolean ShaderProgram::setUniformColor(const std::string &name, const Color4 &color) that are declared in lines 283/134, 308/134 and 337/135, respectively, of Listing 2.35/129.

In our examples, we will frequently use some predefined colors that are specified in lines 218/51–242/51 of Listing 2.14/47. Moreover, in order to express some value dependent color variation, we will also use the color generation function Color4 coldToHotColormap(GLfloat value, GLfloat min_value, GLfloat max_value) that is declared and implemented in the lines 8/127 and 11/127–50/128 of Listings 2.33/127 and 2.34/127, respectively.

© Ágoston Róth, 2018

Examples for the usage of simple color shaders can be found in Listing pairs 3.8/284–3.9/285, 3.10/291–3.11/291, 3.12/294–3.13/296, 3.14/302–3.15/303, 3.16/307–3.17/308 and 3.18/321–3.19/324.

3.2.2 Two-sided per pixel lighting

In order to provide two-sided per pixel lighting, one can use the vertex and fragment shaders presented in Listings 3.5/280 and 3.6/280, respectively. With some modifications, these shaders are based on [Rost and Licea-Kane, 2006]. Examples for their usage can be found in Listing pairs 3.8/284–3.9/285, 3.16/307–3.17/308 and 3.18/321–3.19/324.

Listing 3.5. Vertex shader of the two-sided per pixel lighting and of reflection lines

```
#version 130

uniform mat4 VM;      // product of view and model matrices
uniform mat4 PVM;     // product of projection, view and model matrices
uniform mat4 N;       // normal matrix, i.e., transposed inverse of VM

in vec3 position;     // attributes associated with vertices
in vec3 normal;
in vec4 color;
in vec4 texture;

out vec3 interpolated_position; // outputs of the current vertex shader (i.e., inputs of the fragment shader
out vec3 interpolated_normal;   // listed in Listing 3.6/280)
out vec4 interpolated_color;
out vec4 interpolated_texture;

void main()
{
    // transform the normal vector to the eye space, then normalize it
    interpolated_normal   = normalize(vec3(N * vec4(normal, 0.0)));

    // transform the vertex position to the eye space
    interpolated_position = vec3(VM * vec4(position, 1.0));

    interpolated_color    = color;
    interpolated_texture  = texture;

    // convert the given position to clip coordinates and pass along
    gl_Position           = PVM * vec4(position, 1.0);
}
```

Listing 3.6. Fragment shader of the two-sided per pixel lighting

```
#version 130

in vec3            interpolated_position; // inputs of the current fragment shader (i.e., outputs of the
in vec3            interpolated_normal;   // vertex shader listed in Listing 3.5/280)
in vec4            interpolated_color;
in vec4            interpolated_texture;

out vec4           fragment_color;        // output of the current fragment shader

uniform float      transparency = 0.0;    // it can be used in case of color blending, its value should be in
                                          // the interval [0, 1]
uniform sampler2D  sampler_2D_texture;    // determines the currently active texture@*)

// Our ShaderProgram objects assume that uniform material variables are defined by means of the following structure:
struct Material
{
    vec4    ambient;    // ambient reflection coefficients
    vec4    diffuse;    // diffuse reflection coefficients
    vec4    specular;   // specular reflection coefficients
    vec4    emission;   // emissive color components
    float   shininess;  // shininess of the material, its value should be in the interval [0, 128]
```


© Ágoston Róth, 2018

```
};

uniform Material front_material; // uniform material associated with front faces
uniform Material back_material;  // uniform material associated with back faces

// Our ShaderProgram objects assume that uniform light source variables are defined by means of the next structure:
struct LightSource
{
    bool     enabled;
    vec4     position;
    vec4     half_vector;
    vec4     ambient;
    vec4     diffuse;
    vec4     specular;
    float    spot_cos_cutoff;
    float    constant_attenuation;
    float    linear_attenuation;
    float    quadratic_attenuation;
    vec3     spot_direction;
    float    spot_exponent;
};

const int number_of_light_sources = 1;  // increase this value if you want to use more light sources

uniform LightSource light_source[number_of_light_sources]; // an array of light sources

// distance is measured from the current vertex position to the ith light source.
float calculateAttenuation(in int i, in float distance)
{
    return(1.0 / (light_source[i].constant_attenuation +
                  light_source[i].linear_attenuation * distance +
                  light_source[i].quadratic_attenuation * distance * distance));
}

// N denotes the unit varying normal vector.
void directionalLight(in int i, in vec3 N, in float shininess,
                      inout vec4 ambient, inout vec4 diffuse, inout vec4 specular)
{
    vec3 L = normalize(light_source[i].position.xyz);

    float N_dot_L = dot(N, L);

    if (N_dot_L > 0.0)
    {
        vec3 H = light_source[i].half_vector.xyz;

        float pf = pow(max(dot(N, H), 0.0), shininess);

        diffuse  += light_source[i].diffuse  * N_dot_L;
        specular += light_source[i].specular * pf;
        ambient  += light_source[i].ambient;
    }
}

// N denotes the unit varying normal vector, while V corresponds to the varying vertex position.
void pointLight(in int i, in vec3 N, in vec3 V, in float shininess,
                inout vec4 ambient, inout vec4 diffuse, inout vec4 specular)
{
    vec3 D = light_source[i].position.xyz - V;
    vec3 L = normalize(D);

    float distance = length(D);
    float attenuation = calculateAttenuation(i, distance);

    float N_dot_L = dot(N, L);

    if (N_dot_L > 0.0)
    {
        vec3 E = normalize(-V);
        vec3 R = reflect(-L, N);

        float pf = pow(max(dot(R,E), 0.0), shininess);

        diffuse  += light_source[i].diffuse  * attenuation * N_dot_L;
```


© Ágoston Róth, 2018

```
76          specular += light_source[i].specular * attenuation * pf;
77          ambient  += light_source[i].ambient  * attenuation;
78      }
79  }

80  // N denotes the unit varying normal vector, while V corresponds to the varying vertex position.
81  void spotlight(in int i, in vec3 N, in vec3 V, in float shininess,
82                 inout vec4 ambient, inout vec4 diffuse, inout vec4 specular)
83  {
84      vec3 D = light_source[i].position.xyz - V;
85      vec3 L = normalize(D);

86      float distance = length(D);
87      float attenuation = calculateAttenuation(i, distance);

88      float N_dot_L = dot(N,L);

89      if (N_dot_L > 0.0)
90      {
91          float spot_effect = dot(normalize(light_source[i].spot_direction), -L);

92          if (spot_effect > light_source[i].spot_cos_cutoff)
93          {
94              attenuation *=  pow(spot_effect, light_source[i].spot_exponent);

95              vec3 E = normalize(-V);
96              vec3 R = reflect(-L, N);

97              float pf = pow(max(dot(R, E), 0.0), shininess);

98              diffuse  += light_source[i].diffuse  * attenuation * N_dot_L;
99              specular += light_source[i].specular * attenuation * pf;
100         }
101         ambient  += light_source[i].ambient * attenuation;
102     }
103 }

104 // N denotes the unit varying normal vector, while V corresponds to the varying vertex position.
105 void calculateLighting(in int number_of_light_sources, in vec3 N, in vec3 V,
106                        in float shininess,
107                        inout vec4 ambient, inout vec4 diffuse, inout vec4 specular)
108 {
109     // Just loop through each light, and if its enabled add its contributions to the color of the pixel.
110     for (int i = 0; i < number_of_light_sources; i++)
111     {
112         if (light_source[i].enabled)
113         {
114             if (light_source[i].position.w == 0.0)
115             {
116                 directionalLight(i, N, shininess, ambient, diffuse, specular);
117             }
118             else
119             {
120                 if (light_source[i].spot_cos_cutoff == 180.0)
121                 {
122                     pointLight(i, N, V, shininess, ambient, diffuse, specular);
123                 }
124                 else
125                 {
126                     spotlight(i, N, V, shininess, ambient, diffuse, specular);
127                 }
128             }
129         }
130     }
131 }

132 void main()
133 {
134     // Normalize the interpolated normal. Since a varying variable cannot be modified by a fragment shader,
135     // a new variable needs to be created.
136     vec3 n = normalize(interpolated_normal);

137     vec4 ambient  = vec4(0.0), diffuse = vec4(0.0),
138          specular = vec4(0.0), color   = vec4(0.0);
```


© Ágoston Róth, 2018

```
139     // Initialize the contributions for the front-face-pass over the lights.
140     ambient  = texture2DProj(sampler_2D_texture, interpolated_texture);
141     diffuse  = texture2DProj(sampler_2D_texture, interpolated_texture);
142     calculateLighting(number_of_light_sources, n, interpolated_position,
143                       front_material.shininess, ambient, diffuse, specular);
144     color += front_material.emission +
145              (ambient  * (front_material.ambient + interpolated_color)) +
146              (diffuse  * (front_material.diffuse + interpolated_color)) +
147              (specular * (front_material.specular));
148     // Re-initialize the contributions for the back-face-pass over the lights.
149     ambient  = texture2DProj(sampler_2D_texture, interpolated_texture);
150     diffuse  = texture2DProj(sampler_2D_texture, interpolated_texture);
151     specular = vec4(0.0);
152     // Now caculate the back contribution. All that needs to be done is to flip the normal.
153     calculateLighting(number_of_light_sources, -n, interpolated_position,
154                       back_material.shininess, ambient, diffuse, specular);
155     color += back_material.emission +
156              (ambient  * (back_material.ambient + interpolated_color)) +
157              (diffuse  * (back_material.diffuse + interpolated_color)) +
158              (specular * (back_material.specular));
159     color = clamp(color, 0.0, 1.0);
160     fragment_color = vec4(color.rgb, clamp(1.0 - transparency, 0.0, 1.0));
161 }
```

3.2.3 Reflection lines combined with two-sided per pixel lighting

The vertex shader of the reflection lines coincides with that of the two-sided per pixel lighting presented in Listing 3.5/280. By modifying a free fragment shader created by Gaël Guennebaud for the powerful mesh processing tool MeshLab, the fragment shader of the reflection lines is implemented in Listing 3.7/283 that – compared with the fragment shader of the two-sided per pixel lighting listed in Listing 3.6/280 – introduces some new constants and uniform variables, and at the same time also appends the code with reflection line (or zebra stripe) generation. (The newly appended code snippets are included between consecutive pairs of red and blue horizontal lines.)

Listing 3.7. Fragment shader of reflection lines

```
1  // ...
2  // these lines coincide with lines 1/280–9/280 of Listing 3.6/280
3  // ...
4  uniform float       scale_factor;           // affects the number of reflection lines, its value should be
                                                // greater than zero
5  uniform float       smoothing;              // smooths the common boundaries of adjacent stripes,
                                                // its value should be greater than zero
6  uniform float       shading;                // shades the model, its value should be in the range [0, 1]
7  const   float       TWO_PI = 6.28318530717958647692528676655 9;
8  // ...
9  // these lines coincide with lines 10/280–131/282 of Listing 3.6/280
10 // ...
11 void main()
12 {
13     // ...
14     // these lines coincide with lines 134/282–159/283 of Listing 3.6/280
15     // ...
16     // generating reflection lines...
17     vec3 v  = normalize(interpolated_position);
18     vec3 r  = v - 2.0 * n * dot(n, v); // reflection vector
```


© Ágoston Róth, 2018

```
19      r.z    += 1.0;
20      r.z     = 0.5 / sqrt(dot(r, r));
21      r.xy    = (r.xy * r.z) + 0.5;
22      r      *= 2.0;

23      float sharpness = 1.0 / asin(smoothing * fwidth(r.x) * 2.0 * scale_factor);

24      color   *= vec4(clamp(0.5 + sharpness * sin(TWO_PI * r.x * scale_factor), 0.0, 1.0));
25      color.a  = 1.0;

26      fragment_color = clamp(dot(light_source[0].position.xyz, n) * shading +
27                             (1.0 - shading), 0.0, 1.0) *
28                       min(vec4(color.rgb, clamp(1.0 - transparency, 0.0, 1.0)), vec4(1.0));
29  }
```

Examples for the usage of the reflection lines generating shader can be found in Listing pairs 3.8/284–3.9/285, 3.16/307–3.17/308 and 3.18/321–3.19/324.

Listing 3.8. Using the provided shader programs

```
1  #ifndef YOURGLWIDGET_H
2  #define YOURGLWIDGET_H

3  #include <GL/glew.h> // in order to handle vertex buffer and shader program objects, we rely on GLEW
4                       // we assume that this external dependency is added to your project

5  #include <Core/Geometry/Coordinates/Cartesians3.h>
6  #include <Core/Geometry/Surfaces/Lights.h>
7  #include <Core/Math/SpecialGLTransformations.h>
8  #include <Core/Shaders/ShaderPrograms.h>
9  #include <Core/SmartPointers/SpecializedSmartPointers.h>

10 namespace cagd // all data structures in the proposed function library are defined under the namespace cagd
11 {
12     class YourGLWidget: public ...
13     {
14     private:
15         // variables defining the orthogonal projection matrix
16         GLfloat _aspect;           // aspect ratio of the rendering window
17         GLfloat _left,   _right;   // minimum and maximum values of x-coordinates
18         GLfloat _bottom, _top;     // minimum and maximum values of y-coordinates
19         GLfloat _near,   _far;     // minimum and maximum values of z-coordinates
20         SP<OrthogonalProjection>::Default _P; // smart pointer to an orthogonal projection matrix

21         // variables defining the view matrix
22         Cartesian3           _eye, _center, _up;
23         SP<LookAt>::Default _V; // smart pointer to the view or world matrix

24         // transformation matrices
25         Rotate              _Rx, _Ry, _Rz; // rotation matrices around axis x, y and z, respectively
26         Translate           _T;            // translation
27         Scale               _S;            // scaling
28         GLTransformation    _M,            // model matrix, i.e., the product _Rx * _Ry * _Rz * _T * _S
29                             _VM,           // product of view and model matrices
30                             _PVM,          // product of projection, view and model matrices
31                             _tN;           // transposed normal matrix, i.e., inverse of _VM

32         // a private method that calculates the transformationmatrices _M, _VM, _PVM and _tN
33         GLboolean           _updateTransformationMatrices();

34         // shader program objects
35         ShaderProgram       _color_shader;
36         ShaderProgram       _two_sided_lighting;
37         ShaderProgram       _reflection_lines;

38         // a smart pointer to a directional light object
39         SP<DirectionalLight>::Default _light;

40         // your other declarations, e.g. one may declare smart pointers to objects of type ECSpace, BCurve3,
41         // BSurface3, GenericCurve3 and TriangleMesh3...
```


© Ágoston Róth, 2018

```
42      public:
43          // your default/special constructor
44          YourGLWidget(/* ... */);
45
45          // your rendering method
46          void render();
47
47          // your other methods...
48      };
49  }
50  #endif // YOURGLWIDGET_H
```

Listing 3.9. Using the provided shader programs

```
1   #include "YourGLWidget.h"
2   #include <iostream>
3   #include <fstream>
4   #include <Core/Exceptions.h>
5   #include <Core/Utilities.h>
6   #include <Core/Geometry/Coordinates/Colors4.h>
7   #include <Core/Geometry/Surfaces/Materials.h>
8   using namespace cagd;
9   using namespace std;
10  // your default/special constructor
11  YourGLWidget::YourGLWidget(/* ... */)
12  {
13      try
14      {
15          // try to initialize the OpenGL Extension Wrangler library
16          if (glewInit() != GLEW_OK)
17          {
18              throw Exception("Could not initialize the "
19                              "OpenGL Extension Wrangler Library!");
20          }
21          // test whether your platform is supported from the perspective of the proposed function library
22          if (!platformIsSupported())
23          {
24              throw Exception("The platform is not supported!");
25          }
26          // creating an orthogonal perspective projection matrix
27          _aspect = (float)width() / (float)height();
28          _left   = _bottom = -10.0f;
29          _right  = _top    = +10.0f;
30          _near   = -20.0f;
31          _far    = +20.0f;
32          _P = SP<OrthogonalProjection>::Default(
33                      new (nothrow) OrthogonalProjection(
34                              _aspect, _left, _right, _bottom, _top, _near, _far));
35          if (!_P)
36          {
37              throw Exception("Could not create the orthogonal projection matrix!");
38          }
39          // creating a view (or world) transformation matrix
40          _eye[0]    = _eye[1]    = 0.0, _eye[2]    = 6.0;
41          _center[0] = _center[1] =      _center[2] = 0.0;
42          _up[0]     = _up[2]     = 0.0, _up[1]     = 1.0;
43          _V = SP<LookAt>::Default(new (nothrow) LookAt(_eye, _center, _up));
44          if (!_V)
45          {
46              throw Exception("Could not create the view/world transformation matrix!");
```


© Ágoston Róth, 2018

```
47          }

48          // specifying the axes of rotation matrices
49          _Rx.setDirection(Cartesian3(1.0, 0.0, 0.0));
50          _Ry.setDirection(Cartesian3(0.0, 1.0, 0.0));
51          _Rz.setDirection(Cartesian3(0.0, 0.0, 1.0));

52          // By default, all rotation angles, translation units and scaling factors are set to 0, 0 and 1, respectively.

53          // In what follows, we assume that the vertex and fragment shader files color.vert/frag,
54          // two_sided_lighting.vert/frag and reflection_lines.vert/frag are downloaded and placed
55          // in a folder named Shaders that is created along side of your executable.

56          // If the variable logging_is_enabled is set to true, one obtains logging information at run-time about
57          // creating, loading, compiling, attaching and linking different types of shaders.
58          // If one is convinced that the applied shaders do not contain bugs, this logging mechanism can be deactivated.
59          GLboolean logging_is_enabled = GL_TRUE;

60          // By default, logging information appear in the standard console output, but they can be redirected
61          // to any output streams as it is shown in the following lines.
62          fstream    shader_log("shader.log", ios_base::out);

63          if (!_color_shader.attachNewShaderFromSourceFile(
64                      ShaderProgram::Shader::VERTEX, "Shaders/color.vert",
65                      logging_is_enabled, shader_log))
66          {
67              throw Exception("Could not attach the vertex shader of the "
68                              "color shader program!");
69          }

70          if (!_color_shader.attachNewShaderFromSourceFile(
71                      ShaderProgram::Shader::FRAGMENT, "Shaders/color.frag",
72                      logging_is_enabled, shader_log))
73          {
74              throw Exception("Could not attach the fragment shader of the "
75                              "color shader program!");
76          }

77          if (!_color_shader.linkAttachedShaders(logging_is_enabled, shader_log))
78          {
79              throw Exception("Could not link the attached shaders of the "
80                              "color shader program!");
81          }

82          if (!_two_sided_lighting.attachNewShaderFromSourceFile(
83                      ShaderProgram::Shader::VERTEX, "Shaders/two_sided_lighting.vert",
84                      logging_is_enabled, shader_log))
85          {
86              throw Exception("Could not attach the vertex shader of two "
87                              "sided lighting shader program!");
88          }

89          if (!_two_sided_lighting.attachNewShaderFromSourceFile(
90                      ShaderProgram::Shader::FRAGMENT, "Shaders/two_sided_lighting.frag",
91                      logging_is_enabled, shader_log))
92          {
93              throw Exception("Could not attach the fragment shader of "
94                              "two sided lighting shader program!");
95          }

96          if (!_two_sided_lighting.linkAttachedShaders(logging_is_enabled, shader_log))
97          {
98              throw Exception("Could not link the attached shaders of the "
99                              "two sided lighting shader program!");
100         }

101         if (!_reflection_lines.attachNewShaderFromSourceFile(
102                     ShaderProgram::Shader::VERTEX, "Shaders/reflection_lines.vert",
103                     logging_is_enabled, shader_log))
104         {
105             throw Exception("Could not attach the vertex shader of the "
106                             "reflection lines shader program!");
107         }
```


© Ágoston Róth, 2018

```
108         if (!_reflection_lines.attachNewShaderFromSourceFile(
109                     ShaderProgram::Shader::FRAGMENT, "Shaders/reflection_lines.frag",
110                     logging_is_enabled, shader_log))
111         {
112             throw Exception("Could not attach the fragment shader of the "
113                             "reflection lines shader program!");
114         }
115         if (!_reflection_lines.linkAttachedShaders(logging_is_enabled, shader_log))
116         {
117             throw Exception("Could not link the attached shaders of the "
118                             "reflection lines shader program!");
119         }
120         shader_log.close();
121         // Try to update all required transformation matrices (the method _updateTransformationMatrices() has to
122         // be called after every successful transformation related event handling).
123         if (!_updateTransformationMatrices())
124         {
125             throw Exception("Could not update all transformation matrices!");
126         }
127         // creating a directional light object
128         Cartesian3 direction(0.0, 0.0, 1.0);
129         Cartesian3 eye = _eye;
130         eye.normalize();
131         Cartesian3 half_vector = direction;
132         half_vector += eye;
133         half_vector.normalize();
134         _light = SP<DirectionalLight>::Default(
135                     new DirectionalLight(
136                         Homogeneous3(0.0f, 0.0f, 1.0f, 0.0f), // direction vector of the light
137                         Homogeneous3(half_vector),            // homogeneous half vector
138                         Color4(0.4f, 0.4f, 0.4f, 1.0f),       // ambient light intensity
139                         Color4(0.8f, 0.8f, 0.8f, 1.0f),       // diffuse light intensity
140                         Color4(1.0f, 1.0f, 1.0f, 1.0f)));     // specular light intensity
141         if (!_light)
142         {
143             throw Exception("Could not create the directional light object!");
144         }
145         // examples for communicating with our shader programs via uniform variables
146         _two_sided_lighting.enable();
147         if (!_two_sided_lighting.setUniformDirectionalLight("light_source[0]", *_light))
148         {
149             throw Exception("Two-sided per pixed lighting: could not initialize the "
150                             "uniform variable \"light_source[0]\"!");
151         }
152         if (!_two_sided_lighting.setUniformValue1i("light_source[0].enabled", GL_TRUE))
153         {
154             throw Exception("Two-sided per pixed lighting: could not initialize the "
155                             "uniform variable \"light_source[0].enabled\"!");
156         }
157         if (!_two_sided_lighting.setUniformMaterial("front_material", materials::brass))
158         {
159             throw Exception("Two-sided per pixed lighting: could not initialize the "
160                             "uniform variable \"front_material\"!");
161         }
162         if (!_two_sided_lighting.setUniformMaterial("back_material", materials::chrome))
163         {
164             throw Exception("Two-sided per pixed lighting: could not initialize the "
165                             "uniform variable \"back_material\"!");
166         }
```


