Abstract tensor product surfaces

– projects –

– a CAGD approach based on OpenGL and C++ –

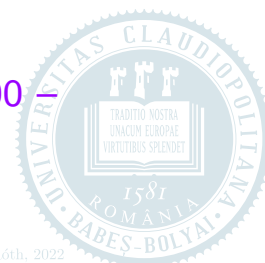
Ágoston Róth

Department of Mathematics and Computer Science, Babeș-Bolyai University, Cluj-Napoca, Romania

(agoston.roth@gmail.com)

Lectures 5 & 6 – March 28 (Part 1) & April 1 & April 4 (Part 2), 2022

– project topics will be specified/assigned on Zoom on April 1 between 15.00 and 17.00 –



Reminder: in CAGD the most widespread description form of curves is the *linear combination*

$$\begin{cases} \mathbf{c} : [u_{\min}, u_{\max}] \to \mathbb{R}^{\delta}, \ \delta \geq 2, \\ \mathbf{c}(u) = \sum_{i=0}^{n} \mathbf{p}_i F_{n,i}(u) \end{cases} \tag{1}$$

of vectors $\mathbf{p}_i \in \mathbb{R}^{\delta}$ and smooth linearly independent non-negative normalized functions of the system

$$\mathcal{F}_n = \{F_{n,i} : [u_{\min}, u_{\max}] \to \mathbb{R}\}_{i=0}^{n}. \tag{2}$$

- In most cases vectors $[\mathbf{p}_i]_{i=0}^{n}$ represent control points forming a control polygon. However, these vectors may correspond to other geometric properties such as tangent and acceleration vectors.
- This means that a curve can be specified by just a few user defined information, which is advantageous from the point of view of data storage and transmission.
- Observe, that curve (1) can be written into the matrix form

$$\mathbf{c}(u) = \begin{bmatrix} \mathbf{p}_0 & \mathbf{p}_1 & \cdots & \mathbf{p}_n \end{bmatrix} \begin{bmatrix} F_{n,0}(u) \\ F_{n,1}(u) \\ \vdots \\ F_{n,n}(u) \end{bmatrix}, \ \forall u \in [u_{\min}, u_{\max}]. \tag{3}$$

- The linear independence of functions (2) is also important, e.g., when interpolating data points by means of the curve (1). The *curve interpolation problem* can be formulated as follows.
- Consider the sequence of data points $\mathbf{d}_j \in \mathbb{R}^\delta$ $(j = 0, 1, \ldots, n)$ associated with parameter values

$$u_{\min} \leq u_0 < u_1 < \ldots < u_n \leq u_{\max}. \tag{4}$$

We refer to the sequence

$$\{(u_j, \mathbf{d}_j)\}_{j=0}^n$$

of pairs as nodes.

- The task is to find control points $\mathbf{p}_i \in \mathbb{R}^\delta$ $(i = 0, 1, \ldots, n)$ for which the *interpolation conditions*

$$\mathbf{c}(u_j) = \sum_{i=0}^{n} \mathbf{p}_i F_{n,i}(u_j) = \mathbf{d}_j,\ j = 0, 1, \ldots, n$$

hold.

- The problem is equivalent with the solution of the linear system

$$\begin{bmatrix} F_{n,0}(u_0) & F_{n,1}(u_0) & \cdots & F_{n,n}(u_0) \\ F_{n,0}(u_1) & F_{n,1}(u_1) & \cdots & F_{n,n}(u_1) \\ \vdots & \vdots & \ddots & \vdots \\ F_{n,0}(u_n) & F_{n,1}(u_n) & \cdots & F_{n,n}(u_n) \end{bmatrix} \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_n \end{bmatrix} = \begin{bmatrix} \mathbf{d}_0 \\ \mathbf{d}_1 \\ \vdots \\ \mathbf{d}_n \end{bmatrix}, \tag{5}$$

which admits a unique solution provided that the function system (2) is linearly independent.