© Ágoston Róth, 2018

```
167         _two_sided_lighting.disable();
168         _reflection_lines.enable();
169         if (!_reflection_lines.setUniformDirectionalLight("light_source[0]", *_light))
170         {
171             throw Exception("Reflection lines: could not initialize the "
172                             "uniform variable \"light_source[0]\"!");
173         }
174         if (!_reflection_lines.setUniformValue1i("light_source[0].enabled", GL_TRUE))
175         {
176             throw Exception("Reflection lines: could not initialize the "
177                             "uniform variable \"light_source[0].enabled\"!");
178         }
179         if (!_reflection_lines.setUniformColorMaterial("front_material", colors::green))
180         {
181             throw Exception("Reflection lines: could not initialize the "
182                             "uniform variable \"front_material\"!");
183         }
184         if (!_reflection_lines.setUniformColorMaterial("back_material", colors::orange))
185         {
186             throw Exception("Reflection lines: could not initialize the "
187                             "uniform variable \"back_material\"!");
188         }
189         if (!_reflection_lines.setUniformValue1f("scale_factor", 9.7f))
190         {
191             throw Exception("Reflection lines: could not initialize the "
192                             "uniform variable \"scale_factor\"!");
193         }
194         if (!_reflection_lines.setUniformValue1f("smoothing", 1.0f))
195         {
196             throw Exception("Reflection lines: could not initialize the "
197                             "uniform variable \"smoothing\"!");
198         }
199         if (!_reflection_lines.setUniformValue1f("shading", 0.1f))
200         {
201             throw Exception("Reflection lines: could not initialize the "
202                             "uniform variable \"shading\"!");
203         }
204         _reflection_lines.disable();
205         // Remark
206         // Lines 260/133–392/135 of Listing 2.35/129 declare many other methods that can be used to initialize
207         // possible uniform variables. Here we used only a few of them.
208         // Remark
209         // If one does not want to use our class ShaderProgram, then lines 59/286–120/287 and 146/287–204/288 of
210         // the current listing should be replaced by using other data types and programming techniques.
211         glEnable(GL_DEPTH_TEST);                // enable depth testing
212         glClearColor(1.0f, 1.0f, 1.0f, 1.0f); // set the background color
213         // try to create your renderable geometry, by instantiating, e.g., our classes ECSpace, BCurve3, BSurface3,
214         // GenericCurve3 and TriangleMesh3...
215     }
216     catch (Exception &e)
217     {
218         cout << e << endl;
219     }
220 }
221 // a private method that calculates the transformationmatrices _M, _VM, _PVM and _tN
222 GLboolean YourGLWidget::_updateTransformationMatrices()
223 {
224     if (_P && _V)
225     {
```


© Ágoston Róth, 2018

```
226             _M                = _Rx * _Ry * _Rz * _T * _S;
227             _VM               = (*_V) * _M;
228             _PVM              = (*_P) * _VM;

229             bool invertible = false;
230             _tN               = _VM.inverse(&invertible);

231             return invertible;
232         }

233         return GL_FALSE;
234     }

235     // your rendering method
236     void YourGLWidget::render()
237     {
238         // clears the color and depth buffers
239         glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

240         // For the sake of simplicity, in this method we have omitted the try-catch mechanism that would detect
241         // possible errors that originate from initializations of non-existent uniform variables.
242         // In what follows we outline three possibilities to use the provided shader programs.

243         // 1st possibility
244         if (/* your curve points, derivatives, control polygons/nets that have to be rendered exist and are renderable*/)
245         {
246             _color_shader.enable();

247             _color_shader.setUniformColor("color", colors::black);
248             _color_shader.setUniformMatrix4fv("PVM", 1, GL_FALSE, _PVM.address());

249             // Render your line geometry with the color shader program, e.g., you may pass the constant reference
250             // of the variable _color_shader as the first input variable to the rendering methods:
251             //
252             // GLboolean GenericCurve3::renderDerivatives(
253             //     const ShaderProgram &program, GLint order, GLenum render_mode) const;
254             //
255             // GLboolean BCurve3::renderData(
256             //     const ShaderProgram &program, GLenum render_mode = GL_LINE_STRIP) const;
257             //
258             // GLboolean BSurface3::renderData(
259             //     const ShaderProgram &program,
260             //     GLenum u_render_mode = GL_LINE_STRIP, GLenum v_render_mode = GL_LINE_STRIP) const.

261             // Remark
262             // If you do not use our class ShaderProgram, then instead of the previously listed three rendering methods
263             // you should use the member functions:
264             //
265             // GLboolean GenericCurve3::renderDerivatives(
266             //     GLint order, GLenum render_mode, GLint vec3_position_location = 0) const;
267             //
268             // GLboolean BCurve3::renderData(
269             //     GLenum render_mode = GL_LINE_STRIP, GLint vec3_position_location = 0) const;
270             //
271             // GLboolean BSurface3::renderData(
272             //     GLenum u_render_mode = GL_LINE_STRIP, GLenum v_render_mode = GL_LINE_STRIP,
273             //     GLint vec3_position_location = 0) const,
274             //
275             // but in these cases you should provide a valid and active attribute location that is associated with positions
276             // of type vec3 in your custom shader programs. (At the same time it is your job to handle possible
277             // uniform variables.)

278             _color_shader.disable();
279         }

280         // 2nd possibility
281         if (/* your triangles meshes that have to be rendered exist and are renderable*/)
282         {
283             _two_sided_lighting.enable();

284             _two_sided_lighting.setUniformMatrix4fv("VM",   1, GL_FALSE, _VM.address());
285             _two_sided_lighting.setUniformMatrix4fv("PVM",  1, GL_FALSE, _PVM.address());
286             _two_sided_lighting.setUniformMatrix4fv("N",    1, GL_TRUE,  _tN.address());
```


© Ágoston Róth, 2018

```
287             // Render your triangles meshes with the two sided lighting shader program, e.g., you may pass the constant
288             // reference of the variable _two_sided_lighting as the first input variable to the rendering method:
289             //
290             // GLboolean TriangleMesh3::render(
291             //      const ShaderProgram &program, GLenum render_mode = GL_TRIANGLES) const.

292             // Remark
293             // If you do not use our class ShaderProgram, then instead of the previous rendering method you should
294             // use the member function:
295             //
296             // GLboolean TriangleMesh3::render(
297             //      GLenum render_mode = GL_TRIANGLES,
298             //      GLint vec3_position_location = 0, GLint vec3_normal_location = 1,
299             //      GLint vec4_color_location = 2, GLint vec4_texture_location = 3) const,
300             //
301             // but in this case you should provide valid and active attribute locations that are associated in your
302             // custom shader programs with positions, normals, colors and texture coordinates of type vec3, vec3,
303             // vec4 and vec4, respectively. (At the same time it is your job to handle possible uniform variables.)

304             _two_sided_lighting.disable();
305         }

306         // 3rd possibility
307         if (/* your triangles meshes that have to be rendered exist and are renderable*/)
308         {
309             _reflection_lines.enable();

310             _reflection_lines.setUniformMatrix4fv("VM",   1, GL_FALSE, _VM.address());
311             _reflection_lines.setUniformMatrix4fv("PVM",  1, GL_FALSE, _PVM.address());
312             _reflection_lines.setUniformMatrix4fv("N",    1, GL_TRUE,  _tN.address());

313             // Render your triangles meshes with the reflection lines shader program, e.g., you may pass the constant
314             // reference of the variable _reflection_lines as the first input variable to the rendering method:
315             //
316             // GLboolean TriangleMesh3::render(
317             //      const ShaderProgram &program, GLenum render_mode = GL_TRIANGLES) const.

318             // Remark
319             // If you do not use our class ShaderProgram, then instead of the previous rendering method you should
320             // use the member function:
321             //
322             // GLboolean TriangleMesh3::render(
323             //      GLenum render_mode = GL_TRIANGLES,
324             //      GLint vec3_position_location = 0, GLint vec3_normal_location = 1,
325             //      GLint vec4_color_location = 2, GLint vec4_texture_location = 3) const,
326             //
327             // but in this case you should provide valid and active attribute locations that are associated in your
328             // custom shader programs with positions, normals, colors and texture coordinates of type vec3, vec3,
329             // vec4 and vec4, respectively. (At the same time it is your job to handle possible uniform variables.)

330             _reflection_lines.disable();
331         }

332         // Remark
333         // Note that, the constant memory addresses of all transformation matrices have to be passed in this rendering
334         // method of your application to the corresponding uniform variables. The reason of this is the fact that the
335         // underlying matrices may change in your transformation related event handling methods. (Do not forget to
336         // call the method _updateTransformationMatrices() before leaving your event handling slots.)
337     }
```

3.3 Differentiating and rendering basis functions

Once an EC space object is created, one can differentiate and render both its ordinary basis and normalized B-basis functions. Listings 3.10/291 and 3.11/291 declare and define data structures for the evaluation and rendering of the unique normalized B-bases of the EC spaces $\mathbb{P}_{10}^{0,1}$, $\mathbb{T}_{10}^{0,\frac{\pi}{2}}$, $\mathbb{H}_{14}^{0,\pi}$, $\mathbb{AT}_{24}^{0,2\pi}$, $\mathbb{AET}_{27}^{-5\pi/4,\,5\pi/4}$ and $\mathbb{M}_{n+4,1,0.2}^{0,\beta}$ that are of types (3.1/269), (3.2/269), (3.3/269), (3.4/269), (3.5/269) and (3.6/269), respectively, where $\beta = 16.694941067922716$.

© Ágoston Róth, 2018

Listing 3.10. Differentiating and rendering basis functions (**YourGLWidget.h**)

```
#ifndef YOURGLWIDGET_H
#define YOURGLWIDGET_H

#include <GL/glew.h> // in order to handle vertex buffer and shader program objects, we rely on GLEW
                     // we assume that this external dependency is added to your project

#include <Core/Geometry/Curves/GenericCurves3.h>
#include <Core/Math/SpecialGLTransformations.h>
#include <Core/Shaders/ShaderPrograms.h>
#include <Core/SmartPointers/SpecializedSmartPointers.h>
#include <EC/ECSpaces.h>

namespace cagd // all data structures in the proposed function library are defined under the namespace cagd
{
    class YourGLWidget: public ...
    {
    private:
        // ...
        // these lines coincide with the lines 15/284–35/284 of Listing 3.8/284
        // ...

        // Parameters of different EC spaces:
        GLint                                   _space_count;   // number of considered EC spaces;
        std::vector<GLdouble>                   _alpha, _beta; // endpoints of definition domains;
        std::vector<GLint>                      _n;             // orders of considered EC spaces;
        std::vector<SP<ECSpace>::Default> _space;         // smart pointers to different EC spaces.

        // Parameters used for the image generation of normalized B-basis functions:
        // the common number of subdivision points in each definition domain;
        GLint _div_point_count;
        // the maximum order of derivatives that have to be evaluated along the basis functions;
        GLint _maximum_order_of_derivatives;
        // an array of smart pointers pointing to row matrices that store the image smart pointers of all
        // normalized B-basis functions of each EC space.
        std::vector< SP< RowMatrix<SP<GenericCurve3>::Default> >::Default > _img_B_basis;

        // your other declarations...

    public:
        // your default/special constructor
        YourGLWidget(/*...*/);

        // your rendering method
        void render();

        // your other methods...
    };
}

#endif // YOURGLWIDGET_H
```

Listing 3.11. Differentiating and rendering basis functions (**YourGLWidget.cpp**)

```
#pragma warning(disable:4503)
#include "YourGLWidget.h"

#include <iostream>
#include <fstream>

#include <Core/Exceptions.h>
#include <Core/Utilities.h>
#include <Core/Math/Constants.h>

#include "../Spaces/SpecializedECSpaces.h"

using namespace cagd;
using namespace std;
```


© Ágoston Róth, 2018

```
11 // your default/special constructor
12 YourGLWidget::YourGLWidget()
13 {
14     try
15     {
16         // ...
17         // these lines coincide with the lines 15/285–81/286 and 120/287–126/287 of Listing 3.9/285
18         // ...


19         // Try to create different EC spaces:
20         _space_count = 6;                       // number of considered EC spaces;

21         _space.resize(_space_count);            // memory allocations;
22         _alpha.resize(_space_count);
23         _beta.resize(_space_count);
24         _n.resize(_space_count);
25         _img_B_basis.resize(_space_count);

26         _n[0] = 10;               // degree of the pure polynomial EC space (3.1/269);
27         _n[1] =  5;               // order of the pure trigonometric EC space (3.2/269);
28         _n[2] =  7;               // order of the pure hyperbolic EC space (3.3/269);
29         _n[3] =  4;               // order of the mixed algebraic-trigonometric EC space (3.4/269);
30         _n[4] =  3;               // order of the mixed algebraic-exponential-trigonometric EC space (3.5/269);
31         _n[5] = 10;               // order of the mixed algebraic-exponential-trigonometric EC space (3.6/269);

32         _alpha[0] = 0.0,          _beta[0] = 1.0;
33         _alpha[1] = 0.0,          _beta[1] = HALF_PI;
34         _alpha[2] = 0.0,          _beta[2] = PI;        // endpoints of corresponding definition domains;
35         _alpha[3] = 0.0,          _beta[3] = TWO_PI;
36         _alpha[4] = -1.25 * PI,   _beta[4] = 1.25 * PI;
37         _alpha[4] = 0.0,          _beta[4] = 16.694941067922716 / 2.0;

38         // additional shape parameters of the mixed algebraic-exponential-trigonometric EC space (3.6/269);
39         double a  = 1.0, b = 0.2;

40         _space[0] = SP<ECSpace>::Default( // dynamical allocation of the EC space P_10^{0,1};
41                 new (nothrow) PolynomialECSpace(_alpha[0], _beta[0], _n[0]));
42         _space[1] = SP<ECSpace>::Default( // dynamical allocation of the EC space T_10^{0,π/2};
43                 new (nothrow) TrigonometricECSpace(_alpha[1], _beta[1], _n[1]));
44         _space[2] = SP<ECSpace>::Default( // dynamical allocation of the EC space H_14^{0,π};
45                 new (nothrow) HyperbolicECSpace(_alpha[2], _beta[2], _n[2]));
46         _space[3] = SP<ECSpace>::Default( // dynamical allocation of the EC space AT_24^{0,2π};
47                 new (nothrow) ATECSpace(_alpha[3], _beta[3], _n[3]));
48         _space[4] = SP<ECSpace>::Default( // dynamical allocation of the EC space AET_27^{-5π/4,5π/4};
49                 new (nothrow) AETECSpace(_alpha[4], _beta[4], _n[4]));
50         _space[5] = SP<ECSpace>::Default( // dynamical allocation of the EC space M_{14,1,0.2}^{0,8.347470533}.
51                 new (nothrow) BrilleaudMazureSpace(_alpha[5], _beta[5], _n[5], a, b));

52         // Parameters used for the image generation of the normalized B-basis functions:

53         // the common number of subdivision points in each definition domain;
54         _div_point_count = 200;

55         // the maximum order of derivatives that have to be evaluated along the basis functions;
56         _maximum_order_of_derivatives = 1;

57         // generate dynamicaly the row matrices that store the image smart pointers of all normalized
58         // B-basis functions of each EC space.
59         for (GLuint i = 0; i < _space.size(); i++)
60         {
61             string th;

62             switch (i % 5)
63             {
64             case 1:  th = "st"; break;
65             case 2:  th = "nd"; break;
66             case 3:  th = "rd"; break;
67             default: th = "th"; break;
68             }

69             if (!_space[i])
```


© Ágoston Róth, 2018

```
70              {
71                  throw Exception("The " + toString(i) + th +
72                                  " EC space could not be created!");
73              }
74              _img_B_basis[i] = SP< RowMatrix<SP<GenericCurve3>::Default> >::Default(
75                  _space[i]->generateImagesOfAllBasisFunctions(
76                      ECSpace::B_BASIS, _maximum_order_of_derivatives, _div_point_count));
77              // Remark
78              // If instead of the normalized B-basis functions one intends to generate the images of all
79              // ordinary basis functions, then in line 76/293 of the current listing one should use the constant
80              // ECSpace::ORDINARY_BASIS instead of ECSpace::B_BASIS.
81              if (!_img_B_basis[i])
82              {
83                  throw Exception("Could not generate the image of all normalized "
84                                  "B-basis functions of the " + toString(i) +
85                                  th + " EC space!");
86              }
87              RowMatrix<SP<GenericCurve3>::Default> &reference = *_img_B_basis[i];
88              for (GLint j = 0; j < reference.columnCount(); j++)
89              {
90                  if (reference[j])
91                  {
92                      if (!reference[j]->updateVertexBufferObjects())
93                      {
94                          throw Exception("Could not update the VBOs of a normalized "
95                                          "B-basis function!");
96                      }
97                  }
98              }
99          }
100         // If required, one can differentiate individual ordinary or B-basis functions, e.g.,
101         GLint     space_index            = 1;
102         GLint     function_index         = _n[space_index];
103         GLint     differentiation_order  = 2;
104         GLdouble  parameter              = (_alpha[space_index]+_beta[space_index])/2.0;
105         GLdouble d_ordinary = (*_space[space_index])(
106             ECSpace::ORDINARY_BASIS, function_index, differentiation_order, parameter);
107         GLdouble d_B_basis  = (*_space[space_index])(
108             ECSpace::B_BASIS, function_index, differentiation_order, parameter);
109         // Do something nice and meaningful with the obtained derivatives d_ordinary and d_B_basis...
110         // For verification purposes, variables d_order and d_B_basis should contain the numerical values of
111         // the constants $\frac{\mathrm{d}^2}{\mathrm{d}u^2}\cos(3u)\Big|_{u=\frac{\pi}{4}} = \frac{9\sqrt{2}}{2} \approx 6.36396$ and $\frac{\mathrm{d}^2}{\mathrm{d}u^2}\left[c_{10,5}\sin^5\left(\frac{\beta-u}{2}\right)\sin^5\left(\frac{u}{2}\right)\right]\Big|_{\beta=\frac{\pi}{2},u=\frac{\pi}{4}} =$
112         // $2220 - \frac{3145\sqrt{2}}{2} \approx -3.85083$, respectively, where $c_{10,5} = 2368\sqrt{2}$.
113         // For debugging purposes one can also list the LaTeX expressions of the ordinary basis functions, e.g.,
114         string expression;
115         for (GLint i = 0; i < _space[space_index]->dimension(); i++)
116         {
117             if (_space[space_index]->LaTeXExpression(i, expression))
118             {
119                 cout << expression << endl;
120             }
121         }
122         glClearColor(1.0, 1.0, 1.0, 1.0);          // set the background color
123         glEnable(GL_DEPTH_TEST);                   // enable depth testing
124         glEnable(GL_LINE_SMOOTH);                  // enable the anti-aliasing of line primitives
125         glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);
126     }
127     catch (Exception &e)
128     {
129         cout << e << endl;
130     }
131 }
132 GLboolean YourGLWidget::_updateTransformationMatrices()
```


© Ágoston Róth, 2018

```
{
    // ...
    // these lines coincide with the lines 224/288–233/289 of Listing 3.9/285
    // ...
}

// your rendering method
void YourGLWidget::render()
{
    // clear the color and depth buffers
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    _color_shader.enable();

    // revert to the original projection-view-model matrix
    _color_shader.setUniformMatrix4fv("PVM", 1, GL_FALSE, _PVM.address());

    for (GLuint i = 0; i < _space.size(); i++)
    {
        if (_img_B_basis[i])
        {
            // constant reference to the ith normalized B-basis function of the current EC space
            const RowMatrix<SP<GenericCurve3>::Default> &reference = *_img_B_basis[i];

            // in order to render each system of B-basis functions at different positions, we will use local
            // translation matrices, based on which we also temporarily update the projection-view-model
            // matrix of the color shader program
            Translate         local_T(-(_alpha[i] + _beta[i]) / 2.0, 1.7 - 0.75 * i, 0.0);
            GLTransformation local_PVM = _PVM * local_T;
            _color_shader.setUniformMatrix4fv("PVM", 1, GL_FALSE, local_PVM.address());

            for (GLint j = 0; j < reference.columnCount(); j++)
            {
                // each B-basis function will be rendered by using different colors
                _color_shader.setUniformColor(
                        "color", coldToHotColormap(j, 0, reference.columnCount() - 1));
                reference[j]->renderDerivatives(_color_shader, 0, GL_LINE_STRIP);

                // uncomment the line below in order to see the tangent vectors of the basis functions
                // reference[j]->renderDerivatives(_color_shader, 1, GL_LINES);
            }
        }
    }

    _color_shader.disable();

    // Fig. 3.1/295 illustrates the image obtained at run-time.
}
```

3.4 Defining and using specialized B-curves

The next two subsections provide examples for the description and manipulation of B-curves defined in different types of EC spaces.

3.4.1 B-curve generation, order elevation and subdivision

EC space objects can also be used to define, evaluate, differentiate, manipulate and render B-curves of type (1.19/4) and to perform order elevations and subdivisions on them as it is illustrated in Listings 3.12/294 and 3.13/296 in case of five different EC spaces.

Listing 3.12. Defining, rendering, order elevating and subdividing B-curves (**YourGLWidget.h**)

```
#ifndef YOURGLWIDGET_H
#define YOURGLWIDGET_H

#include <GL/glew.h> // in order to handle vertex buffer and shader program objects, we rely on GLEW
```


© Ágoston Róth, 2018

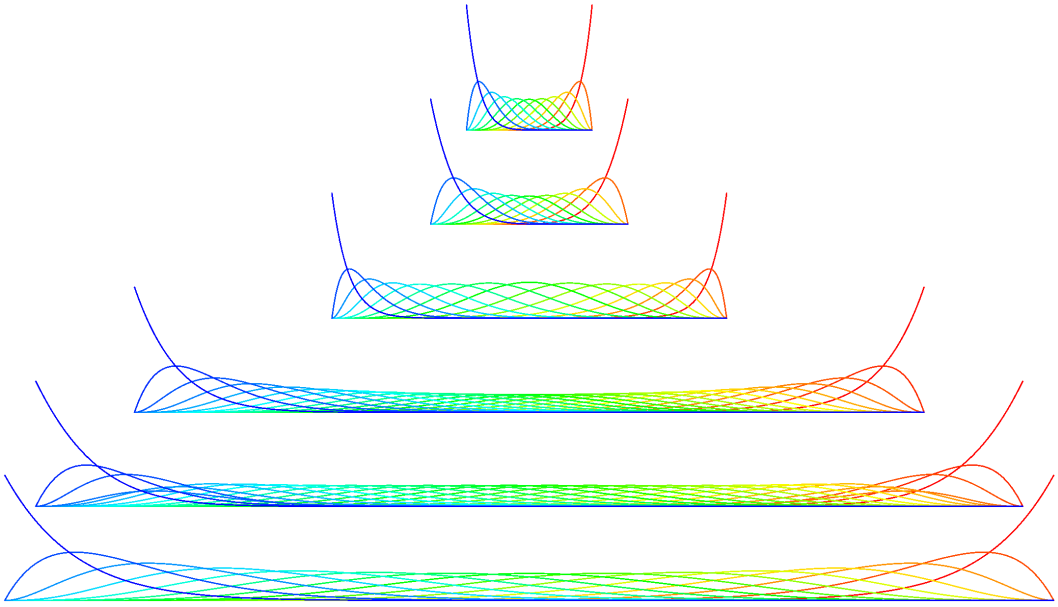


Fig. 3.1: From top to bottom: evaluation and rendering of the unique normalized B-bases of the EC spaces $\mathbb{P}_{10}^{0,1}$, $\mathbb{T}_{10}^{0,\frac{\pi}{2}}$, $\mathbb{H}_{14}^{0,\pi}$, $\mathbb{AT}_{24}^{0,2\pi}$, $\mathbb{AET}_{27}^{-5\pi/4,5\pi/4}$ and $\mathbb{M}_{14,1,0.2}^{0,\beta}$ that are of types (3.1/269), (3.2/269), (3.3/269), (3.4/269), (3.5/269) and (3.6/269), respectively, where $\beta = \frac{1}{2} \cdot 16.694941067922716 = 8.347470533961358$. (The figure was generated by the source codes listed in Listings 3.10/291 and 3.11/291.)

```
4                       // we assume that this external dependency is added to your project

5  #include <Core/Geometry/Curves/GenericCurves3.h>
6  #include <Core/Math/SpecialGLTransformations.h>
7  #include <Core/Shaders/ShaderPrograms.h>
8  #include <Core/SmartPointers/SpecializedSmartPointers.h>
9  #include <EC/ECSpaces.h>
10 #include <EC/BCurves3.h>

11 namespace cagd // all data structures in the proposed function library are defined under the namespace cagd
12 {
13     class YourGLWidget: public ...
14     {
15     private:
16         // ...
17         // these lines coincide with the lines 15/284–35/284 of Listing 3.8/284
18         // ...

19         // Parameters of different EC spaces:
20         GLint                                  _space_count;   // number of considered EC spaces;
21         std::vector<GLdouble>                  _alpha, _beta; // endpoints of definition domains;
22         std::vector<GLint>                     _n;            // orders of considered EC spaces;
23         std::vector<SP<ECSpace>::Default>      _space;        // smart pointers to different EC spaces;
24         std::vector<Color4>                    _color;        // colors associated with different EC spaces.

25         // number of uniform subdivision points in the definition domains, it is used for image generation
26         GLint                                  _div_point_count;
27         // determines the maximum order of derivatives that have to be evaluated along the B-curves
28         GLint                                  _maximum_order_of_derivatives;

29         // a vector of smart pointers to dynamically allocated B-curves
30         std::vector<SP<BCurve3>::Default>      _bcurve;
31         // a vector of smart pointers to dynamically allocated B-curve images (i.e., GenericCurve3 objects)
32         std::vector<SP<GenericCurve3>::Default> _img_bcurve;
```


© Ágoston Róth, 2018

```
33          // a vector of smart pointers to dynamically allocated order elevated B-curves
34          std::vector<SP<BCurve3>::Default>          _oe_bcurve;
35          // a vector of smart pointers to dynamically allocated order elevated B-curve images
36          std::vector<SP<GenericCurve3>::Default> _img_oe_bcurve;

37          // determines the subdivision point of the definition domains, where the initial B-curves have to be subdivided
38          GLdouble                                                    _ratio;
39          // a vector of smart pointers to row matrices that store two smart pointers to the subdivided arcs
40          std::vector< SP< RowMatrix<SP<BCurve3>::Default> >::Default > _subdivision;
41          // a vector of row matrices that store two smart pointers to the images of the subdivided arcs
42          std::vector< RowMatrix<SP<GenericCurve3>::Default> >        _img_subdivision;

43      public:
44          // your default/special constructor
45          YourGLWidget(/*...*/);

46          // your rendering method
47          void render();

48          // your other methods...
49      };
50  }

51  #endif // YOURGLWIDGET_H
```

Listing 3.13. Defining, rendering, order elevating and subdividing B-curves (**YourGLWidget.cpp**)

```
1   #pragma warning(disable:4503)
2   #include "YourGLWidget.h"

3   #include <iostream>
4   #include <fstream>

5   #include <Core/Exceptions.h>
6   #include <Core/Utilities.h>
7   #include <Core/Math/Constants.h>

8   #include "../Spaces/SpecializedECSpaces.h"

9   using namespace cagd;
10  using namespace std;

11  // your default/special constructor
12  void YourGLWidget::YourGLWidget(/*...*/)
13  {
14      try
15      {
16          // ...
17          // these lines coincide with the lines 15/285–81/286 and 120/287–126/287 of Listing 3.9/285
18          // ...

19          // Remark
20          //
21          // Several methods of classes ECSpace, BCurve3 and BSurface3 expect a boolean flag named
22          // check_for_ill_conditioned_matrices and a non-negative integer named expected_correct_significant_digits.
23          // These variables appear, e.g., in the input argument lists of the constructor of EC spaces and of the
24          // methods that perform either order elevation or subdivision on B-curves and B-surfaces.
25          //
26          // If the flag check_for_ill_conditioned_matrices is set to true, these methods will calculate the condition
27          // number of each matrix that appears in the construction of the unique normalized B-basis of the underlying
28          // EC space. Using singular value decomposition, each condition number is determined as the ratio of the
29          // largest and smallest singular values of the corresponding matrices.
30          //
31          // If at least one of the obtained condition numbers is too large, i.e., when the number of estimated
32          // correct significant digits is less than number of expected ones, these methods will throw an exception
33          // that states that one of the systems of linear equations is ill-conditioned and therefore its solution
34          // may be not accurate.
35          //
36          // If the user catches such an exception, one can try:
37          //   1) to lower the number of expected correct significant digits;
38          //   2) to decrease the dimension of the underlying EC space;
```


© Ágoston Róth, 2018

```
39          //   3) to change the endpoints of the definition domain [α, β];
40          //   4) to run the code without testing for ill-conditioned matrices and hope for the best.
41          //
42          // Note that the standard condition number may lead to an overly pessimistic estimate for the overall error
43          // and, by activating this boolean flag, the run-time of these methods will increase. Several numerical tests
44          // show that ill-conditioned matrices appear only when one defines EC spaces with relatively big dimensions.
45          // Considering that, in practice, curves and surfaces are mostly composed of smoothly joined lower order arcs
46          // and patches, by default we opted for speed, i.e., initially the flag check_for_ill_conditioned_matrices is set
47          // to false. Naturally, if one obtains mathematically or geometrically unexpected results, then one should
48          // (also) study the condition numbers mentioned above.
49          GLboolean check_for_ill_conditioned_matrices  = GL_FALSE;
50          GLint     expected_correct_significant_digits = 3;

51          // Try to create different EC spaces:
52          _space_count = 5;                         // number of considered EC spaces;

53          _space.resize(_space_count);              // memory allocations;
54          _alpha.resize(_space_count);
55          _beta.resize(_space_count);
56          _n.resize(_space_count);

57          _color.resize(_space_count);

58          _bcurve.resize(_space_count);
59          _img_bcurve.resize(_space_count);

60          _oe_bcurve.resize(_space_count);
61          _img_oe_bcurve.resize(_space_count);

62          _subdivision.resize(_space_count);
63          _img_subdivision.resize(_space_count);

64          // orders of the EC spaces (3.1/269), (3.2/269), (3.3/269), (3.4/269) and (3.6/269), respectively;
65          _n[0] = 6; _n[1] = 3; _n[2] = 3; _n[3] = 2; _n[4] = 2;

66          // endpoints of the corresponding definition domains;
67          _alpha[0] = 0.0, _beta[0] = 1.0;
68          _alpha[1] = 0.0, _beta[1] = HALF_PI;
69          _alpha[2] = 0.0, _beta[2] = PI;
70          _alpha[3] = 0.0, _beta[3] = 4.0 * PI / 3.0;
71          _alpha[4] = 0.0, _beta[4] = 16.694941067922716;

72          // additional shape parameters of the mixed algebraic-exponential-trigonometric EC space (3.6/269);
73          double a  = 1.0, b = 0.2;

74          _space[0] = SP<ECSpace>::Default(     // dynamical allocation of the EC space P_6^{0,1};
75              new (nothrow) PolynomialECSpace(
76                _alpha[0], _beta[0], _n[0],
77                check_for_ill_conditioned_matrices, expected_correct_significant_digits));
78          _space[1] = SP<ECSpace>::Default(     // dynamical allocation of the EC space T_6^{0,π/2};
79              new (nothrow) TrigonometricECSpace(
80                _alpha[1], _beta[1], _n[1],
81                check_for_ill_conditioned_matrices, expected_correct_significant_digits));
82          _space[2] = SP<ECSpace>::Default(     // dynamical allocation of the EC space H_6^{0,π};
83              new (nothrow) HyperbolicECSpace(
84                _alpha[2], _beta[2], _n[2],
85                check_for_ill_conditioned_matrices, expected_correct_significant_digits));
86          _space[3] = SP<ECSpace>::Default(     // dynamical allocation of the EC space AT_8^{0,4π/3};
87              new (nothrow) ATECSpace(
88                _alpha[3], _beta[3], _n[3],
89                check_for_ill_conditioned_matrices, expected_correct_significant_digits));
90          _space[4] = SP<ECSpace>::Default(   // dynamical allocation of the EC space M_{6,1,0.2}^{0,16.694941067922716}.
91            new (nothrow) BrilleaudMazureSpace(
92                _alpha[4], _beta[4], _n[4], a, b,
93                check_for_ill_conditioned_matrices, expected_correct_significant_digits));

94          // colors associated with different EC spaces
95          for (GLint i = 0; i < _space_count; i++)
96          {
97              _color[i] = coldToHotColormap(i, 0, _space_count - 1);
98          }
```


© Ágoston Róth, 2018

```
99          // we will evaluate the zeroth, first and second order derivatives at 50 uniform subdivision points
100         _maximum_order_of_derivatives =  2;
101         _div_point_count              = 50;
102         // determines the subdivision point of the definition domains, where the initial B-curves have to be subdivided
103          _ratio                       = 0.5;
104         // labels associated with the left and right arcs of a subdivided B-curve
105         string direction[2] = {"left", "right"};
106         // generating B-curves, performing order elevations and subdivisions on them
107         for (GLint i = 0; i < _space_count; i++)
108         {
109             string th;
110             switch (i % 5)
111             {
112             case 1:  th = "st"; break;
113             case 2:  th = "nd"; break;
114             case 3:  th = "rd"; break;
115             default: th = "th"; break;
116             }
117             if (!_space[i])
118             {
119                 throw Exception("The " + toString(i) + th +
120                                 " EC space could not be created!");
121             }
122             _bcurve[i] = SP<BCurve3>::Default(new (nothrow) BCurve3(*_space[i]));
123             if (!_bcurve[i])
124             {
125                 throw Exception("Could not genereate the " + toString(i) + th +
126                                 " EC B-curve!");
127             }
128             // the control points of each B-curve will be obtained by uniform sampling the unit circle
129             GLdouble step = TWO_PI / _bcurve[i]->dataCount();
130             for (GLint j = 0; j < _bcurve[i]->dataCount(); j++)
131             {
132                 GLdouble u = j * step;
133                 Cartesian3 &reference = (*_bcurve[i])[j];
134                 reference[0] = -cos(u);
135                 reference[1] =  sin(u);
136             }
137             // updating the VBOs of the ith control polygon
138             if (!_bcurve[i]->updateVertexBufferObjectsOfData())
139             {
140                 throw Exception("Could not update the VBO of the " +
141                                 toString(i) + th + " control polygon!");
142             }
143             // generating the image of the ith B-curve and updating its VBOs
144             _img_bcurve[i] = SP<GenericCurve3>::Default(
145                     _bcurve[i]->generateImage(
146                             _maximum_order_of_derivatives, _div_point_count));
147             if (!_img_bcurve[i])
148             {
149                 throw Exception("Could not generate the image of the " +
150                                 toString(i) + th + " B-curve!");
151             }
152             if (!_img_bcurve[i]->updateVertexBufferObjects())
153             {
154                 throw Exception("Could not update the VBOs of the " +
155                                 toString(i) + th + " B-curve's image!");
156             }
```


© Ágoston Róth, 2018

```
157             // generating the ith order elevated B-curve: we insert the complex zero z = 0 of mulitplicity n_i + 2
158             _oe_bcurve[i] = SP<BCurve3>::Default(
159                     _bcurve[i]->performOrderElevation(
160                             0.0, 0.0, _n[i] + 2,
161                             check_for_ill_conditioned_matrices,
162                             expected_correct_significant_digits));
163             if (!_oe_bcurve[i])
164             {
165                 throw Exception("Could not perform order elevation on the " +
166                                 toString(i) + th + " B-curve!");
167             }
168             // updating the VBOs of the ith order elevated control polygon
169             if (!_oe_bcurve[i]->updateVertexBufferObjectsOfData())
170             {
171                 throw Exception("Could not update the VBO of the " + toString(i) + th +
172                                 " order elevated B-curve's control polygon!");
173             }
174             // generating the image of the ith order elevated B-curve and updating its VBOs
175             _img_oe_bcurve[i] = SP<GenericCurve3>::Default(
176                     _oe_bcurve[i]->generateImage(
177                             _maximum_order_of_derivatives, _div_point_count));
178             if (!_img_oe_bcurve[i])
179             {
180                 throw Exception("Could not generate the image of the " +
181                                 toString(i) + th + " order elevated B-curve!");
182             }
183             if (!_img_oe_bcurve[i]->updateVertexBufferObjects())
184             {
185                 throw Exception("Could not update the VBOs of the " +
186                                 toString(i) + th + " order elevated B-curve's image!");
187             }
188             // subdivision of the initial B-curves
189             _subdivision[i] = SP< RowMatrix<SP<BCurve3>::Default> >::Default(
190                     _bcurve[i]->performSubdivision(
191                         _alpha[i] + _ratio * (_beta[i] - _alpha[i]),
192                         check_for_ill_conditioned_matrices,
193                         expected_correct_significant_digits));
194             if (!_subdivision[i])
195             {
196                 throw Exception("Could not perform subdivision on the " +
197                                 toString(i) + th + " B-curve!");
198             }
199             _img_subdivision[i].resizeColumns(2);
200             for (GLint j = 0; j < 2; j++)
201             {
202                 if (!(*_subdivision[i])[j]->updateVertexBufferObjectsOfData())
203                 {
204                     throw Exception("Could not update the VBO of the control polygon "
205                                     "of the " + direction[j] + " arc of the " +
206                                     toString(i) + th + " subdivided B-curve!");
207                 }
208                 _img_subdivision[i][j] = SP<GenericCurve3>::Default(
209                         (*_subdivision[i])[j]->generateImage(
210                             _maximum_order_of_derivatives, _div_point_count));
211                 if (!_img_subdivision[i][j])
212                 {
213                     throw Exception("Could not generate the image of the " +
214                                     direction[j] + " arc of the " + toString(i) + th +
215                                     " subdivided B-curve!");
216                 }
217                 if (!_img_subdivision[i][j]->updateVertexBufferObjects())
```


© Ágoston Róth, 2018

```
218                 {
219                     throw Exception("Could not update the VBOs of the image of the " +
220                                     direction[j] + " arc of the " + toString(i) + th +
221                                     " subdivided B-curve!");
222                 }
223             }
224         }

225         glClearColor(1.0, 1.0, 1.0, 1.0);           // set the background color
226         glEnable(GL_DEPTH_TEST);                    // enable depth testing
227         glEnable(GL_LINE_SMOOTH);                   // enable the anti-aliasing of line primitives
228         glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);
229         glEnable(GL_POINT_SMOOTH);                  // enable the anti-aliasing of point primitives
230         glHint(GL_POINT_SMOOTH_HINT, GL_NICEST);
231     }
232     catch (Exception &e)
233     {
234         cout << e << endl;
235     }
236 }

237 GLboolean YourGLWidget::_updateTransformationMatrices()
238 {
239     // ...
240     // these lines coincide with the lines 224/288–233/289 of Listing 3.9/285
241     // ...
242 }

243 // your rendering method
244 void YourGLWidget::render()
245 {
246     // clear the color and depth buffers
247     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

248     _color_shader.enable();

249     GLfloat x_min = -4.5f, x_step = 11.0f / _space_count;

250     glLineWidth(2.0);
251     glPointSize(8.0);
252     for (GLint i = 0; i < _space_count; i++)
253     {
254         if (_bcurve[i] && _img_bcurve[i] &&
255             _oe_bcurve[i] && _img_oe_bcurve[i] &&
256             _subdivision[i])
257         {
258             // rendering the control polygons and images of the initial B-curves
259             _color_shader.setUniformColor("color", _color[i]);

260             Translate         local_T(x_min + i * x_step, +2.5f, 0.0f);
261             GLTransformation local_PVM = _PVM * local_T;
262             _color_shader.setUniformMatrix4fv("PVM", 1, GL_FALSE, local_PVM.address());

263             _bcurve[i]->renderData(_color_shader, GL_LINE_STRIP);
264             _bcurve[i]->renderData(_color_shader, GL_POINTS);

265             _img_bcurve[i]->renderDerivatives(_color_shader, 0, GL_LINE_STRIP);

266             // uncomment the line below in order to see the tangent vectors of the ith B-curve
267             // _img_bcurve[i]->renderDerivatives(_color_shader, 1, GL_LINES);

268             // uncomment the line below in order to see the acceleration vectors of the ith B-curve
269             // _img_bcurve[i]->renderDerivatives(_color_shader, 2, GL_LINES);

270             // rendering the control polygons and images of the order elevated B-curves
271             local_T.setYDirectionalUnits(0.0f);
272             local_PVM = _PVM * local_T;

273             _color_shader.setUniformMatrix4fv("PVM", 1, GL_FALSE, local_PVM.address());

274             _oe_bcurve[i]->renderData(_color_shader, GL_LINE_STRIP);
275             _oe_bcurve[i]->renderData(_color_shader, GL_POINTS);

276             _img_oe_bcurve[i]->renderDerivatives(_color_shader, 0, GL_LINE_STRIP);
```


© Ágoston Róth, 2018

```
277             // for the sake of comparison, we re-render the control polygons of the initial B-curves
278             _color_shader.setUniformColor("color", colors::gray);
279             glEnable(GL_LINE_STIPPLE);
280                 glLineStipple(1, 0xf0f0);
281                 _bcurve[i]->renderData(_color_shader, GL_LINE_STRIP);
282                 _bcurve[i]->renderData(_color_shader, GL_POINTS);
283             glDisable(GL_LINE_STIPPLE);

284             // rendering the control polygons and images of the subdivided arcs of the initial B-curves
285             local_T.setYDirectionalUnits(-2.5f);
286             local_PVM = _PVM * local_T;
287             _color_shader.setUniformMatrix4fv("PVM", 1, GL_FALSE, local_PVM.address());

288             for (GLint j = 0; j < 2; j++)
289             {
290                 _color_shader.setUniformColor("color", _color[(i + j) % _space_count]);

291                 (*_subdivision[i])[j]->renderData(_color_shader, GL_LINE_STRIP);
292                 (*_subdivision[i])[j]->renderData(_color_shader, GL_POINTS);

293                 _img_subdivision[i][j]->renderDerivatives(
294                         _color_shader, 0, GL_LINE_STRIP);
295             }

296             // for the sake of comparison, we re-render the control polygons of the initial B-curves
297             _color_shader.setUniformColor("color", colors::gray);
298             glEnable(GL_LINE_STIPPLE);
299                 glLineStipple(1, 0xf0f0);
300                 _bcurve[i]->renderData(_color_shader, GL_LINE_STRIP);
301                 _bcurve[i]->renderData(_color_shader, GL_POINTS);
302             glDisable(GL_LINE_STIPPLE);
303         }
304     }
305     glPointSize(1.0);
306     glLineWidth(1.0);

307     // revert to the original projection-view-model matrix
308     _color_shader.setUniformMatrix4fv("PVM", 1, GL_FALSE, _PVM.address());

309     // render other geometries...

310     _color_shader.disable();

311     // Fig. 3.2/302 illustrates the image obtained at run-time.
312 }
```

3.4.2 B-representation of ordinary integral curves

B-curves can also be used to provide control point based exact descriptions (or B-representations) for ordinary integral curves of type (1.48/11) as it is illustrated in Listings 3.14/302 and 3.15/303, which at first define the EC space

$$\mathbb{S}_4^{0,\beta} = \left\langle \left\{ \varphi_{4,0}(u) \equiv 1, \varphi_{4,1}(u) = e^{-\omega u}\cos(u), \varphi_{4,2}(u) = e^{-\omega u}\sin(u), \right.\right. \\ \left.\left. \varphi_{4,3}(u) = e^{\omega u}\cos(u), \varphi_{4,4}(u) = e^{\omega u}\sin(u) : u \in [0,\beta] \right\} \right\rangle, \ \beta \in (0,\pi]$$

where $\omega = \frac{1}{3\pi}$, then they also generate B-curves for the B-representations of the logarithmic spiral arcs

$$\mathbf{c}_k(u) = \begin{bmatrix} e^{\omega(u+k\beta)}\cos(u+k\beta) \\ e^{\omega(u+k\beta)}\sin(u+k\beta) \end{bmatrix} \\ = e^{k\omega\beta}\begin{bmatrix} \cos(k\beta) \\ \sin(k\beta) \end{bmatrix}\varphi_{4,3}(u) + e^{k\omega\beta}\begin{bmatrix} -\sin(k\beta) \\ \cos(k\beta) \end{bmatrix}\varphi_{4,4}(u), \ u \in [0,\beta], \ k = 0,\dots,6. \quad (3.8)$$

© Ágoston Róth, 2018

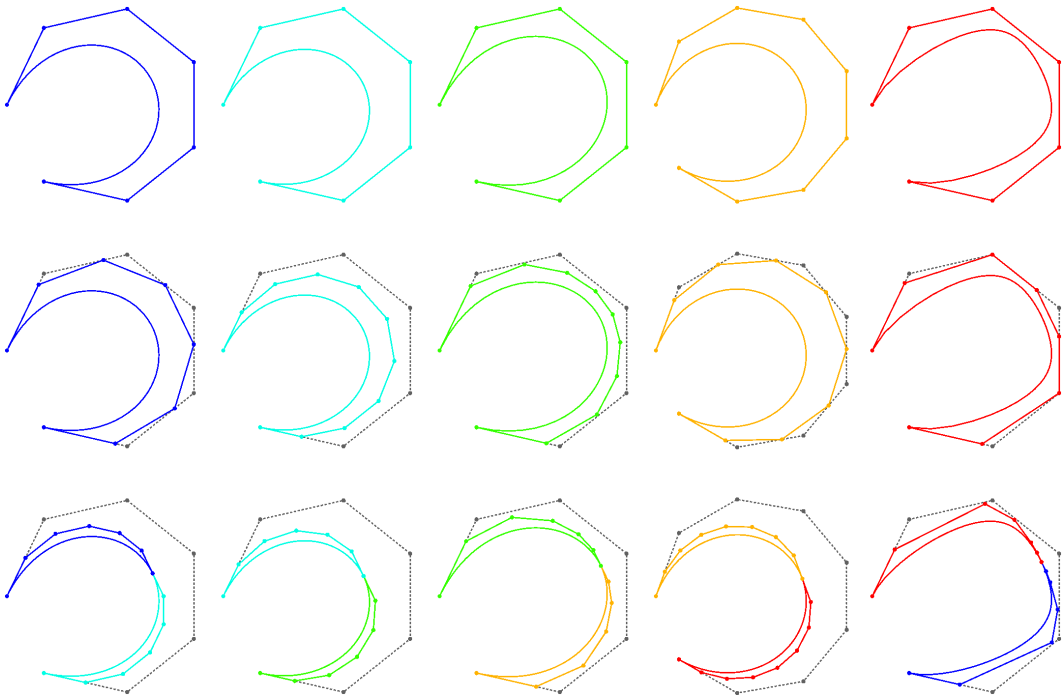


Fig. 3.2: From left to right: the columns illustrate the order elevation and subdivision of different B-curves in EC spaces $\mathbb{P}_6^{0,1}$, $\mathbb{T}_6^{0,\frac{\pi}{2}}$, $\mathbb{H}_6^{0,\pi}$, $\mathbb{AT}_8^{0,\frac{4\pi}{3}}$ and $\mathbb{M}_{6,1,0.2}^{0,16.694941067922716}$ that are of types (3.1/269), (3.2/269), (3.3/269), (3.4/269) and (3.6/269), respectively. (The figure was generated by the source codes listed in Listings 3.12/294 and 3.13/296.)