- A drawback of this method is that the interpolating curve will be globally controlled eihter by the data points $[\mathbf{d}_j]_{j=0}^n$ or by the parameter values $\{u_j\}_{j=0}^n$ associated with them, i.e., the displacement of any $\mathbf{d}_j$ or the variation of any u_j results in the change of the shape of the whole curve, even if the combining functions (2) have local support.
- Fig. 1 shows this phenomenon in case of fixed data points, but varying parameter values.

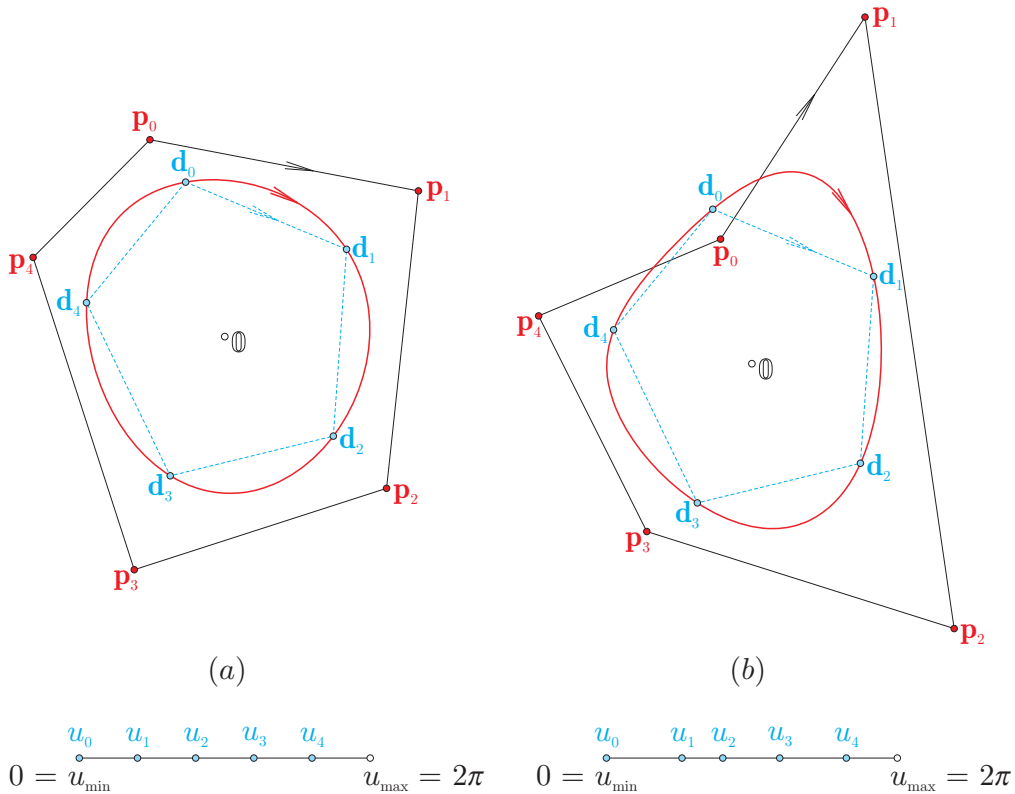


Fig. 1: Different solutions of second order cyclic curve interpolation problems in case of which the data points $[\mathbf{d}_j]_{j=0}^4$ to be interpolated are fixed, but the strictly increasing parameter values $\{u_j\}_{j=0}^4 \subset [0, 2\pi]$ assigned to them follow (a) uniform and (b) non-uniform distribution, respectively.

- Assume that the function system (2) is normalized and linearly independent, and also consider the non-negative *weight vector* $\mathbf{w} = [w_i]_{i=0}^n$ of rank 1, i.e., $w_i \geq 0$ $(i = 0, 1, \ldots, n)$ and $\langle \mathbf{w}, \mathbf{w} \rangle \neq 0$.
- In this case, the *rational (or quotient) functions*

$$R_{i,n}^{\mathbf{w}}(u) = \frac{w_i F_{n,i}(u)}{\sum_{j=0}^{n} w_j F_{n,j}(u)}, \; u \in [u_{\min}, u_{\max}], \; i = 0, 1, \ldots, n$$

form a new system

$$\mathcal{R}_n^{\mathbf{w}} = \left\{ R_{n,i}^{\mathbf{w}}(u) : u \in [u_{\min}, u_{\max}] \right\}_{i=0}^{n} \tag{6}$$

that inherits the properties of (2), i.e., the system $\mathcal{R}_n^{\mathbf{w}}$ is also normalized and linearly independent.