Listing 3.14. B-representations of ordinary integral curves (**YourGLWidget.h**)

```
1  #ifndef YOURGLWIDGET_H
2  #define YOURGLWIDGET_H

3  #include <GL/glew.h> // in order to handle vertex buffer and shader program objects, we rely on GLEW
4                       // we assume that this external dependency is added to your project

5  #include <Core/Geometry/Curves/GenericCurves3.h>
6  #include <Core/Math/SpecialGLTransformations.h>
7  #include <Core/Shaders/ShaderPrograms.h>
8  #include <Core/SmartPointers/SpecializedSmartPointers.h>
9  #include <EC/ECSpaces.h>
10 #include <EC/BCurves3.h>

11 namespace cagd // all data structures in the proposed function library are under the namespace cagd
12 {
13     class YourGLWidget: public ...
14     {
15     private:
16         // ...
17         // these lines coincide with the lines 15/284–35/284 of Listing 3.8/284
18         // ...

19         // EC space parameters:
20         GLdouble                                    _alpha, _beta; // endpoints;
21         GLdouble                                    _omega;        // shape parameter;
22         SP<ECSpace>::Default                        _space;        // smart pointer to the EC space.

23         // number of uniform subdivision points in the definition domains, it is used for image generation
24         GLint                                       _maximum_order_of_derivatives;
```


© Ágoston Róth, 2018

```
25          // determines the maximum order of derivatives that have to be evaluated along the B-curves
26          GLint                                       _div_point_count;

27          // number of arcs along the logarithmic spireal (3.8/301)
28          GLint                                       _arc_count;
29          // a vector of smart pointers to dynamically allocated B-curves
30          std::vector<SP<BCurve3>::Default>           _bcurve;
31          // a vector of smart pointers to dynamically allocated B-curve images (i.e., GenericCurve3 objects)
32          std::vector<SP<GenericCurve3>::Default>     _img_bcurve;

33      public:
34          // your default/special constructor
35          YourGLWidget(/*...*/);

36          // your rendering method
37          void render();

38          // your other methods...
39      };
40  }

41  #endif // YOURGLWIDGET_H
```

Listing 3.15. B-representations of ordinary integral curves (**YourGLWidget.cpp**)

```
1   #pragma warning(disable:4503)
2   #include "YourGLWidget.h"

3   #include <fstream>
4   #include <iostream>


5   #include <Core/Exceptions.h>
6   #include <Core/Utilities.h>
7   #include <Core/Math/Constants.h>
8   #include <Core/Utilities.h>

9   using namespace cagd;
10  using namespace std;

11  // your default/special constructor
12  YourGLWidget::YourGLWidget(/*...*/)
13  {
14      try
15      {
16          //...
17          // these lines coincide with the lines 15/285–81/286 and 120/287–126/287 of Listing 3.9/285
18          //...

19          //...
20          // these lines coincide with the lines 19/296–50/297 of Listing 3.13/296
21          //...

22          _alpha = 0.0;
23          _beta  = 5.0 * PI / 6.0;
24          _space = SP<ECSpace>::Default(new ECSpace(_alpha, _beta,
25                                                    check_for_ill_conditioned_matrices,
26                                                    expected_correct_significant_digits));

27          _omega = 1.0 / 3.0 / PI;

28          // the order of the following zero insertions is important, since it determines the order of the ordinary
29          // basis functions and consequently the order of their coefficients in case of B-representations
30          _space->insertZero(-_omega, 1.0, 1, false);
31          _space->insertZero(+_omega, 1.0, 1, false);

32          if (!_space->updateBothBases(check_for_ill_conditioned_matrices,
33                                       expected_correct_significant_digits))
34          {
35              throw Exception("Could not update the normalized B-basis of the EC space!");
36          }
```


© Ágoston Róth, 2018

```
37          // If no exception occured so far, one should have the ordinary basis functions φ_{4,0}(u) ≡ 1,
38          // φ_{4,1}(u) = e^{−ωu} cos(u), φ_{4,2}(u) = e^{−ωu} sin(u), φ_{4,3}(u) = e^{ωu} cos(u), φ_{4,4}(u) = e^{ωu} sin(u),
39          // where u ∈ [0, β].

40          // memory allocations
41          _arc_count = 7;
42          _bcurve.resize(_arc_count);
43          _img_bcurve.resize(_arc_count);

44          // we will evaluate the zeroth and first order derivatives at 30 uniform subdivision points
45          _maximum_order_of_derivatives = 1;
46          _div_point_count = 30;

47          // multi-threaded operations performed on the CPU-side (if one is convinced that there are no errors
48          // in the lines 50/304–112/305 below, variables B_representations_aborted and reason can be eliminated from
49          // the code in order to further increase the performance)
50          bool     B_representations_aborted = false;
51          string   reason;

52          #pragma omp parallel for
53          for (GLint k = 0; k < _arc_count; k++)
54          {
55              #pragma omp flush (B_representations_aborted)
56              if (!B_representations_aborted)
57              {
58                  string th;

59                  switch (k % 5)
60                  {
61                  case 1:  th = "st"; break;
62                  case 2:  th = "nd"; break;
63                  case 3:  th = "rd"; break;
64                  default: th = "th"; break;
65                  }

66                  // a smart pointer to the B-representation of the kth arc of the logarithmic spiral (3.8/301)
67                  _bcurve[k] = SP<BCurve3>::Default(new BCurve3(*_space));

68                  if (!_bcurve[k])
69                  {
70                      B_representations_aborted = true;
71                      #pragma omp flush (B_representations_aborted)
72                      reason = "Could not create the " + toString(k) + th + " B-arc!";
73                  }
74                  else
75                  {
76                      // calculating the coefficients of the ordinary basis functions cf. formula (3.8/301), i.e.,
77                      // λ_i = [0; 0], i = 0, 1, 2, λ_3 = e^{kωβ} [cos(kβ); sin(kβ)], λ_4 = e^{kωβ} [−sin(kβ); cos(kβ)]
78                      GLdouble scale = exp(k * _omega * _beta);

79                      RowMatrix<Cartesian3> lambda(_space->dimension());
80                      lambda[3][0]  =  scale * cos(k * _beta);
81                      lambda[3][1]  =  scale * sin(k * _beta);
82                      lambda[4][0]  = -lambda[3][1];
83                      lambda[4][1]  =  lambda[3][0];

84                      // updating the control points for B-representation
85                      if (!_bcurve[k]->updateControlPointsForExactDescription(lambda))
86                      {
87                          B_representations_aborted = true;
88                          #pragma omp flush (B_representations_aborted)
89                          reason = "Could not perform the " + toString(k) + th +
90                                   " B-representation!";
91                      }
92                      else
93                      {
94                          // generating the image of the B-curve and updating its VBOs
95                          _img_bcurve[k] = SP<GenericCurve3>::Default(
96                              _bcurve[k]->generateImage(
97                                       _maximum_order_of_derivatives, _div_point_count));

98                          if (!_img_bcurve[k])
99                          {
```


© Ágoston Róth, 2018

```
100                                 B_representations_aborted = true;
101                                 #pragma omp flush (B_representations_aborted)
102                                 reason = "Could not generate the image of the " +
103                                         toString(k) + th + " B-representation!";
104                             }
105                         }
106                     }
107                 }
108             }

109             if (B_representations_aborted)
110             {
111                 throw Exception(reason);
112             } (

113             // sequential operations performed on the GPU-side
114             for (GLint k = 0; k < _arc_count; k++)
115             {
116                 string th;

117                 switch (k % 5)
118                 {
119                 case 1:  th = "st"; break;
120                 case 2:  th = "nd"; break;
121                 case 3:  th = "rd"; break;
122                 default: th = "th"; break;
123                 }

124                 // updating the VBO of the kth control polygon
125                 if (!_bcurve[k]->updateVertexBufferObjectsOfData())
126                 {
127                     throw Exception("Could not update the VBO of the control polygon of the "
128                                     + toString(k) + th + " B-curve!");
129                 }

130                 // updating the VBOs of the kth B-curve's image
131                 if (!_img_bcurve[k]->updateVertexBufferObjects())
132                 {
133                     throw Exception("Could not update the VBOs of the image of the " +
134                                     toString(k) + th + " B-curve!");
135                 }
136             }

137             glClearColor(1.0, 1.0, 1.0, 1.0);
138             glEnable(GL_DEPTH_TEST);
139             glEnable(GL_LINE_SMOOTH);
140             glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);
141             glEnable(GL_POINT_SMOOTH);
142             glHint(GL_POINT_SMOOTH_HINT, GL_NICEST);
143         }
144         catch (Exception &e)
145         {
146             cout << e << endl;
147         }
148     }

149     GLboolean YourGLWidget::_updateTransformationMatrices()
150     {
151         // ...
152         // these lines coincide with the lines 224/288–233/289 of Listing 3.9/285
153         // ...
154     }

155     // your rendering method
156     void YourGLWidget::render()
157     {
158         // clear the color and depth buffers
159         glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

160         _color_shader.enable();

161         // set the projection-view-model matrix
162         _color_shader.setUniformMatrix4fv("PVM", 1, GL_FALSE, _PVM.address());
```


© Ágoston Róth, 2018

```
163     // render the control polygons and images of the B-representations
164     glLineWidth(2.0);
165     glPointSize(8.0);
166     for (GLint k = 0; k < _arc_count; k++)
167     {
168         if (_bcurve[k] && _img_bcurve[k])
169         {
170             _color_shader.setUniformColor(
171                     "color", coldToHotColormap(k, 0, _arc_count - 1));
172             _bcurve[k]->renderData(_color_shader, GL_LINE_STRIP);
173             _bcurve[k]->renderData(_color_shader, GL_POINTS);
174             _img_bcurve[k]->renderDerivatives(_color_shader, 0, GL_LINE_STRIP);
175             _img_bcurve[k]->renderDerivatives(_color_shader, 1, GL_LINES);
176         }
177     }
178     glPointSize(1.0);
179     glLineWidth(1.0);
180     // render other geometries...
181     _color_shader.disable();
182     // Fig. 3.3/306 illustrates the image obtained at run-time.
183 }
```

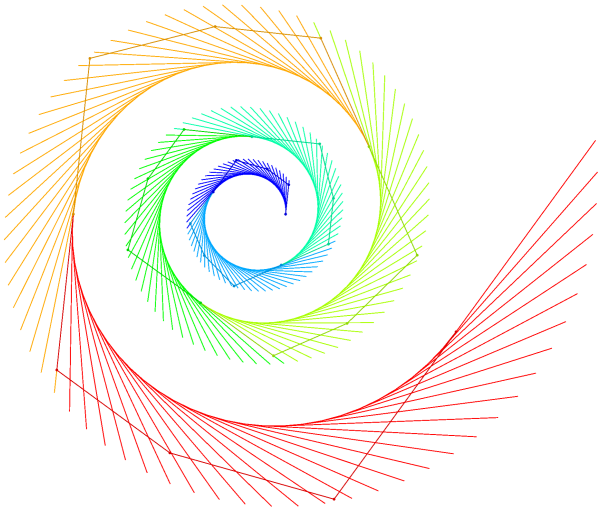


Fig. 3.3: Control point based exact description (or B-representation) of different arcs of a logarithmic spiral. (The figure was generated by the source codes given in Listings 3.14/302 and 3.15/303.)

3.5 Defining and using specialized B-surfaces

The next two subsections provide examples for the description and manipulation of B-surfaces defined in different types of EC spaces.

3.5.1 B-surface generation, order elevation and subdivision

Listings 3.16/307 and 3.17/308 provide examples for the generation, evaluation, order elevation, subdivision and rendering of B-surfaces.

© Ágoston Róth, 2018

Listing 3.16. Defining, rendering, order elevating and subdividing B-surfaces (**YourGLWidget.h**)

```
#ifndef YOURGLWIDGET_H
#define YOURGLWIDGET_H

#include <GL/glew.h>

#include <Core/SmartPointers/SpecializedSmartPointers.h>
#include <Core/Geometry/Surfaces/Lights.h>
#include <Core/Geometry/Surfaces/TriangleMeshes3.h>
#include <Core/Math/SpecialGLTransformations.h>
#include <Core/Shaders/ShaderPrograms.h>
#include <EC/BSurfaces3.h>
#include <EC/ECSpaces.h>

namespace cagd // all data structures in the proposed function library are defined under the namespace cagd
{
    class YourGLWidget: public ...
    {
    private:
        // ...
        // these lines coincide with the lines 15/284–39/284 of Listing 3.8/284
        // ...

        // A triangle mesh that will store a triangulated unit sphere centered at the origin. Using translation and
        // scaling transformations, it will be rendered multiple times at the positions of the control points.
        TriangleMesh3                           _sphere;

        // Determines the common scaling factor of the unit sphere that has to be rendered at the positions
        // of the control points.
        GLdouble                                _control_point_radius;

        // Determines the scaling transformation of the unit sphere that has to be rendered at the positions
        // of the control points.
        Scale                                   _sphere_S;

        // Parameters of the EC spaces associated with directions u and v:
        std::vector<GLdouble>                   _alpha, _beta; // endpoints of definition domains;
        std::vector<GLint>                      _n;            // orders of the corresponding EC spaces;
        std::vector<SP<ECSpace>::Default>       _space;        // an array of smart pointers to EC spaces.

        // an array of numbers of uniform subdivision points in the corresponding definition domains,
        // they will be used for image generation
        std::vector<GLint>                      _div_point_count;

        // Determines the color scheme of the images (i.e., triangle meshes) of the B-surfaces that have to be
        // rendered. The user can choose from 13 possible color schemes such as DEFAULT_NULL_FRAGMENT,
        // X_VARIATION_FRAGMENT, Y_VARIATION_FRAGMENT, Z_VARIATION_FRAGMENT,
        // NORMAL_LENGTH_FRAGMENT, GAUSSIAN_CURVATURE_FRAGMENT,
        // MEAN_CURVATURE_FRAGMENT, WILLMORE_ENERGY_FRAGMENT,
        // LOG_WILLMORE_ENERGY_FRAGMENT, UMBILIC_DEVIATION_ENERGY_FRAGMENT,
        // LOG_UMBILIC_DEVIATION_ENERGY_FRAGMENT, TOTAL_CURVATURE_ENERGY_FRAGMENT,
        // LOG_TOTAL_CURVATURE_ENERGY_FRAGMENT. For more details see the lines 31/195–54/195 of
        // Listing 2.44/194 and Fig. 3.5/320. (With the exception of DEFAULT_NULL_FRAGMENT all remaining
        // color schemes should be used together with uniform black (color) materials.)
        TensorProductSurface3::ImageColorScheme                 _color_scheme;

        // smart pointer to a dynamically allocated randomly generated B-surface
        SP<BSurface3>::Default                                  _bsurface;

        // smart pointer to a dynamically allocated B-surface image (i.e., TriangleMesh3 object)
        SP<TriangleMesh3>::Default                              _img_bsurface;

        // smart pointer to dynamically allocated order elevated B-surface
        SP<BSurface3>::Default                                  _oe_bsurface;

        // smart pointer to the image of the dynamically allocated order elevated B-surface
        SP<TriangleMesh3>::Default                              _oe_img_bsurface;

        // an array of percentages that determine the subdivision points of the u- and v-directional definition
        // domains, where the initial B-surfaces have to be subdivided
        std::vector<GLdouble>                                   _ratio;
```


© Ágoston Róth, 2018

```
57          // an array of smart pointers to row matrices that store two smart pointers to the u- and v-directional
58          // subdivided B-surfaces
59          std::vector<SP< RowMatrix<SP<BSurface3>::Default> >::Default> _subdivision;

60          // an array of row matrices that store two smart pointers to the images of the u- and v-directional
61          // subdivided B-surfaces
62          std::vector< RowMatrix<SP<TriangleMesh3>::Default> >          _img_subdivision;

63          // decides whether the two-sided lighting or the reflection lines shader program should be used
64          // during rendering
65          bool _apply_reflection_lines;                          // synchronize it with a check box

66          // visibility flags that should be synchronized with radio buttons and check boxes of your graphical
67          // user interface
68          bool _show_randomly_generated_initial_B_surface; // synchronize it with a radio button
69          bool _show_order_elevated_B_surface;             // synchronize it with a radio button
70          bool _show_u_subdivided_B_surfaces;              // synchronize it with a radio button
71          bool _show_v_subdivided_B_surfaces;              // synchronize it with a radio button
72          bool _compare_control_nets;                      // synchronize it with a check box

73          // auxiliary private methods used for control point, control net and transparent triangle mesh rendering
74          void _renderSpheresAtControlPoints(
75              const BSurface3 &surface, const Color4 &front_color_material) const;

76          void _renderControlNet(
77              const BSurface3 &surface, const Color4 &color,
78              bool use_dashed_line = false) const;

79          void _renderTransparentMesh(
80              const ShaderProgram &shader, const TriangleMesh3 &mesh,
81              const Color4 &front_color_material, const Color4 &back_color_material,
82              GLfloat transparency = 0.5f) const;

83      public:
84          // your default/special constructor
85          YourGLWidget(/*...*/);

86          // your rendering method
87          void render();

88          // your other methods...
89      };
90  }

91  #endif // YOURGLWIDGET_H
```

Listing 3.17. Defining, rendering, order elevating and subdividing B-surfaces (**YourGLWidget.cpp**)

```
1   #pragma warning(disable:4503)
2   #include "YourGLWidget.h"

3   #include <iostream>
4   #include <fstream>

5   #include <Core/Exceptions.h>
6   #include <Core/Utilities.h>
7   #include <Core/Geometry/Surfaces/Materials.h>
8   #include <Core/Math/Constants.h>
9   #include "../Spaces/SpecializedECSpaces.h"

10  using namespace std;
11  using namespace cagd;
12  using namespace cagd::variable;

13  // your default/special constructor
14  void YourGLWidget::YourGLWidget(/*...*/)
15  {
16      try
17      {
18          // ...
19          // these lines coincide with the lines 15/285–144/287 of Listing 3.9/285
20          // ...
```


© Ágoston Róth, 2018

```
21          // Communicating with our shader programs via uniform variables...
22          _color_shader.enable();
23          if (!_color_shader.setUniformColor("color", colors::gray))
24          {
25              throw Exception("Color shader program: could not initialize the "
26                              "uniform variable \"color\"!");
27          }
28          _color_shader.disable();
29          _two_sided_lighting.enable();
30          if (!_two_sided_lighting.setUniformDirectionalLight(
31                  "light_source[0]", *_light))
32          {
33              throw Exception("Two-sided per pixed lighting: could not initialize the "
34                              "uniform variable \"light_source[0]\"!");
35          }
36          if (!_two_sided_lighting.setUniformValue1i(
37                  "light_source[0].enabled", GL_TRUE))
38          {
39              throw Exception("Two-sided per pixed lighting: could not initialize the "
40                              "uniform variable \"light_source[0].enabled\"!");
41          }
42          _two_sided_lighting.disable();
43          _reflection_lines.enable();
44          if (!_reflection_lines.setUniformDirectionalLight(
45                  "light_source[0]", *_light))
46          {
47              throw Exception("Reflection lines: could not initialize the "
48                              "uniform variable \"light_source[0]\"!");
49          }
50          if (!_reflection_lines.setUniformValue1i("light_source[0].enabled", GL_TRUE))
51          {
52              throw Exception("Reflection lines: could not initialize the "
53                              "uniform variable \"light_source[0].enabled\"!");
54          }
55          if (!_reflection_lines.setUniformValue1f("scale_factor", 9.7f))
56          {
57              throw Exception("Reflection lines: could not initialize the "
58                              "uniform variable \"scale_factor\"!");
59          }
60          if (!_reflection_lines.setUniformValue1f("smoothing", 1.0f))
61          {
62              throw Exception("Reflection lines: could not initialize the "
63                              "uniform variable \"smoothing\"!");
64          }
65          if (!_reflection_lines.setUniformValue1f("shading", 0.1f))
66          {
67              throw Exception("Reflection lines: could not initialize the "
68                              "uniform variable \"shading\"!");
69          }
70          _reflection_lines.disable();
71          // Loading a triangulated unit sphere centered at the origin. Using translation and scaling transformations,
72          // it will be rendered multiple times at the positions of the control points.
73          if (!_sphere.loadFromOFF("Models/sphere.off"))
74          {
75              throw Exception("Could not load model file!");
76          }
77          if (!_sphere.updateVertexBufferObjects())
78          {
```


© Ágoston Róth, 2018

```
79              throw Exception("Could not update the VBOs of the model!");
80          }
81          // Determines the common scaling factor of the unit sphere that has to be rendered at the positions
82          // of the control points.
83          _control_point_radius = 0.05;
84          // Determines the scaling transformation of the unit sphere that has to be rendered at the positions
85          // of the control points.
86          _sphere_S.setScalingFactors(
87                  _control_point_radius, _control_point_radius, _control_point_radius);
88          // ...
89          // these lines coincide with the lines 19/296–50/297 of Listing 3.13/296
90          // ...
91          // Memory allocations and parameter settings...
92          _alpha.resize(2);
93          _beta.resize(2);
94          _n.resize(2);
95          _ratio.resize(2);
96          _subdivision.resize(2);
97          _img_subdivision.resize(2);
98          _div_point_count.resize(2);
99          _alpha[U]           = 0.0;
100         _beta[U]            = 1.0;
101         _n[U]               = 4;
102         _ratio[U]           = 0.5;
103         _div_point_count[U] = 50;
104         _alpha[V]           = -PI / 3.0;
105         _beta[V]            =  PI / 3.0;
106         _n[V]               =  2;
107         _ratio[V]           =  0.5;
108         _div_point_count[V] =  70;
109         _space.resize(2);
110         // Try to create different EC spaces:
111         _space[U] = SP<ECSpace>::Default(
112                     new (nothrow) PolynomialECSpace(
113                         _alpha[U], _beta[U], _n[U],
114                         check_for_ill_conditioned_matrices,
115                         expected_correct_significant_digits));
116         if (!_space[U])
117         {
118             throw Exception("Could not create the EC space associated with the "
119                             "direction u!");
120         }
121         _space[V] = SP<ECSpace>::Default(
122                     new (nothrow) TrigonometricECSpace(
123                         _alpha[V], _beta[V], _n[V],
124                         check_for_ill_conditioned_matrices,
125                         expected_correct_significant_digits));
126         if (!_space[V])
127         {
128             throw Exception("Could not create the EC space associated with the "
129                             "direction v!");
130         }
131         // Try to create a B-surface:
132         _bsurface = SP<BSurface3>::Default(
133                 new (nothrow) BSurface3(*_space[U], *_space[V]));
134         if (!_bsurface)
135         {
136             throw Exception("Could not create the B-surface!");
137         }
138         // Generate randomly the control points of the B-surface:
```