- By means of these quotient basis functions we can generate curves like

$$\mathbf{c}^{\mathbf{w}}(u) = \sum_{i=0}^{n} \mathbf{p}_i R_{n,i}^{\mathbf{w}}(u) = \frac{1}{\sum_{j=0}^{n} w_j F_{n,j}(u)} \sum_{i=0}^{n} w_i \mathbf{p}_i F_{n,i}(u), \; u \in [u_{\min}, u_{\max}] \tag{7}$$

which is of type (1) and it is called the *rational counterpart* of (1).

Remark

The application of rational curves and surfaces in geometric modeling was pioneered in [Coons, 1964, Coons, 1968, Forrest, 1968, Versprille, 1975].

- Observe, that in formula (7) the weights w_i can also be associated with the control points $\mathbf{p}_i$ $(i = 0, 1, \ldots, n)$.
- Thus, besides the usual control point repositioning, we have gained a new modeling tool, namely, by altering the weight of a fixed control point we can increase or decrease its effect on the shape of the *rational curve* (7) as it is shown in Fig. 2.

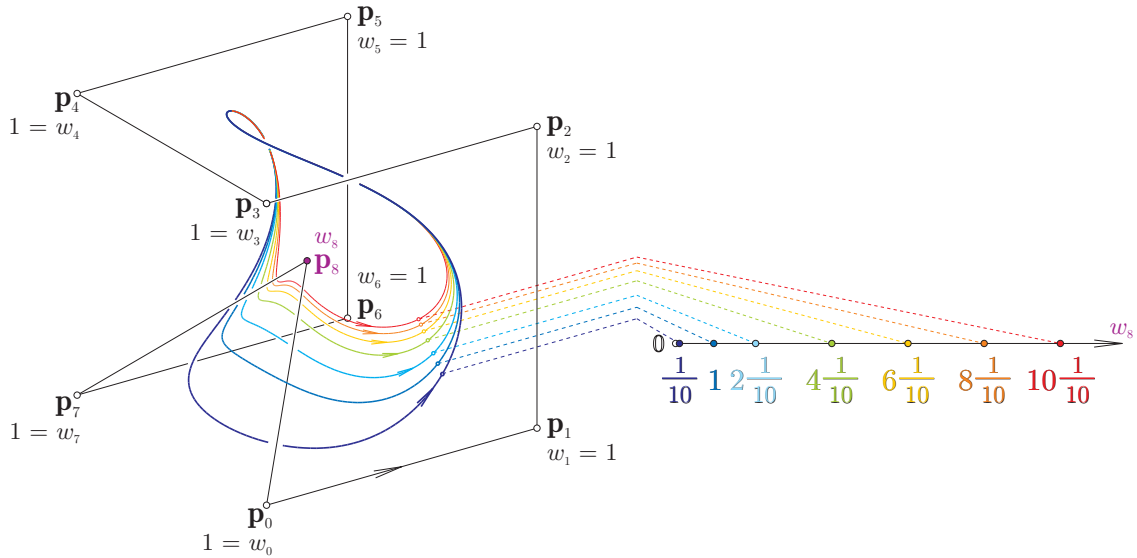


Fig. 2: Weight based deformation of a rational cyclic curve of order 4. Apart from the non-negative weight w_8 all parameters and control points are fixed. Observe, that as the weight w_8 increases, the effect on the shape of the curve of the control point $\mathbf{p}_8$ increases as well.