© Ágoston Róth, 2018

```
139         GLdouble u_min = -3.0, u_max = 3.0,
140                  u_step = (u_max - u_min) / (_bsurface->rowCount() - 1);
141         GLdouble v_min = -2.0, v_max = 2.0,
142                  v_step = (v_max - v_min) / (_bsurface->columnCount() - 1);
143         GLdouble z_min = -3.5, z_max = 3.5, length = z_max - z_min;

144         for (int r = 0; r < _bsurface->rowCount(); r++)
145         {
146             for (int c = 0; c < _bsurface->columnCount(); c++)
147             {
148                 Cartesian3 &reference = (*_bsurface)(r, c);

149                 reference[0] = min(u_min + r * u_step, u_max);
150                 reference[1] = min(v_min + c * v_step, v_max);
151                 reference[2] = z_min + length * (GLdouble)rand()/(GLdouble)RAND_MAX;
152             }
153         }

154         // Try to update the VBO of the initial control net:
155         if (!_bsurface->updateVertexBufferObjectsOfData())
156         {
157             throw Exception("Could not update the VBO of the initial control net!");
158         }

159         // Select a color scheme:
160         _color_scheme = TensorProductSurface3::DEFAULT_NULL_FRAGMENT;

161         // Try to generate the image of the initial B-surface:
162         _img_bsurface = SP<TriangleMesh3>::Default(
163                     _bsurface->generateImage(
164                         _div_point_count[U], _div_point_count[V],
165                         _color_scheme));

166         if (!_img_bsurface)
167         {
168             throw Exception("Could not generate the image of the initial B-surface!");
169         }

170         // Try to update the VBOs of initial B-surface's image:
171         if (!_img_bsurface->updateVertexBufferObjects())
172         {
173             throw Exception("Could not update the VBOs of the initial B-surface's "
174                             "image!");
175         }

176         // Try to perform order elevation in direction u:
177         CharacteristicPolynomial::Zero u_zero(0.0, 0.0, 9);

178         _oe_bsurface = SP<BSurface3>::Default(
179                     _bsurface->performOrderElevation(
180                         U,  u_zero,
181                         check_for_ill_conditioned_matrices,
182                         expected_correct_significant_digits));

183         if (!_oe_bsurface)
184         {
185             throw Exception("Could not perform order elevation in direction u!");
186         }

187         // Try to perform order elevation in direction v:
188         CharacteristicPolynomial::Zero v_zero(0.0, 1.0, 3);

189         _oe_bsurface = SP<BSurface3>::Default(
190                     _oe_bsurface->performOrderElevation(
191                         V,  v_zero,
192                         check_for_ill_conditioned_matrices,
193                         expected_correct_significant_digits));

194         if (!_oe_bsurface)
195         {
196             throw Exception("Could not perform order elevation in direction v!");
197         }

198         // Remark
```


© Ágoston Róth, 2018

```
199         // After successful u- and v-directional order elevations, the obtained surface is described by means
200         // of the unique normalized B-bases of the EC spaces P_8^{0,1} = span{u^i : u ∈ [0,1]}_{i=0}^8 and
201         // AT_8^{-π/3,π/3} = span{1, cos(v), sin(v), v cos(v), v sin(v), v^2 cos(v), v^2 sin(v), cos(2v), sin(2v) : v ∈ [-π/3, π/3]}.

202         // Try to update the VBO of the control net of the order elevated B-surface:
203         if (!_oe_bsurface->updateVertexBufferObjectsOfData())
204         {
205             throw Exception("Could not update the VBOs of the control net of the "
206                             "order elevated B-surface!");
207         }

208         // Try to generate the image of the order elevated B-surface:
209         _oe_img_bsurface = SP<TriangleMesh3>::Default(
210                     _oe_bsurface->generateImage(
211                         _div_point_count[U], _div_point_count[V],
212                         _color_scheme));

213         if (!_oe_img_bsurface)
214         {
215             throw Exception("Could not generate the image of the order elevated "
216                             "B-surface!");
217         }

218         // Try to update the VBOs of the order elevated B-surface's image:
219         if (!_oe_img_bsurface->updateVertexBufferObjects())
220         {
221             throw Exception("Could not update the VBOs of the order elevated "
222                             "B-surface's image!");
223         }

224         // Try to perform subdivision on the initial B-surface in both directions:
225         for (GLint direction = U; direction <= V; direction++)
226         {
227             GLdouble gamma = (1.0 - _ratio[direction]) * _alpha[direction] +
228                              _ratio[direction]  * _beta[direction];

229             _subdivision[direction] = SP< RowMatrix<SP<BSurface3>::Default> >::Default(
230                     _bsurface->performSubdivision(
231                             (variable::Type)direction, gamma,
232                             check_for_ill_conditioned_matrices,
233                             expected_correct_significant_digits));

234             if (!_subdivision[direction])
235             {
236                 throw Exception("Could not perform subdivision in direction " +
237                                 string(direction ? "v" : "u") + ".!");
238             }

239             _img_subdivision[direction].resizeColumns(2);

240             for (GLint part = 0; part < 2; part++)
241             {
242                 if (!(*_subdivision[direction])[part]->updateVertexBufferObjectsOfData())
243                 {
244                     throw Exception("Could not update the VBOs of the control net of "
245                                     "the " + string(part ? "right" : "left") +
246                                     " part of the " + string(direction ? "v" : "u") +
247                                     "-directional subdivisions!");
248                 }

249                 _img_subdivision[direction][part] = SP<TriangleMesh3>::Default(
250                         (*_subdivision[direction])[part]->generateImage(
251                                 _div_point_count[U], _div_point_count[V],
252                                 _color_scheme));

253                 if (!_img_subdivision[direction][part])
254                 {
255                     throw Exception("Could not generate the image of the " +
256                                     string(part ? "right" : "left") + " part of the " +
257                                     string(direction ? "v" : "u") +
258                                     "-directional subdivisions!");
259                 }
```


© Ágoston Róth, 2018

```
260                 if (!_img_subdivision[direction][part]->updateVertexBufferObjects())
261                 {
262                     throw Exception("Could not update the VBOs of the image of the " +
263                                     string(part ? "right" : "left") + " part of the " +
264                                     string(direction ? "v" : "u") +
265                                     "-directional subdivisions!");
266                 }
267             }
268         }

269         // decides whether the two-sided lighting or the reflection lines shader program should be used
270         // during rendering
271         _apply_reflection_lines                         = false;

272         // visibility flags that should be synchronized with radio buttons and check boxes of your graphical
273         // user interface
274         _show_randomly_generated_initial_B_surface = true;  // synchronize it with a radio button
275         _show_order_elevated_B_surface             = false; // synchronize it with a radio button
276         _show_u_subdivided_B_surfaces              = false; // synchronize it with a radio button
277         _show_v_subdivided_B_surfaces              = false; // synchronize it with a radio button
278         _compare_control_nets                      = false; // synchronize it with a check box

279         glClearColor(1.0, 1.0, 1.0, 1.0);          // set the background color
280         glEnable(GL_DEPTH_TEST);                   // enable depth testing
281         glEnable(GL_LINE_SMOOTH);                  // enable the anti-aliasing of line primitives
282         glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);
283         glEnable(GL_POINT_SMOOTH);                 // enable the anti-aliasing of point primitives
284         glHint(GL_POINT_SMOOTH_HINT, GL_NICEST);

285     }
286     catch (Exception &e)
287     {
288         cout << e << endl;
289     }
290 }

291 GLboolean GLWidget::_updateTransformationMatrices()
292 {
293     // ...
294     // these lines coincide with the lines 224/288–233/289 of Listing 3.9/285
295     // ...
296 }

297 // your rendering method
298 void YourGLWidget::render()
299 {
300     // clear the color and depth buffers
301     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

302     // revert to the original transformation matrices in case of the color shader program
303     _color_shader.enable();
304         _color_shader.setUniformMatrix4fv("PVM", 1, GL_FALSE, _PVM.address());
305     _color_shader.disable();

306     // At first, we render non-transparent geometries like control nets and spheres that represent control points:

307     if (_show_order_elevated_B_surface && _oe_bsurface)
308     {
309         // render small spheres at the positions of the control points of the order elevated B-surface...
310         _renderSpheresAtControlPoints(*_oe_bsurface,
311                                       colors::light_purple, colors::silver);

312         // render the control net of the order elevated B-surface...
313         _renderControlNet(*_oe_bsurface, colors::gray);
314     }

315     // auxiliary arrays that store boolean values or pointers to existing color objects...
316     // these values will be used in the next for-loop...
317     bool show_subdivision[2] = {_show_u_subdivided_B_surfaces,
318                                 _show_v_subdivided_B_surfaces};

319     const Color4 * const control_point_color[2] = {&colors::ice_blue, &colors::purple};

320     for (GLint direction = U; direction <= V; direction++)
```


© Ágoston Róth, 2018

```
321     {
322         if (show_subdivision[direction] && _subdivision[direction])
323         {
324             for (GLint part = 0; part < 2; part++)
325             {
326                 // render small spheres at the positions of the control points of the u- and v-directional
327                 // subdivisions...
328                 _renderSpheresAtControlPoints(
329                         *(*_subdivision[direction])[part],
330                         *control_point_color[part], colors::silver);
331 
332                 // render the control nets of the u- and v-directional subdivisions...
333                 _renderControlNet(*(*_subdivision[direction])[part], colors::gray);
334             }
335         }
336     }
337 
338     if ((_show_randomly_generated_initial_B_surface || _compare_control_nets) &&
339         _bsurface)
340     {
341         // render small spheres at the positions of the control points of the initial B-surface...
342         _renderSpheresAtControlPoints(*_bsurface, colors::red, colors::silver);
343 
344         // render the control net of the initial B-surface... based on the true or false values of the variable
345         // _compare_control_nets we will use either dashed or continuous line styles, respectively...
346         _renderControlNet(*_bsurface, colors::gray, _compare_control_nets);
347     }
348 
349     // Second, we render transparent geometries like the triangle mesh images of all B-surfaces:
350 
351     // select the constant reference of either the two-sided lighting or the reflection line generating shader program...
352     const ShaderProgram &surface_shader = _apply_reflection_lines ?
353                                           _reflection_lines : _two_sided_lighting;
354 
355     // revert to the original transformation matrices...
356     surface_shader.enable();
357 
358         surface_shader.setUniformMatrix4fv("VM",   1, GL_FALSE, _VM.address());
359         surface_shader.setUniformMatrix4fv("PVM",  1, GL_FALSE, _PVM.address());
360         surface_shader.setUniformMatrix4fv("N",    1, GL_TRUE,  _tN.address());
361 
362     surface_shader.disable();
363 
364     // in order to obtain the best coloring effects, color schemes different than the DEFAULT_NULL_FRAGMENT
365     // should be used together with black uniform (color) materials...
366     bool default_color_scheme =
367             (_color_scheme == TensorProductSurface3::DEFAULT_NULL_FRAGMENT);
368 
369     if (_show_randomly_generated_initial_B_surface && _img_bsurface)
370     {
371         // render the triangle mesh of the initial B-surface...
372         _renderTransparentMesh(
373                 surface_shader, *_img_bsurface,
374                 default_color_scheme ? colors::light_blue : colors::black,
375                 default_color_scheme ? colors::silver : colors::black);
376     }
377 
378     if (_show_order_elevated_B_surface && _oe_bsurface && _oe_img_bsurface)
379     {
380         // render the triangle mesh of the order elevated B-surface...
381         _renderTransparentMesh(
382                 surface_shader, *_oe_img_bsurface,
383                 default_color_scheme ? colors::light_blue : colors::black,
384                 default_color_scheme ? colors::silver : colors::black);
385     }
386 
387     // an auxiliary array of pointers to existing color objects that will be used in the next foor-loop...
388     const Color4 * const surface_color[2] = {&colors::light_blue, &colors::light_purple};
389 
390     for (GLint direction = U; direction <= V; direction++)
391     {
392         if (show_subdivision[direction])
393         {
394             for (GLint part = 0; part < 2; part++)
```


© Ágoston Róth, 2018

```
382             {
383                 // render the triangle meshes of the left and right B-patches obtained during u- and v-directional
384                 // subdivisions...
385                 if (_img_subdivision[direction][part])
386                 {
387                     _renderTransparentMesh(
388                             surface_shader, *_img_subdivision[direction][part],
389                             default_color_scheme ? *surface_color[part] : colors::black,
390                             default_color_scheme ? colors::silver : colors::black);
391                 }
392             }
393         }
394     }
395     // render other geometries...
396 }
397 // auxiliary private method used for control point rendering
398 void YourGLWidget::_renderSpheresAtControlPoints(
399         const BSurface3 &surface,
400         const Color4 &front_color_material, const Color4 &back_color_material) const
401 {
402     _two_sided_lighting.enable();
403         _two_sided_lighting.setUniformColorMaterial(
404                 "front_material", front_color_material);
405         _two_sided_lighting.setUniformColorMaterial(
406                 "back_material", back_color_material);
407         for (int r = 0; r < surface.rowCount(); r++)
408         {
409             for (int c = 0; c < surface.columnCount(); c++)
410             {
411                 const Cartesian3 &reference = surface(r, c);
412                 Translate sphere_T(reference[0], reference[1], reference[2]);
413                 GLTransformation sphere_VM  = _VM * sphere_T * _sphere_S;
414                 GLTransformation sphere_PVM = (*_P) * sphere_VM;
415                 _two_sided_lighting.setUniformMatrix4fv(
416                         "VM", 1, GL_FALSE, sphere_VM.address());
417                 _two_sided_lighting.setUniformMatrix4fv(
418                         "PVM", 1, GL_FALSE, sphere_PVM.address());
419                 _two_sided_lighting.setUniformMatrix4fv(
420                         "N", 1, GL_TRUE, sphere_VM.inverse().address());
421                 _sphere.render(_two_sided_lighting);
422             }
423         }
424     _two_sided_lighting.disable();
425 }
426 // auxiliary private method used for control net rendering
427 void YourGLWidget::_renderControlNet(
428         const BSurface3 &surface, const Color4 &color, bool use_dashed_line) const
429 {
430     glLineWidth(2.0);
431     _color_shader.enable();
432         _color_shader.setUniformColor("color", color);
433         if (use_dashed_line)
434         {
435             glEnable(GL_LINE_STIPPLE);
436                 glLineStipple(1, 0xf0f0);
437                 surface.renderData(_color_shader);
438             glDisable(GL_LINE_STIPPLE);
439         }
440         else
441         {
442             surface.renderData(_color_shader);
```


© Ágoston Róth, 2018

```
443         }

444     _color_shader.disable();
445     glLineWidth(1.0);
446 }

447 // auxiliary private method used for transparent triangle mesh rendering
448 void YourGLWidget::_renderTransparentMesh(
449         const ShaderProgram &shader, const TriangleMesh3 &mesh,
450         const Color4 &front_color_material, const Color4 &back_color_material,
451         GLfloat transparency) const
452 {
453     shader.enable();

454         if (!shader.setUniformColorMaterial("front_material", front_color_material))
455         {
456             throw Exception("Currently active shader program: could not initialize the "
457                             "uniform variable \"front_material\"!");
458         }

459         if (!shader.setUniformColorMaterial("back_material", back_color_material))
460         {
461             throw Exception("Currently active shader program: could not initialize the "
462                             "uniform variable \"back_material\"!");
463         }

464         if (!shader.setUniformValue1f("transparency", 0.5f))
465         {
466             throw Exception("Currently active shader program: could not initialize the "
467                             "uniform variable \"transparency\"!");
468         }

469         if (transparency)
470         {
471             // in order to ensure transparency on white background, we render the geometry in two steps:
472             glEnable(GL_BLEND);

473                 glBlendFunc(GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA);
474                 glDepthMask(GL_FALSE);
475                 glEnable(GL_CULL_FACE);

476                     // at first we cull the front faces and render the back ones,
477                     glCullFace(GL_FRONT);

478                     if (!mesh.render(shader))
479                     {
480                         throw Exception("YourGLWidget::_renderTransparentMesh : "
481                                         "could not render the given triangle mesh!");
482                     }

483                     // then we cull the back faces and render the front ones
484                     glCullFace(GL_BACK);
485                     mesh.render(shader);

486                 glDisable(GL_CULL_FACE);
487                 glDepthMask(GL_TRUE);

488             glDisable(GL_BLEND);
489         }
490         else
491         {
492             mesh.render(shader);
493         }

494     shader.disable();
495 }

496 // The following comments specify the parameter settings that have to be applied in the constructor of the class
497 // YourGLWidget in order to obtain the images presented in Figs. 3.4/319(a)–(f), 3.5/320(a)–(ℓ), 3.6/321(a)–(d) and
498 // 3.7/322(a)–(d) of the current subsection. One has to modify the lines 160/311 and 271/313–278/313 of the constructor.

499 // Figs. 3.4/319(a), 3.6/321(a) and 3.7/322(a) are the same and illustrate the randomly generated initial B-surface.
500 // The image was obtained by using the parameter settings:
501 //         _color_scheme = TensorProductSurface3::DEFAULT_NULL_FRAGMENT;
```


© Ágoston Róth, 2018

```
502 //          _apply_reflection_lines = false;
503 //          _show_randomly_generated_initial_B_surface = true;
504 //          _show_order_elevated_B_surface = false;
505 //          _show_u_subdivided_B_surfaces = false;
506 //          _show_v_subdivided_B_surfaces = false;
507 //          _compare_control_nets = false;

508 // Figs. 3.4/319(b)–(f) and 3.5/320(a)–(ℓ) illustrate the same order elevated B-surface.

509 // Fig. 3.4/319(b) was obtained by using the parameter settings:
510 //          _color_scheme = TensorProductSurface3::DEFAULT_NULL_FRAGMENT;
511 //          _apply_reflection_lines = false;
512 //          _show_randomly_generated_initial_B_surface = false;
513 //          _show_order_elevated_B_surface = true;
514 //          _show_u_subdivided_B_surfaces = false;
515 //          _show_v_subdivided_B_surfaces = false;
516 //          _compare_control_nets = true;

517 // Fig. 3.4/319(c) was obtained by using the parameter settings:
518 //          _color_scheme = TensorProductSurface3::DEFAULT_NULL_FRAGMENT;
519 //          _apply_reflection_lines = false;
520 //          _show_randomly_generated_initial_B_surface = false;
521 //          _show_order_elevated_B_surface = true;
522 //          _show_u_subdivided_B_surfaces = false;
523 //          _show_v_subdivided_B_surfaces = false;
524 //          _compare_control_nets = false;

525 // Fig. 3.4/319(d) was obtained by using the parameter settings:
526 //          _color_scheme = TensorProductSurface3::DEFAULT_NULL_FRAGMENT;
527 //          _apply_reflection_lines = true;
528 //          _show_randomly_generated_initial_B_surface = false;
529 //          _show_order_elevated_B_surface = true;
530 //          _show_u_subdivided_B_surfaces = false;
531 //          _show_v_subdivided_B_surfaces = false;
532 //          _compare_control_nets = false;

533 // Fig. 3.4/319(e) was obtained by using the parameter settings:
534 //          _color_scheme = TensorProductSurface3::LOG_UMBILIC_DEVIATION_ENERGY_FRAGMENT;
535 //          _apply_reflection_lines = false;
536 //          _show_randomly_generated_initial_B_surface = false;
537 //          _show_order_elevated_B_surface = true;
538 //          _show_u_subdivided_B_surfaces = false;
539 //          _show_v_subdivided_B_surfaces = false;
540 //          _compare_control_nets = false;

541 // Fig. 3.4/319(f) was obtained by using the parameter settings:
542 //          _color_scheme = TensorProductSurface3::LOG_UMBILIC_DEVIATION_ENERGY_FRAGMENT;
543 //          _apply_reflection_lines = true;
544 //          _show_randomly_generated_initial_B_surface = false;
545 //          _show_order_elevated_B_surface = true;
546 //          _show_u_subdivided_B_surfaces = false;
547 //          _show_v_subdivided_B_surfaces = false;
548 //          _compare_control_nets = false;

549 // Figs. 3.5/320(a)–(ℓ) were obtained by using the common parameter settings:
550 //          _apply_reflection_lines = false;
551 //          _show_randomly_generated_initial_B_surface = false;
552 //          _show_order_elevated_B_surface = true;
553 //          _show_u_subdivided_B_surfaces = false;
554 //          _show_v_subdivided_B_surfaces = false;
555 //          _compare_control_nets = false;
556 // and by applying the color schemes:
557 //          _color_scheme = TensorProductSurface3::X_VARIATION_FRAGMENT;
558 //          _color_scheme = TensorProductSurface3::Y_VARIATION_FRAGMENT;
559 //          _color_scheme = TensorProductSurface3::Z_VARIATION_FRAGMENT;
560 //          _color_scheme = TensorProductSurface3::NORMAL_LENGTH_FRAGMENT;
561 //          _color_scheme = TensorProductSurface3::GAUSSIAN_CURVATURE_FRAGMENT;
562 //          _color_scheme = TensorProductSurface3::MEAN_CURVATURE_FRAGMENT;
563 //          _color_scheme = TensorProductSurface3::WILLMORE_ENERGY_FRAGMENT;
564 //          _color_scheme = TensorProductSurface3::LOG_WILLMORE_ENERGY_FRAGMENT;
565 //          _color_scheme = TensorProductSurface3::UMBILIC_DEVIATION_ENERGY_FRAGMENT;
566 //          _color_scheme = TensorProductSurface3::TOTAL_CURVATURE_ENERGY_FRAGMENT;
567 //          _color_scheme = TensorProductSurface3::LOG_TOTAL_CURVATURE_ENERGY_FRAGMENT;
568 // respectively.
```


© Ágoston Róth, 2018

```
// Fig. 3.6/321(b) was obtained by using the parameter settings:
//          color_scheme = TensorProductSurface3::DEFAULT_NULL_FRAGMENT;
//          apply_reflection_lines = false;
//          show_randomly_generated_initial_B_surface = false;
//          show_order_elevated_B_surface = false;
//          show_u_subdivided_B_surfaces = true;
//          show_v_subdivided_B_surfaces = false;
//          compare_control_nets = true;

// Fig. 3.6/321(c) was obtained by using the parameter settings:
//          color_scheme = TensorProductSurface3::DEFAULT_NULL_FRAGMENT;
//          apply_reflection_lines = false;
//          show_randomly_generated_initial_B_surface = false;
//          show_order_elevated_B_surface = false;
//          show_u_subdivided_B_surfaces = true;
//          show_v_subdivided_B_surfaces = false;
//          compare_control_nets = false;

// Fig. 3.6/321(d) was obtained by using the parameter settings:
//          color_scheme = TensorProductSurface3::DEFAULT_NULL_FRAGMENT;
//          apply_reflection_lines = true;
//          show_randomly_generated_initial_B_surface = false;
//          show_order_elevated_B_surface = false;
//          show_u_subdivided_B_surfaces = true;
//          show_v_subdivided_B_surfaces = false;
//          compare_control_nets = false;

// Fig. 3.7/322(b) was obtained by using the parameter settings:
//          color_scheme = TensorProductSurface3::DEFAULT_NULL_FRAGMENT;
//          apply_reflection_lines = false;
//          show_randomly_generated_initial_B_surface = false;
//          show_order_elevated_B_surface = false;
//          show_u_subdivided_B_surfaces = false;
//          show_v_subdivided_B_surfaces = true;
//          compare_control_nets = true;

// Fig. 3.7/322(c) was obtained by using the parameter settings:
//          color_scheme = TensorProductSurface3::DEFAULT_NULL_FRAGMENT;
//          apply_reflection_lines = false;
//          show_randomly_generated_initial_B_surface = false;
//          show_order_elevated_B_surface = false;
//          show_u_subdivided_B_surfaces = false;
//          show_v_subdivided_B_surfaces = true;
//          compare_control_nets = false;

// Fig. 3.7/322(d) was obtained by using the parameter settings:
//          color_scheme = TensorProductSurface3::DEFAULT_NULL_FRAGMENT;
//          apply_reflection_lines = true;
//          show_randomly_generated_initial_B_surface = false;
//          show_order_elevated_B_surface = false;
//          show_u_subdivided_B_surfaces = false;
//          show_v_subdivided_B_surfaces = true;
//          compare_control_nets = false;
```

3.5.2 B-representation of ordinary integral surfaces

B-surfaces can also be used to provide control point configurations for the exact description (or B-representation) of ordinary integral surfaces of type (1.50/12) as it is illustrated in Listings 3.18/321 and 3.19/324. We also show how can one evaluate and render the zeroth and higher order partial derivatives of the isoparametric lines of arbitrary B-surfaces. As we will see, the mentioned listings at first define the mixed exponential-trigonometric and pure trigonometric EC spaces

$$\mathbb{ET}_6^{\alpha_0^k,\beta_0^k} = \langle\{\varphi_{6,0}(u)\equiv 1, \varphi_{6,1}(u)=\cos(u), \varphi_{6,2}(u)=\sin(u), \varphi_{6,3}(u)=e^{\omega_0 u}, \varphi_{6,4}(u)=e^{\omega_1 u},$$
$$\varphi_{6,5}(u)=e^{\omega_0 u}\cos(u), \varphi_{6,6}(u)=e^{\omega_0 u}\sin(u) : u\in\left[\alpha_0^k,\beta_0^k\right]\}\rangle,\ k=0,1,\ldots,4$$

and

$$\mathbb{T}_2^{\alpha_1^\ell,\beta_1^\ell} = \left\langle\left\{\varphi_{2,0}(v)\equiv 1, \varphi_{2,1}(v)=\cos(v), \varphi_{2,2}(v)=\sin(v) : v\in\left[\alpha_1^\ell,\beta_1^\ell\right]\right\}\right\rangle,\ \ell=0,1,\ldots,\lfloor 3\tau\rfloor,$$

© Ágoston Róth, 2018

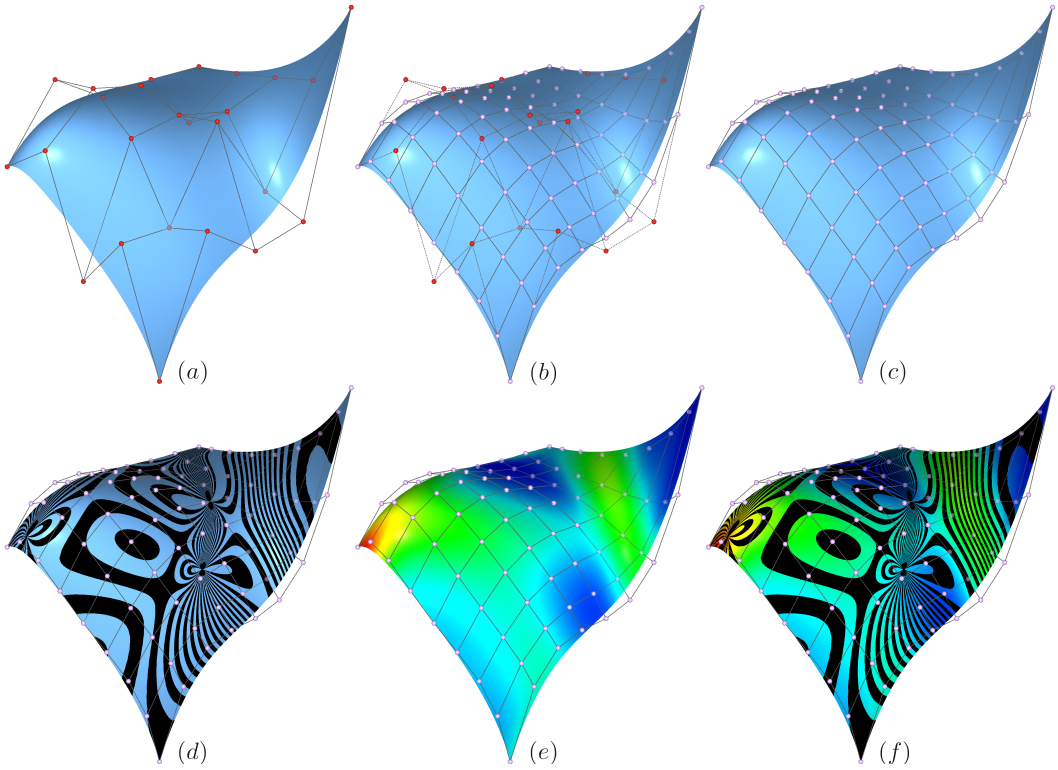


Fig. 3.4: (a) Randomly generated initial B-surface, rendered with a simple color material together with its control net. (b) Order elevation of the initial B-surface, rendered with a simple color material together with its initial and order elevated control nets. (c) The order elevated B-surface, rendered with a simple color material together with its control net. (d) The order elevated B-surface, rendered with a simple color material together with its control net and reflection lines. (d) The order elevated B-surface, rendered with the logarithmic umbilic deviation color scheme together with its control net. (e) The order elevated B-surface, rendered with the translated logarithmic scale of the umbilic deviation color scheme together with its control net and reflection lines. (The figure was generated by the source codes given in Listings 3.16/307 and 3.17/308.)

respectively, where

$$
\begin{aligned}
\omega_0 &= \frac{1}{6\pi}, \\
\omega_1 &= \frac{1}{3\pi}, \\
\alpha_0^k &= \frac{7\pi}{2} + \frac{29\pi}{40} \cdot k, \;\; k = 0, 1, \ldots, 4, \\
\beta_0^k &= \alpha_0^k + \frac{29\pi}{40}, \;\; k = 0, 1, \ldots, 4, \\
\alpha_1^\ell &= -\frac{\pi}{3\tau} + \frac{2\pi}{\lfloor 3\tau \rfloor} \cdot \ell, \;\; \ell = 0, 1, \ldots, \lfloor 3\tau \rfloor, \;\; \tau \geq 1, \\
\beta_1^\ell &= \alpha_1^\ell + \frac{2\pi}{\lfloor 3\tau \rfloor}, \;\; \ell = 0, 1, \ldots, \lfloor 3\tau \rfloor,
\end{aligned}
$$

© Ágoston Róth, 2018

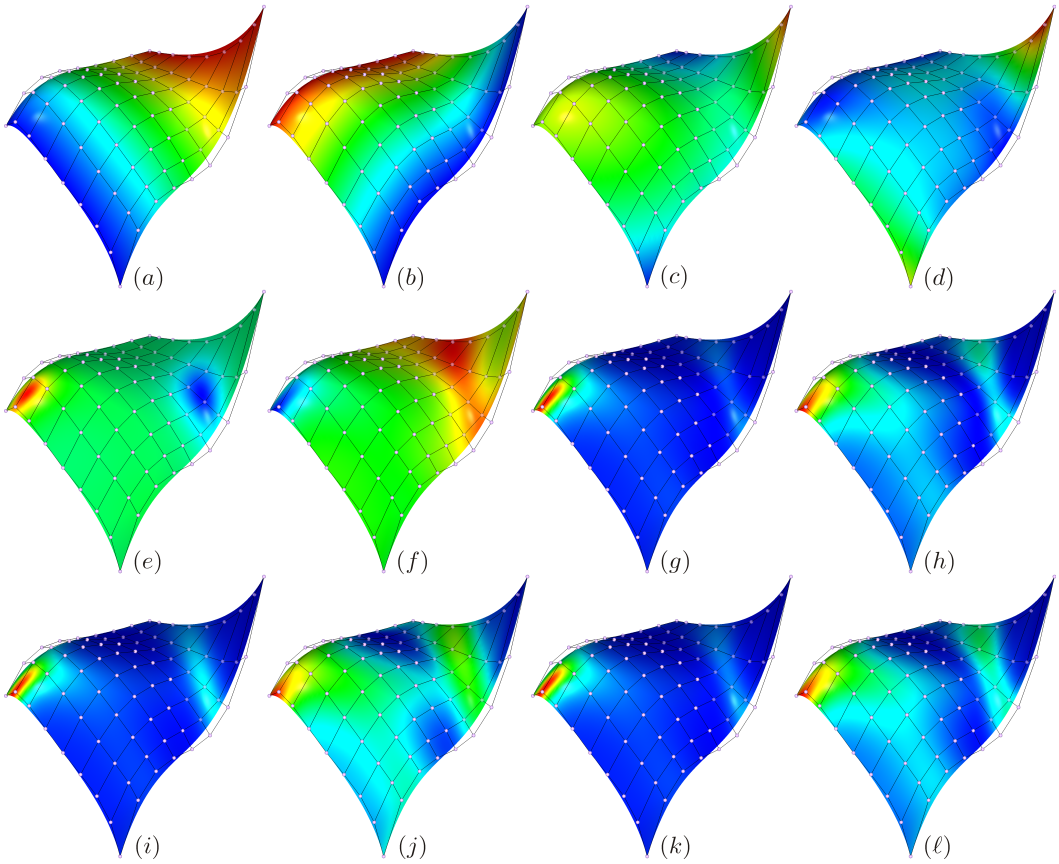


Fig. 3.5: The figure was generated by the source codes given in Listings 3.16/307 and 3.17/308. (a)–(ℓ) The color schemes correspond to the point-wise variations of the x-, y- and z-coordinates, of the length of the normal vectors, of the Gaussian- and mean curvatures, of the Willmore energy and its translated logarithmic counterpart, of the umbilic deviation and its translated logarithmic scale, of the total curvature and its translated logarithmic variant, respectively. (In each case the applied color map behaves like a temperature variation that ranges from the cold dark blue to the hot red, by passing through the colors cyan, green, yellow and orange such that the minimal and maximal values of a fixed energy type correspond to the extremal colors dark blue and red, respectively. For more details, see the lines 10/127–50/128 and 31/195–54/195 of Listings 2.34/127 and 2.44/194, respectively.)

then they also generate B-surfaces for the B-representations of the ordinary surface patches

$$\mathbf{s}_{k,\ell}\,(u,v) = \begin{bmatrix} (1-e^{\omega_0 u})\cos(u)\left(\frac{5}{4}+\cos(v)\right) \\ (e^{\omega_0 u}-1)\sin(u)\left(\frac{5}{4}+\cos(v)\right) \\ 7-e^{\omega_1 u}-\sin(v)+e^{\omega_0 u}\sin(v) \end{bmatrix}, \quad (u,v)\in\left[\alpha_0^k,\beta_0^k\right]\times\left[\alpha_1^\ell,\beta_1^\ell\right], \qquad (3.9)$$

where $k = 0, 1, \ldots, 4$ and $\ell = 0, 1, \ldots, \lfloor 3\tau \rfloor$.

© Ágoston Róth, 2018

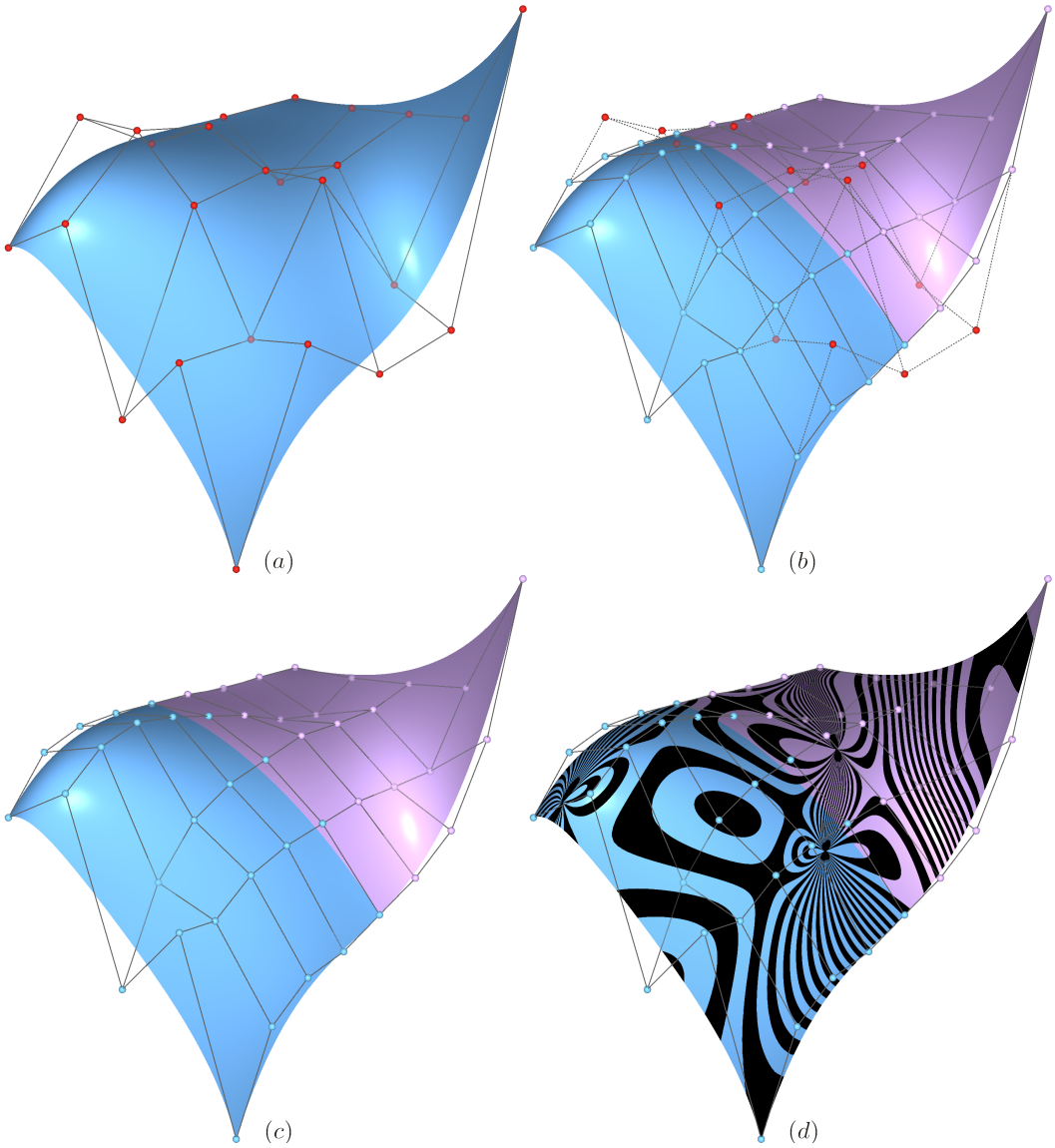


Fig. 3.6: (*a*) A randomly generated initial B-surface, rendered with a simple color material together with its control net. (*b*) Subdivision of the given B-surface in direction u. The obtained B-patches are rendered with simple color materials together with the control net of the initial B-surface and with their control nets. (*c*) The obtained B-patches are rendered with simple color materials together with their control nets. (*d*) The obtained B-patches are rendered with simple color materials together with their control nets and smooth reflection lines. (The figure was generated by the source codes given in Listings 3.16/307 and 3.17/308.)

Listing 3.18. B-representations of ordinary integral surfaces and their isoparametric lines (**YourGLWidget.h**)

```
1 #ifndef YOURGLWIDGET_H
2 #define YOURGLWIDGET_H
```


© Ágoston Róth, 2018

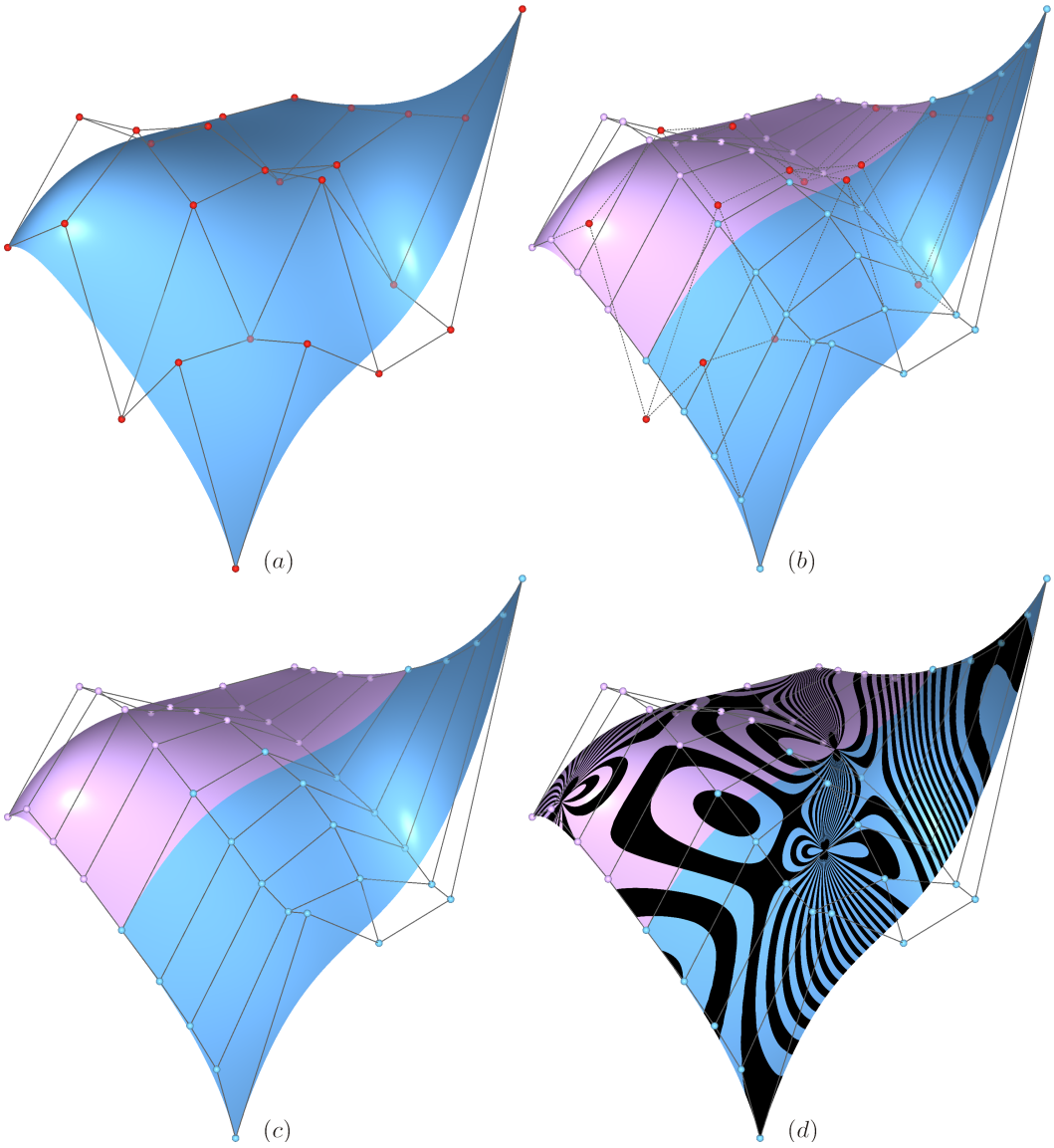


Fig. 3.7: (*a*) A randomly generated initial B-surface, rendered with a simple color material together with its control net. (*b*) Subdivision of the given B-surface in direction v. The obtained B-patches are rendered with simple color materials together with the control net of the initial B-surface and with their control nets. (*c*) The obtained B-patches are rendered with simple color materials together with their control nets. (*d*) The obtained B-patches are rendered with simple color materials together with their control nets and smooth reflection lines. (The figure was generated by the source codes given in Listings 3.16/307 and 3.17/308.)

```
3 #include <GL/glew.h>

4 #include <Core/SmartPointers/SpecializedSmartPointers.h>
5 #include <Core/Geometry/Surfaces/Lights.h>
6 #include <Core/Geometry/Surfaces/TriangleMeshes3.h>
7 #include <Core/Math/SpecialGLTransformations.h>
```


© Ágoston Róth, 2018

```
8  #include <Core/Shaders/ShaderPrograms.h>
9  #include <EC/BSurfaces3.h>
10 #include <EC/ECSpaces.h>

11 namespace cagd // all data structures in the proposed function library are defined under the namespace cagd
12 {
13     class YourGLWidget: public ...
14     {
15     private:
16         // ...
17         // these lines coincide with the lines 15/284–39/284 of Listing 3.8/284
18         // ...

19         // A triangle mesh that will store a triangulated unit sphere centered at the origin. Using translation and
20         // scaling transformations, it will be rendered multiple times at the positions of the control points.
21         TriangleMesh3                          _sphere;

22         // Determines the common scaling factor of the unit sphere that has to be rendered at the positions
23         // of the control points.
24         GLdouble                               _control_point_radius;

25         // Determines the scaling transformation of the unit sphere that has to be rendered at the positions
26         // of the control points.
27         Scale                                  _sphere_S;

28         // A triangle mesh that will store a triangulated right circular cone with unit base radius and appex (0, 0, 2).
29         // Using translation and scaling transformation, this cone will be rendered multiple times at the endpoints
30         // of the tangent vectors of the isoparametric lines.
31         TriangleMesh3                          _cone;

32         // Determines the scaling transformation of the cone that has to be rendered at the endpoints of the
33         // tangent vectors of the isoparametric lines.
34         Scale                                  _cone_S;

35         // Parameters of the EC spaces associated with directions u and v:
36         std::vector<GLdouble>                  _alpha, _beta; // endpoints of definition domains;
37         std::vector<GLint>                     _n;            // orders of the corresponding EC spaces;
38         std::vector<SP<ECSpace>::Default>      _space;        // an array of smart pointers to EC spaces.

39         // a 2-element array that stores the dimensions of the EC spaces applied in direction u and v
40         std::vector<GLint>                     _dimension;

41         // a 2-element array of numbers of uniform subdivision points in the corresponding definition domains,
42         // they will be used for surface image (i.e., TriangleMesh3) generation
43         std::vector<GLint>                     _surface_div_point_count;

44         // a 2-element array of isoparametric line counts in the corresponding directions
45         std::vector<GLint>                     _isoparametric_line_count;

46         // a 2-element array of maximum differentiation orders in the corresponding directions
47         std::vector<GLint>                     _maximum_order_of_derivatives;

48         // a 2-element array of numbers of uniform subdivision points in the corresponding definition domains,
49         // they will be used for curve image (i.e., GenericCurve3) generation
50         std::vector<GLint>                     _curve_div_point_count;

51         // Determines the color scheme of the images (i.e., triangle meshes) of the B-surfaces that have to be
52         // rendered. The user can choose from 13 possible color schemes such as DEFAULT_NULL_FRAGMENT,
53         // X_VARIATION_FRAGMENT, Y_VARIATION_FRAGMENT, Z_VARIATION_FRAGMENT,
54         // NORMAL_LENGTH_FRAGMENT, GAUSSIAN_CURVATURE_FRAGMENT,
55         // MEAN_CURVATURE_FRAGMENT, WILLMORE_ENERGY_FRAGMENT,
56         // LOG_WILLMORE_ENERGY_FRAGMENT, UMBILIC_DEVIATION_ENERGY_FRAGMENT,
57         // LOG_UMBILIC_DEVIATION_ENERGY_FRAGMENT, TOTAL_CURVATURE_ENERGY_FRAGMENT,
58         // LOG_TOTAL_CURVATURE_ENERGY_FRAGMENT. For more details see the lines 31/195–54/195 of
59         // Listing 2.44/194 and Fig. 3.5/320. (With the exception of DEFAULT_NULL_FRAGMENT all remaining
60         // color schemes should be used together with uniform black (color) materials.)
61         TensorProductSurface3::ImageColorScheme _color_scheme;

62         // a matrix of smart pointers to dynamically allocated B-surface patches
63         Matrix<SP<BSurface3>::Default>         _patches;

64         // a matrix of smart pointers to dynamically allocated B-surface images (i.e., TriangleMesh3 object)
65         Matrix<SP<TriangleMesh3>::Default>     _img_patches;
```


© Ágoston Róth, 2018

```
66          // a matrix of smart pointers to dynamically allocated row matrices that store smart pointers to the
67          // dynamically allocated images (i.e., GenericCurve3 objects) of the u-directional isoparametric lines
68          // of all B-surface patches
69          Matrix<SP< RowMatrix<SP<GenericCurve3>::Default> >::Default>
70                  _u_isoparametric_lines;

71          // a matrix of smart pointers to dynamically allocated row matrices that store smart pointers to the
72          // dynamically allocated images (i.e., GenericCurve3 objects) of the v-directional isoparametric lines
73          // of all B-surface patches
74          Matrix<SP< RowMatrix<SP<GenericCurve3>::Default> >::Default>
75                  _v_isoparametric_lines;

76          // decides whether the two-sided lighting or the reflection lines shader program should be used
77          // during rendering
78          bool      _apply_reflection_lines;                    // synchronize it with a check box

79          // visibility flags that should be synchronized with check boxes of your graphical user interface
80          bool      _show_patches;                              // synchronize it with a check box
81          bool      _show_control_nets;                         // synchronize it with a check box
82          bool      _show_u_isoparametric_lines;                // synchronize it with a check box
83          bool      _show_v_isoparametric_lines;                // synchronize it with a check box
84          bool      _show_tangents_of_u_isoparametric_lines;    // synchronize it with a check box
85          bool      _show_tangents_of_v_isoparametric_lines;    // synchronize it with a check box

86          // determines the transparency of the rendered triangle meshes, its values should be in the interval [0,1]
87          GLfloat       _transparency;

88          // auxiliary private methods used for control point, control net, triangle mesh and tangent vector rendering
89          void _renderSpheresAtControlPoints(
90              const BSurface3 &surface, const Color4 &front_color_material) const;

91          void _renderControlNet(
92              const BSurface3 &surface, const Color4 &color,
93              bool use_dashed_line = false) const;

94          void _renderTransparentMesh(
95              const ShaderProgram &shader, const TriangleMesh3 &mesh,
96              const Color4 &front_color_material, const Color4 &back_color_material,
97              GLfloat transparency = 0.5f) const;

98          bool _renderSpheresAndConesAtEndpointsOfTangentVectors(
99              const GenericCurve3 &curve,
100             const Color4 &front_color_material,
101             const Color4 &back_color_material) const;

102     public:
103         // your default/special constructor
104         YourGLWidget(/*...*/);

105         // your rendering method
106         void render();

107         // your other methods...
108     };
109 }

110 #endif // YOURGLWIDGET_H
```

Listing 3.19. B-representations of ordinary integral surfaces and their isoparametric lines (**YourGLWidget.cpp**)

```
1 #pragma warning(disable:4503)
2 #include "YourGLWidget.h"

3 #include <iostream>
4 #include <fstream>

5 #include <Core/Exceptions.h>
6 #include <Core/Utilities.h>
7 #include <Core/Geometry/Surfaces/Materials.h>
8 #include <Core/Math/Constants.h>
9 #include "../Spaces/SpecializedECSpaces.h"
```


© Ágoston Róth, 2018

```
10 using namespace std;
11 using namespace cagd;
12 using namespace cagd::variable;

13 // your default/special constructor
14 void YourGLWidget::YourGLWidget(/*...*/)
15 {
16     try
17     {
18         // ...
19         // these lines coincide with the lines 15/285–144/287 of Listing 3.9/285
20         // ...

21         // Communicating with our shader programs via uniform variables...
22         _two_sided_lighting.enable();

23         if (!_two_sided_lighting.setUniformDirectionalLight("light_source[0]", *_light))
24         {
25             throw Exception("Two-sided per pixed lighting: could not initialize the "
26                             "uniform variable \"light_source[0]\"!");
27         }

28         if (!_two_sided_lighting.setUniformValue1i("light_source[0].enabled", GL_TRUE))
29         {
30             throw Exception("Two-sided per pixed lighting: could not initialize the "
31                             "uniform variable \"light_source[0].enabled\"!");
32         }

33         _two_sided_lighting.disable();

34         _reflection_lines.enable();

35         if (!_reflection_lines.setUniformDirectionalLight("light_source[0]", *_light))
36         {
37             throw Exception("Reflection lines: could not initialize the "
38                             "uniform variable \"light_source[0]\"!");
39         }

40         if (!_reflection_lines.setUniformValue1i("light_source[0].enabled", GL_TRUE))
41         {
42             throw Exception("Reflection lines: could not initialize the "
43                             "uniform variable \"light_source[0].enabled\"!");
44         }

45         if (!_reflection_lines.setUniformValue1f("scale_factor", 3.0f))
46         {
47             throw Exception("Reflection lines: could not initialize the "
48                             "uniform variable \"scale_factor\"!");
49         }

50         if (!_reflection_lines.setUniformValue1f("smoothing", 1.0f))
51         {
52             throw Exception("Reflection lines: could not initialize the "
53                             "uniform variable \"smoothing\"!");
54         }

55         if (!_reflection_lines.setUniformValue1f("shading", 0.01f))
56         {
57             throw Exception("Reflection lines: could not initialize the "
58                             "uniform variable \"shading\"!");
59         }

60         _reflection_lines.disable();

61         // Loading a triangulated unit sphere centered at the origin. Using translation and scaling transformations,
62         // it will be rendered multiple times at the positions of the control points.
63         if (!_sphere.loadFromOFF("Models/sphere.off"))
64         {
65             throw Exception("Could not load the model file sphere.off!");
66         }

67         if (!_sphere.updateVertexBufferObjects())
68         {
69             throw Exception("Could not update the VBOs of the triangulated sphere!");
```


© Ágoston Róth, 2018

```
70          }
71          // Determines the common scaling factor of the unit sphere that has to be rendered at the positions
72          // of the control points.
73          _control_point_radius = 0.06875;
74          // Determines the scaling transformation of the unit sphere that has to be rendered at the positions
75          // of the control points.
76          _sphere_S.setScalingFactors(
77                  _control_point_radius, _control_point_radius, _control_point_radius);
78          // Loading a triangulated right circular cone with unit base radius and appex (0, 0, 2).
79          // Using translation and scaling transformation, this cone will be rendered multiple times at the endpoints
80          // of the tangent vectors of the isoparametric lines.
81          if (!_cone.loadFromOFF("Models/cone.off"))
82          {
83              throw Exception("Could not load model file cone.off!");
84          }
85          if (!_cone.updateVertexBufferObjects())
86          {
87              throw Exception("Could not update the VBO of the triangulated cone!");
88          }
89          _cone_S.setScalingFactors(_control_point_radius / 1.5f,
90                                    _control_point_radius / 1.5f,
91                                    _control_point_radius / 1.5f);
92          // ...
93          // these lines coincide with the lines 19/296–50/297 of Listing 3.13/296
94          // ...
95          // Memory allocations and parameter settings...
96          _dimension.resize(2);
97          _surface_div_point_count.resize(2);
98          _isoparametric_line_count.resize(2);
99          _maximum_order_of_derivatives.resize(2);
100         _curve_div_point_count.resize(2);
101         _dimension[U]                        =  7;
102         _surface_div_point_count[U]          = 50;
103         _isoparametric_line_count[U]         =  5;
104         _maximum_order_of_derivatives[U]     =  1;
105         _curve_div_point_count[U]            = 20;
106         _dimension[V]                        =  3;
107         _surface_div_point_count[V]          = 70;
108         _isoparametric_line_count[V]         =  3;
109         _maximum_order_of_derivatives[V]     =  1;
110         _curve_div_point_count[V]            = 10;
111         // select a color scheme...
112         _color_scheme = TensorProductSurface3::DEFAULT_NULL_FRAGMENT;
113         // In order to provide B-representations of the patches of the ordinary integral surface
114         // s(u,v) = [ s^0(u,v) ] = [ (1 - e^{ω0 u}) cos(u) (5/4 + cos(v)) ]
                    [ s^1(u,v) ]   [ (e^{ω0 u} - 1) sin(u) (5/4 + cos(v)) ]
                    [ s^2(u,v) ]   [ 7 - e^{ω1 u} - sin(v) + e^{ω0 u} sin(v) ],
115         // we will define the exponential-trigonometric and pure trigonometric EC spaces
116         // ET_6^{α_0^k, β_0^k} = < { φ_{6,0}(u) ≡ 1, φ_{6,1}(u) = cos(u), φ_{6,2}(u) = sin(u), φ_{6,3}(u) = e^{ω0 u}, φ_{6,4}(u) = e^{ω1 u},
117         //                         φ_{6,5}(u) = e^{ω0 u} cos(u), φ_{6,6}(u) = e^{ω0 u} sin(u) : u ∈ [α_0^k, β_0^k] } >, k = 0, 1, ..., 4
118         // and
119         //   T_2^{α_1^ℓ, β_1^ℓ} = < { φ_{2,0}(v) ≡ 1, φ_{2,1}(v) = cos(v), φ_{2,2}(v) = sin(v) : v ∈ [α_1^ℓ, β_1^ℓ] } >, ℓ = 0, 1, ..., ⌊3τ⌋,
120         // respectively, where ω_0 = 1/(6π), ω_1 = 1/(3π), {α_0^k}_{k=0}^4 = { 7π/2 + 29π/40 · k }_{k=0}^4, {β_0^k}_{k=0}^4 = {α_0^k + 29π/40}_{k=0}^4,
121         // {α_1^ℓ}_{ℓ=0}^{⌊3τ⌋} = { -π/(3τ) + 2π/⌊3τ⌋ · ℓ }_{ℓ=0}^{⌊3τ⌋}, {β_1^ℓ}_{ℓ=0}^{⌊3τ⌋} = {α_1^ℓ + 2π/⌊3τ⌋}_{ℓ=0}^{⌊3τ⌋} and τ ≥ 1. Moreover, we have to store
122         // the function coefficients that appear in the parametric form of the given ordinary integral surface.
123         // To achieve this, we will instantiate and initialize an object from the class OrdinarySurfaceCoefficients.
124         // For more details consider the lines 14/250–45/252 of Listing 2.52/250.
125         std::vector<GLint> sigma(3);
```


© Ágoston Róth, 2018

```
126         sigma[0] = 1;
127         sigma[1] = 1;
128         sigma[2] = 3;

129         OrdinarySurfaceCoefficients lambda(_dimension[U], _dimension[V], sigma);

130         // coefficients appearing in the single (i.e., zeroth) seperable product of the coordinate function s^0(u,v)
131         lambda(0, 0, U, 1)     =  1.0;          //  1 · φ_{6,1}(u) = cos(u)
132         lambda(0, 0, U, 5)     = -1.0;          // -1 · φ_{6,5}(u) = -e^{ω_0 u} cos(u)
133         lambda(0, 0, V, 0)     =  5.0 / 4.0;    //  5/4 · φ_{2,0}(v) = 5/4
134         lambda(0, 0, V, 1)     =  1.0;          //  1 · φ_{2,1}(v) = cos(v)

135         // coefficients appearing in the single (i.e., zeroth) seperable product of the coordinate function s^1(u,v)
136         lambda(1, 0, U, 2)     = -1.0;          // -1 · φ_{6,2}(u) = -sin(u)
137         lambda(1, 0, U, 6)     =  1.0;          //  1 · φ_{6,6}(u) = e^{ω_0 u} sin(u)
138         lambda(1, 0, V, 0)     =  5.0 / 4.0;    //  5/4 · φ_{2,0}(v) = 5/4
139         lambda(1, 0, V, 1)     =  1.0;          //  1 · φ_{2,1}(v) = cos(v)

140         // coefficients appearing in the zeroth seperable product of the coordinate function s^2(u,v)
141         lambda(2, 0, U, 0)     =  7.0;          //  7 · φ_{6,0}(u) = 7
142         lambda(2, 0, U, 4)     = -1.0;          // -1 · φ_{6,4}(u) = e^{ω_1 u}
143         lambda(2, 0, V, 0)     =  1.0;          //  1 · φ_{2,0}(v) = 1

144         // coefficients appearing in the first seperable product of the coordinate function s^2(u,v)
145         lambda(2, 1, U, 0)     = -1.0;          // -1 · φ_{6,0}(u) = -1
146         lambda(2, 1, V, 2)     =  1.0;          //  1 · φ_{2,2}(v) = sin(v)

147         // coefficients appearing in the second seperable product of the coordinate function s^2(u,v)
148         lambda(2, 2, U, 3)     =  1.0;          //  1 · φ_{6,3}(u) = e^{ω_0 u}
149         lambda(2, 2, V, 2)     =  1.0;          //  1 · φ_{2,2}(v) = sin(v)

150         GLdouble u_min         =  7.0 * PI / 2.0,
151                  u_max         = 57.0 * PI / 8.0;
152         GLint    row_count     =  5;

153         GLdouble tau           =  2.0; // one can also try other values like 1.0, 1.25, 1.5, 1.75, etc.

154         GLdouble v_min         = -PI / 3.0 / tau,
155                  v_max         = TWO_PI + v_min;
156         GLint    column_count  = floor(3.0 * tau);

157         _patches.resizeRows(row_count);
158         _patches.resizeColumns(column_count);

159         _img_patches.resizeRows(row_count);
160         _img_patches.resizeColumns(column_count);

161         _u_isoparametric_lines.resizeRows(row_count);
162         _u_isoparametric_lines.resizeColumns(column_count);

163         _v_isoparametric_lines.resizeRows(row_count);
164         _v_isoparametric_lines.resizeColumns(column_count);

165         Matrix<SP<RowMatrix< SP<GenericCurve3>::Default> >::Default>
166                 *matrix_of_isoparametric_lines[2] =
167                     {&_u_isoparametric_lines, &_v_isoparametric_lines};

168         GLdouble u_step = (u_max - u_min) / row_count;
169         GLdouble v_step = (v_max - v_min) / column_count;

170         for (GLint k = 0; k < row_count; k++)
171         {
172             GLdouble u = u_min + k * u_step;

173             // the exponential-trigonometric EC space ET_6^{α_0^k, β_0^k}
174             SnailUECSpace ET(u, u + u_step,                                  // α_0^k, β_0^k
175                              check_for_ill_conditioned_matrices,
176                              expected_correct_significant_digits);

177             for (GLint l = 0; l < column_count; l++)
178             {
179                 GLdouble v = v_min + l * v_step;
```


© Ágoston Róth, 2018

```
180             // the first order pure trigonometric EC space T_2^{α_1^ℓ, β_1^ℓ}
181             SnailVECSpace T(v, v + v_step,                          // α_1^ℓ, β_1^ℓ
182                             check_for_ill_conditioned_matrices,
183                             expected_correct_significant_digits);
184             _patches(k, l) = SP<BSurface3>::Default(new BSurface3(ET, T));
185             if (!_patches(k, l))
186             {
187                 throw Exception("Could not create the B-surface patch in row " +
188                                 toString(k) + " and column " + toString(l) + "!");
189             }
190             BSurface3 &surface = *_patches(k, l);
191             if (!surface.updateControlPointsForExactDescription(lambda))
192             {
193                 throw Exception("Could not perform the B-representation of the "
194                                 "ordinary surface patch in row " + toString(k) +
195                                 " and column " + toString(l) + "!");
196             }
197             if (!surface.updateVertexBufferObjectsOfData())
198             {
199                 throw Exception("Could not update the VBO of the control net of the "
200                                 "B-surface patch in row " + toString(k) +
201                                 " and column " + toString(l) + "!");
202             }
203             _img_patches(k, l) = SP<TriangleMesh3>::Default(
204                         surface.generateImage(
205                             _surface_div_point_count[U], _surface_div_point_count[V],
206                             _color_scheme));
207             if (!_img_patches(k, l))
208             {
209                 throw Exception("Could not generate the image of the B-surface in "
210                                 "row " + toString(k) + " and column " +
211                                 toString(l) + "!");
212             }
213             if (!_img_patches(k, l)->updateVertexBufferObjects())
214             {
215                 throw Exception("Could not update the VBOs of the image of the "
216                                 "B-surface in row " + toString(k) + " and column " +
217                                 toString(l) + "!");
218             }
219             for (GLint direction = U; direction <= V; direction++)
220             {
221                 SP< RowMatrix<SP<GenericCurve3>::Default> >::Default
222                     &sp_isoparametric_lines =
223                         (*matrix_of_isoparametric_lines[direction])(k, l);
224                 sp_isoparametric_lines =
225                         SP< RowMatrix<SP<GenericCurve3>::Default> >::Default(
226                             surface.generateIsoparametricLines(
227                                 (variable::Type)direction,
228                                 _isoparametric_line_count[direction],
229                                 _maximum_order_of_derivatives[direction],
230                                 _curve_div_point_count[direction]));
231                 if (!sp_isoparametric_lines)
232                 {
233                     throw Exception("Could not generate the " +
234                                     string(direction ? "v" : "u") + "-directional "
235                                     "isoparametric lines of the B-surface in row " +
236                                     toString(k) + " and column " + toString(l) + "!");
237                 }
238                 RowMatrix<SP<GenericCurve3>::Default>
239                         &isoparametric_lines = *sp_isoparametric_lines;
```


© Ágoston Róth, 2018

```
240                     for (GLint i = 0; i < isoparametric_lines.columnCount(); i++)
241                     {
242                         if (!isoparametric_lines[i]->updateVertexBufferObjects())
243                         {
244                             string th;

245                             switch (i % 4)
246                             {
247                             case 1:  th = "st"; break;
248                             case 2:  th = "nd"; break;
249                             case 3:  th = "rd"; break;
250                             default: th = "th"; break;
251                             }

252                             throw Exception("Could not update the VBOs of the " +
253                                             toString(i) + th + " " +
254                                             string(direction ? "v" : "u") +
255                                             "-directional isoparametric line of "
256                                             "the B-surface in row " + toString(k) +
257                                             " and column " + toString(l) + "!");
258                         }
259                     }
260                 }
261             }
262         }

263         _apply_reflection_lines                 = false; // synchronize it with a check box
264         _show_patches                           = true;  // synchronize it with a check box
265         _show_control_nets                      = true;  // synchronize it with a check box
266         _show_u_isoparametric_lines             = false; // synchronize it with a check box
267         _show_v_isoparametric_lines             = false; // synchronize it with a check box
268         _show_tangents_of_u_isoparametric_lines = false; // synchronize it with a check box
269         _show_tangents_of_v_isoparametric_lines = false; // synchronize it with a check box
270         _transparency                           = 0.0f;  // synchronize it with a double spin box

271         glClearColor(1.0, 1.0, 1.0, 1.0);           // set the background color
272         glEnable(GL_DEPTH_TEST);                    // enable depth testing
273         glEnable(GL_LINE_SMOOTH);                   // enable the anti-aliasing of line primitives
274         glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);
275         glEnable(GL_POINT_SMOOTH);                  // enable the anti-aliasing of point primitives
276         glHint(GL_POINT_SMOOTH_HINT, GL_NICEST);
277     }
278     catch (Exception &e)
279     {
280         cout << e << endl;
281     }
282 }

283 GLboolean GLWidget::_updateTransformationMatrices()
284 {
285     // ...
286     // these lines coincide with the lines 224/288–233/289 of Listing 3.9/285
287     // ...
288 }

289 // your rendering method
290 void YourGLWidget::render()
291 {
292     // clear the color and depth buffers
293     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

294     // revert to the original transformation matrices in case of the color shader program
295     _color_shader.enable();
296         _color_shader.setUniformMatrix4fv("PVM", 1, GL_FALSE, _PVM.address());
297     _color_shader.disable();

298     // At first, we render non-transparent geometries like control nets, spheres that represent control points,
299     // isoparametric lines and their tangent vectors:

300     bool show_isoparametric_lines[2] =
301         {_show_u_isoparametric_lines, _show_v_isoparametric_lines};

302     bool show_tangents_of_isoparametric_lines[2] =
```


© Ágoston Róth, 2018

```
303         {_show_tangents_of_u_isoparametric_lines,
304          _show_tangents_of_v_isoparametric_lines};
305     const Color4 *isoparametric_line_color[2] = {&colors::dark_purple, &colors::blue};
306     const Color4 *isoparametric_line_tangent_color[2] = {&colors::purple, &colors::cyan};
307     const Matrix<SP<RowMatrix< SP<GenericCurve3>::Default> >::Default>
308             *matrix_of_isoparametric_lines[2] =
309                 {&_u_isoparametric_lines, &_v_isoparametric_lines};
310     for (GLint direction = U; direction <= V; direction++)
311     {
312         // try to render the isoparametric lines in the selected direction...
313         if (show_isoparametric_lines[direction] &&
314             matrix_of_isoparametric_lines[direction])
315         {
316             const Matrix<SP<RowMatrix< SP<GenericCurve3>::Default> >::Default>
317                     &matrix = *matrix_of_isoparametric_lines[direction];
318             _color_shader.enable();
319                 glLineWidth(3.0);
320                 for (GLint k = 0; k < matrix.rowCount(); k++)
321                 {
322                     for (GLint l = 0; l < matrix.columnCount(); l++)
323                     {
324                         if (matrix(k, l))
325                         {
326                             const RowMatrix<SP<GenericCurve3>::Default>
327                                     &isoparametric_lines = *matrix(k, l);
328                             _color_shader.setUniformColor(
329                                     "color", *isoparametric_line_color[(k + l) % 2]);
330                             for (GLint i = 0; i < isoparametric_lines.columnCount(); i++)
331                             {
332                                 if (isoparametric_lines[i])
333                                 {
334                                     isoparametric_lines[i]->renderDerivatives(
335                                             0, GL_LINE_STRIP);
336                                 }
337                             }
338                         }
339                     }
340                 }
341                 glLineWidth(1.0);
342             _color_shader.disable();
343         }
344         // try to render the tangent vectors of the isoparametric lines in the selected direction...
345         if (show_tangents_of_isoparametric_lines[direction] &&
346             matrix_of_isoparametric_lines[direction])
347         {
348             const Matrix<SP<RowMatrix< SP<GenericCurve3>::Default> >::Default>
349                     &matrix = *matrix_of_isoparametric_lines[direction];
350             for (GLint k = 0; k < matrix.rowCount(); k++)
351             {
352                 GLint index_offset = (k == matrix.rowCount() - 1);
353                 for (GLint l = 0; l < matrix.columnCount(); l++)
354                 {
355                     if (matrix(k, l))
356                     {
357                         const RowMatrix<SP<GenericCurve3>::Default>
358                                 &isoparametric_lines = *matrix(k, l);
359                         const Color4 &front_color =
360                                 *isoparametric_line_tangent_color[(k + l) % 2];
```


© Ágoston Róth, 2018

```
361                         for (GLint i = 0;
362                              i <= direction *
363                                   (isoparametric_lines.columnCount() - 2 + index_offset);
364                              i++)
365                         {
366                             if (isoparametric_lines[i])
367                             {
368                                 const GenericCurve3 &curve = *isoparametric_lines[i];

369                                 _color_shader.enable();
370                                     _color_shader.setUniformColor("color", front_color);
371                                     curve.renderDerivatives(1, GL_LINES);
372                                 _color_shader.disable();

373                                 _renderSpheresAndConesAtEndpointsOfTangentVectors(
374                                         curve, front_color, colors::silver);
375                             }
376                         }
377                     }
378                 }
379             }

380         }
381     }

382     // try to render the control nets of the B-surface patches...
383     if (_show_control_nets)
384     {
385         for (GLint k = 0; k < _patches.rowCount(); k++)
386         {
387             for (GLint l = 0; l < _patches.columnCount(); l++)
388             {
389                 if (_patches(k, l))
390                 {
391                     const BSurface3 &surface = *_patches(k, l);

392                     if ((k + l) % 2)
393                     {
394                         _renderSpheresAtControlPoints(
395                             surface, colors::ice_blue, colors::silver);
396                     }
397                     else
398                     {
399                         _renderSpheresAtControlPoints(
400                             surface, colors::purple, colors::silver);
401                     }

402                     _renderControlNet(surface, colors::gray);
403                 }
404             }
405         }
406     }

407     // Second, we render possible transparent geometries like the triangle mesh images of all B-surfaces:
408     if (_show_patches)
409     {
410         glEnable(GL_POLYGON_OFFSET_FILL);
411         glPolygonOffset(30, 1.0);

412         // select the constant reference of either the two-sided lighting or the reflection line generating shader program...
413         const ShaderProgram &surface_shader = _apply_reflection_lines ?
414                                               _reflection_lines : _two_sided_lighting;

415         // revert to the original transformation matrices...
416         surface_shader.enable();

417             surface_shader.setUniformMatrix4fv("VM",  1, GL_FALSE, _VM.address());
418             surface_shader.setUniformMatrix4fv("PVM", 1, GL_FALSE, _PVM.address());
419             surface_shader.setUniformMatrix4fv("N",   1, GL_TRUE,  _tN.address());

420         surface_shader.disable();

421         // in order to obtain the best coloring effects, color schemes different than the DEFAULT_NULL_FRAGMENT
422         // should be used together with black uniform (color) materials...
```


© Ágoston Róth, 2018

```
423         bool default_color_scheme =
424                 (_color_scheme == TensorProductSurface3::DEFAULT_NULL_FRAGMENT);
425         for (GLint k = 0; k < _img_patches.rowCount(); k++)
426         {
427             for (GLint l = 0; l < _img_patches.columnCount(); l++)
428             {
429                 if (_img_patches(k, l))
430                 {
431                     const TriangleMesh3 &mesh = *_img_patches(k, l);
432                     if ((k + l) % 2)
433                     {
434                         _renderTransparentMesh(
435                             surface_shader, mesh,
436                             default_color_scheme ? colors::baby_blue  : colors::black,
437                             default_color_scheme ? colors::light_blue : colors::black,
438                             _transparency);
439                     }
440                     else
441                     {
442                         _renderTransparentMesh(
443                             surface_shader, mesh,
444                             default_color_scheme ? colors::purple       : colors::black,
445                             default_color_scheme ? colors::light_purple : colors::black,
446                             _transparency);
447                     }
448                 }
449             }
450         }
451         glDisable(GL_POLYGON_OFFSET_FILL);
452     }
453 }
454 // auxiliary private method used for control point rendering
455 void YourGLWidget::_renderSpheresAtControlPoints(
456         const BSurface3 &surface,
457         const Color4 &front_color_material, const Color4 &back_color_material) const
458 {
459     // ...
460     // these lines coincide with the lines 402/315–424/315 of Listing 3.17/308
461     // ...
462 }
463 // auxiliary private method used for control net rendering
464 void YourGLWidget::_renderControlNet(
465         const BSurface3 &surface, const Color4 &color, bool use_dashed_line) const
466 {
467     // ...
468     // these lines coincide with the lines 430/315–445/316 of Listing 3.17/308
469     // ...
470 }
471 // auxiliary private method used for transparent triangle mesh rendering
472 void YourGLWidget::_renderTransparentMesh(
473         const ShaderProgram &shader, const TriangleMesh3 &mesh,
474         const Color4 &front_color_material, const Color4 &back_color_material,
475         GLfloat transparency) const
476 {
477     // ...
478     // these lines coincide with the lines 453/316–494/316 of Listing 3.17/308
479     // ...
480 }
481 // auxiliary private method used for tangent vector rendering
482 bool GLWidget::_renderSpheresAndConesAtEndpointsOfTangentVectors(
483         const GenericCurve3 &curve,
484         const Color4 &front_color_material, const Color4 &back_color_material) const
485 {
486     if (curve.maximumOrderOfDerivatives() < 1)
487     {
488         return false;
489     }
```


© Ágoston Róth, 2018

```
490     _two_sided_lighting.enable();

491         _two_sided_lighting.setUniformColorMaterial(
492                 "front_material", front_color_material);
493         _two_sided_lighting.setUniformColorMaterial(
494                 "back_material", back_color_material);

495         for (int i = 0; i < curve.pointCount(); i++)
496         {
497             const Cartesian3 &point = curve(0, i);

498             Translate         sphere_T(point[0], point[1], point[2]);
499             GLTransformation sphere_VM  = _VM * sphere_T * _cone_S;
500             GLTransformation sphere_PVM = (*_P) * sphere_VM;

501             _two_sided_lighting.setUniformMatrix4fv(
502                     "VM", 1, GL_FALSE, sphere_VM.address());
503             _two_sided_lighting.setUniformMatrix4fv(
504                     "PVM", 1, GL_FALSE, sphere_PVM.address());
505             _two_sided_lighting.setUniformMatrix4fv(
506                     "N", 1, GL_TRUE,  sphere_VM.inverse().address());

507             _sphere.render(_two_sided_lighting);

508             const Cartesian3 &tangent = curve(1, i);

509             Cartesian3 sum = point; sum += tangent;
510             Cartesian3 K = tangent, I((GLdouble)rand() / (GLdouble)RAND_MAX,
511                                       (GLdouble)rand() / (GLdouble)RAND_MAX,
512                                       (GLdouble)rand() / (GLdouble)RAND_MAX);
513             K.normalize();
514             I.normalize();

515             Cartesian3 J = K ^ I; J.normalize();

516             I = J ^ K; I.normalize();

517             GLTransformation C;

518             C[ 0] =   I[0], C[ 1] =   I[1], C[ 2] =   I[2], C[ 3] = 0.0f;
519             C[ 4] =   J[0], C[ 5] =   J[1], C[ 6] =   J[2], C[ 7] = 0.0f;
520             C[ 8] =   K[0], C[ 9] =   K[1], C[10] =   K[2], C[11] = 0.0f;
521             C[12] = sum[0], C[13] = sum[1], C[14] = sum[2], C[15] = 1.0f;

522             GLTransformation cone_VM  = _VM * C * _cone_S;
523             GLTransformation cone_PVM = (*_P) * cone_VM;

524             _two_sided_lighting.setUniformMatrix4fv(
525                     "VM", 1, GL_FALSE, cone_VM.address());
526             _two_sided_lighting.setUniformMatrix4fv(
527                     "PVM", 1, GL_FALSE, cone_PVM.address());
528             _two_sided_lighting.setUniformMatrix4fv(
529                     "N", 1, GL_TRUE,  cone_VM.inverse().address());

530             _cone.render(_two_sided_lighting);
531         }

532     _two_sided_lighting.disable();

533     return true;
534 }

535 // The following comments specify the parameter settings that have to be applied in the constructor of the
536 // class YourGLWidget in order to obtain the images shown in Figs. 3.8/334(a)–(b) and 3.9/335(a)–(c).

537 // Fig. 3.8/334(a) was obtained by using the parameter settings:
538 //         _apply_reflection_lines = false;
539 //         _show_patches = true;
540 //         _show_control_nets = true;
541 //         _show_u_isoparametric_lines = false;
542 //         _show_v_isoparametric_lines = false;
543 //         _show_tangents_of_u_isoparametric_lines = false;
544 //         _show_tangents_of_v_isoparametric_lines = false;
```


© Ágoston Róth, 2018

```
// Fig. 3.8/334(b) was obtained by using the parameter settings:
//          apply_reflection_lines = true;
//          show_patches = true;
//          show_control_nets = true;
//          show_u_isoparametric_lines = false;
//          show_v_isoparametric_lines = false;
//          show_tangents_of_u_isoparametric_lines = false;
//          show_tangents_of_v_isoparametric_lines = false;

// Fig. 3.9/335(a) was obtained by using the parameter settings:
//          apply_reflection_lines = false;
//          show_patches = true;
//          show_control_nets = true;
//          show_u_isoparametric_lines = true;
//          show_v_isoparametric_lines = true;
//          show_tangents_of_u_isoparametric_lines = false;
//          show_tangents_of_v_isoparametric_lines = false;

// Fig. 3.9/335(b) was obtained by using the parameter settings:
//          apply_reflection_lines = false;
//          show_patches = true;
//          show_control_nets = false;
//          show_u_isoparametric_lines = true;
//          show_v_isoparametric_lines = false;
//          show_tangents_of_u_isoparametric_lines = true;
//          show_tangents_of_v_isoparametric_lines = false;

// Fig. 3.9/335(c) was obtained by using the parameter settings:
//          apply_reflection_lines = false;
//          show_patches = true;
//          show_control_nets = false;
//          show_u_isoparametric_lines = false;
//          show_v_isoparametric_lines = true;
//          show_tangents_of_u_isoparametric_lines = false;
//          show_tangents_of_v_isoparametric_lines = true;
```

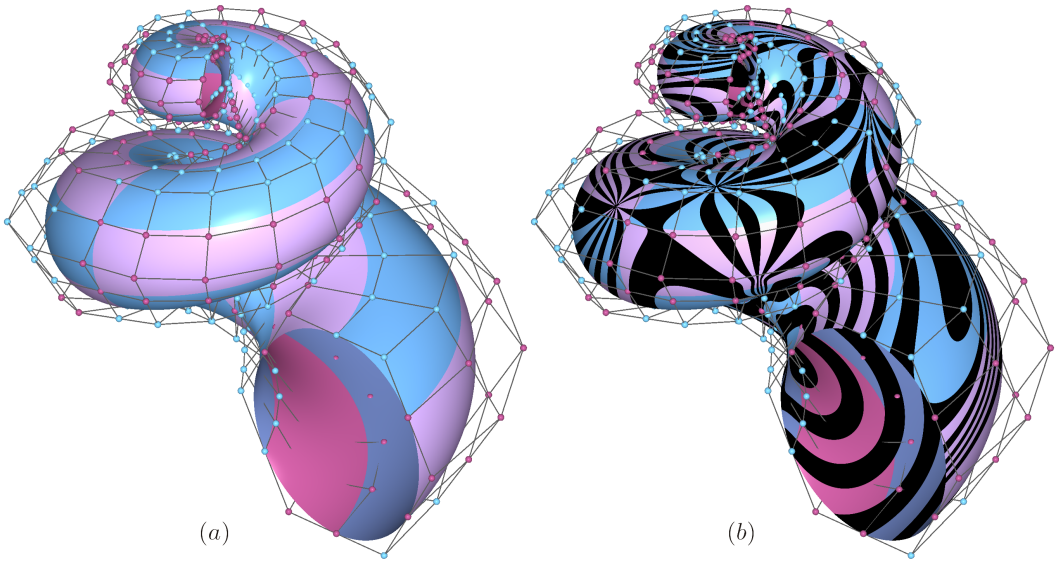


Fig. 3.8: A possible control point based exact description (or B-representation) of the ordinary exponential-trigonometric integral surface (3.9/320) by means of a 5×6 matrix of B-surface patches. (a) Using alternating simple color materials, the obtained B-surface patches are rendered together with their control nets. (b) Using alternating simple color materials, the obtained B-surface patches are rendered together with their control nets and smooth reflection lines. (The figure was generated by the source codes given in Listings 3.18/321 and 3.19/324.)

© Ágoston Róth, 2018

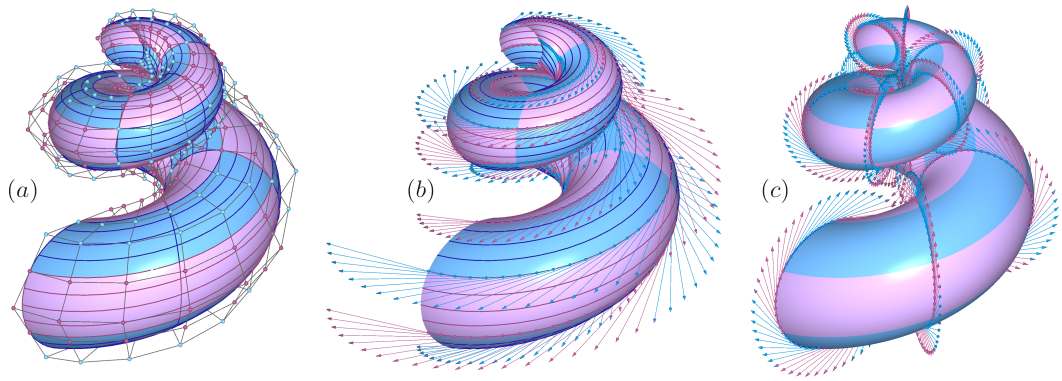


Fig. 3.9: Isoparametric lines and their tangent vectors in case of a possible B-representation of the ordinary exponential-trigonometric integral surface (3.9/320). Along all $5 \times 6 = 30$ B-surface patches we have generated 5 and 3 isoparametric lines in directions u and v, respectively. The u- and v-isoparametric lines consist of 20 and 10 subdivision points, respectively. (For better visibility, we have rendered the tangent vectors only of some of the isoparametric lines.) (a) Surface patches, rendered with alternating simple color materials together with their control nets and isoparametric lines. (b) u-directional isoparametric lines and their tangent vectors. (c) v-directional isoparametric lines and their tangent vectors. (The figure was generated by the source codes given in Listings 3.18/321 and 3.19/324.)

© Ágoston Róth, 2018

© Ágoston Róth, 2018

Bibliography

A. Alexandrescu. 2001. *Modern C++ Design: Generic Programming and Design Patterns Applied, 1st edition.* Addison-Wesley Professional, USA.

M. Brilleaud and M.-L. Mazure. 2012. Mixed hyperbolic/trigonometric spaces for design. *Computers and Mathematics with Applications* 64, 8 (2012), 2459–2477. DOI:http://dx.doi.org/10.1016/j.camwa.2012.05.019

J.-M. Carnicer, E. Mainar, and J. M. Peña. 2004. Critical length for design purposes and extended Chebyshev spaces. *Constructive Approximation* 20, 1 (2004), 55–71. DOI:http://dx.doi.org/10.1007/s00365-002-0530-1

J.-M. Carnicer, E. Mainar, and J. M. Peña. 2007. Shape preservation regions for six-dimensional spaces. *Advances in Computational Mathematics* 26, 1–3 (2007), 121–136. DOI:http://dx.doi.org/10.1007/s10444-005-7505-2

J.-M. Carnicer, E. Mainar, and J. M. Peña. 2014. On the critical length of cycloidal spaces. *Constructive Approximation* 39, 3 (2014), 573–583. DOI:http://dx.doi.org/10.1007/s00365-013-9223-1

J.-M. Carnicer and J. M. Peña. 1993. Shape preserving representations and optimality of the Bernstein basis. *Advances in Computational Mathematics* 1, 2 (1993), 173–196. DOI:http://dx.doi.org/10.1007/BF02071384

J.-M. Carnicer and J. M. Peña. 1995. On transforming a Tchebycheff system into a strictly totally positive system. *Journal of Approximation Theory* 81, 2 (1995), 274–295. DOI:http://dx.doi.org/10.1006/jath.1995.1050

P. Costantini, T. Lyche, and C. Manni. 2005. On a class of weak Tchebycheff systems. *Numer. Math.* 101, 2 (2005), 333–354. DOI:http://dx.doi.org/10.1007/s00211-005-0613-6

M. Gasca and J. M. Peña. 1996. On factorizations of totally positive matrices. In *Total Positivity and its Applications*, M. Gasca and C. A. Micchelli (Eds.). Kluwer Academic, Dordrecht, 109–130. DOI:http://dx.doi.org/10.1007/978-94-015-8674-0_7

S. Karlin and W. Studden. 1966. *Tchebycheff Systems: with Applications in Analysis and Statistics.* Wiley, New York, NY.

Y. Lü, G. Wang, and X. Yang. 2002. Uniform hyperbolic polynomial B-spline curves. *Computer Aided Geometric Design* 19, 6 (2002), 379–393. DOI:http://dx.doi.org/10.1016/S0167-8396(02)00092-4

T. Lyche. 1985. A recurrence relation for Chebyshevian B-splines. *Constructive Approximation* 1, 1 (1985), 155–173. DOI:http://dx.doi.org/10.1007/BF01890028

E. Mainar and J. M. Peña. 1999. Corner cutting algorithms associated with optimal shape preserving representations. *Computer Aided Geometric Design* 16, 9 (1999), 883–906. DOI:http://dx.doi.org/10.1016/S0167-8396(99)00035-7

© Ágoston Róth, 2018

E. Mainar and J. M. Peña. 2004. Quadratic-cycloidal curves. *Advances in Computational Mathematics* 20, 1–3 (2004), 161–175. DOI:http://dx.doi.org/10.1023/A:1025813919473

E. Mainar and J. M. Peña. 2010. Optimal bases for a class of mixed spaces and their associated spline spaces. *Computers and Mathematics with Applications* 59, 4 (2010), 1509–1523. DOI:http://dx.doi.org/10.1016/j.camwa.2009.11.009

E. Mainar, J. M. Peña, and J. M. Sánchez-Reyes. 2001. Shape preserving alternatives to the rational Bézier model. *Computer Aided Geometric Design* 18, 1 (2001), 37–60. DOI:http://dx.doi.org/10.1016/S0167-8396(01)00011-5

M.-L. Mazure. 1999. Chebyshev–Bernstein bases. *Computer Aided Geometric Design* 16, 7 (1999), 649–669. DOI:http://dx.doi.org/10.1016/S0167-8396(99)00029-1

M.-L. Mazure and P.-J. Laurent. 1998. Nested sequences of Chebyshev spaces and shape parameters. *RAIRO–Modélisation mathématique et analyse numérique* 32, 6 (1998), 773–788. http://www.numdam.org/article/M2AN_1998__32_6_773_0.pdf

H. Pottmann. 1993. The geometry of Tchebycheffian splines. *Computer Aided Geometric Design* 10, 3–4 (1993), 181–210. DOI:http://dx.doi.org/10.1016/0167-8396(93)90036-3

W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. 2007. *Numerical Recipes 3rd Edition: The art of Scientific Computing.* Cambridge University Press, New York.

Randi J. Rost and Bill Licea-Kane. 2006. *OpenGL Shading Language, 2nd edition.* Addison-Wesley Professional, USA.

Á. Róth. 2015a. Control point based exact description of curves and surfaces in extended Chebyshev spaces. *Computer Aided Geometric Design* 40 (December 14 2015), 40–58. DOI:http://dx.doi.org/10.1016/j.cagd.2015.09.005

Á. Róth. 2015b. Control point based exact description of trigonometric/hyperbolic curves, surfaces and volumes. *J. Comput. Appl. Math.* 290, C (December 1 2015), 74–91. DOI:http://dx.doi.org/10.1016/j.cam.2015.05.003

Á. Róth. 2019. Algorithm 992: An OpenGL- and C++-based function library for curve and surface modeling in a large class of extended Chebyshev spaces. *ACM Trans. Math. Software* 45, 1 (March 2019), Article 13 (32 pages). DOI:http://dx.doi.org/10.1145/3284979

J. Sánchez-Reyes. 1998. Harmonic rational Bézier curves, p-Bézier curves and trigonometric polynomials. *Computer Aided Geometric Design* 15, 9 (1998), 909–923. DOI:http://dx.doi.org/10.1016/S0167-8396(98)00031-4

L. L. Schumaker. 2007. *Spline Functions: Basic Theory, 3rd edition.* Cambridge University Press, UK. DOI:http://dx.doi.org/10.1017/CBO9780511618994

W.-Q. Shen and G.-Z. Wang. 2005. A class of quasi Bézier curves based on hyperbolic polynomials. *Journal of Zhejiang University SCIENCE* 6A (Suppl. I), 9 (2005), 116–123. DOI:http://dx.doi.org/10.1007/BF02887226

© Ágoston Róth, 2018