- It is clear that the weights w_i $(i = 0, 1, \ldots, n)$ are determined up to a strictly positive scaling factor, i.e., in case of a fixed control polygon $[\mathbf{p}_i]_{i=0}^n$ the non-negative weight vectors $\mathbf{w} = [w_i]_{i=0}^n$ and $\lambda\mathbf{w} = [\lambda w_i]_{i=0}^n$ of rank 1 specify the same curve (7) for all values of $\lambda > 0$.
- If all weights are equal, the function system (6) degenerates to the original set (2) of basis functions as a special case.
- The rational curve (7) can also be considered as the *central projection* of the curve

$$\begin{cases} \mathbf{c}_{\wp}^{\mathbf{w}} : [u_{\min}, u_{\max}] \to \mathbb{R}^{\delta+1}, \\ \mathbf{c}_{\wp}^{\mathbf{w}}(u) = \displaystyle\sum_{i=0}^{n} \begin{bmatrix} w_i \mathbf{p}_i \\ w_i \end{bmatrix} F_{n,i}(u) \end{cases} \tag{8}$$

  in the $\delta + 1$ dimensional space from the origin onto the δ dimensional hyperplane $x_\delta = 1$ (assuming that the coordinates of the Euclidean space $\mathbb{R}^{\delta+1}$ are denoted by $x_0, x_1, \ldots, x_\delta$).
- The curve (8) is called the *pre-image* of the rational curve (7), while the vector space $\mathbb{R}^{\delta+1}$ is called its *pre-image space*.
- Figs. 3 and 4 provide examples for this curve generation method.

- The control point based exact description of Bernoulli's lemniscate shown in Fig. 3 is based on the rational counter part of the cyclic basis functions introduced in the case study of Lecture 3 (pp. 55–62, cf. articles [Juhász, Róth, 2010, Róth et al., 2009]).

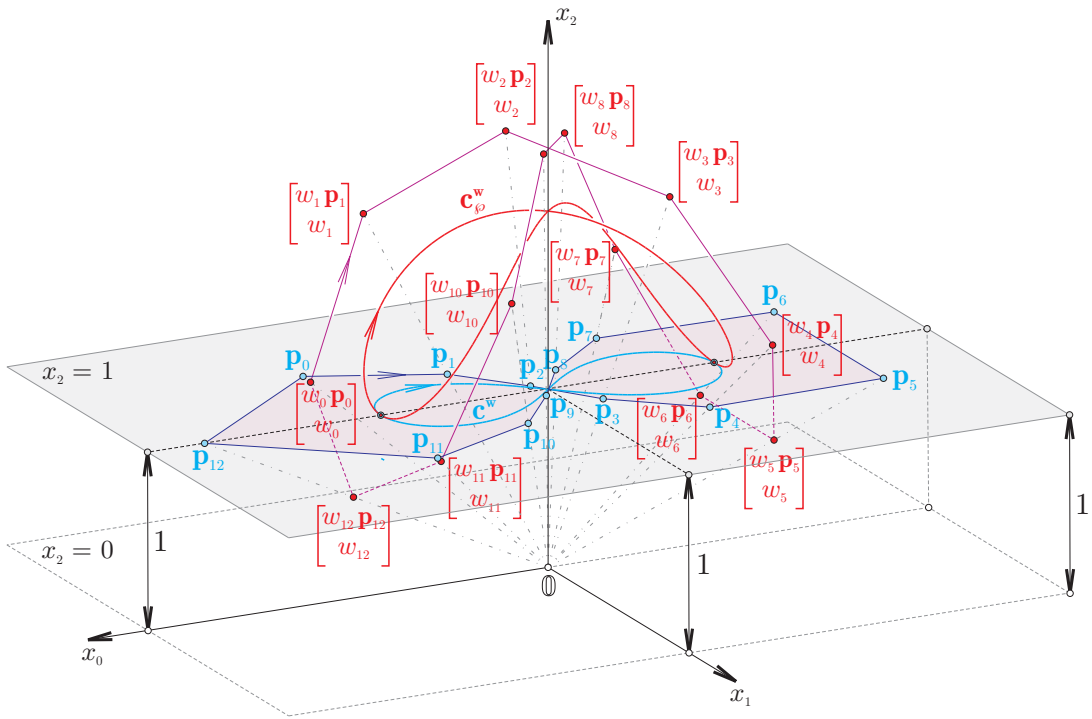


Fig. 3: Control point based exact description of Bernoulli's lemniscate by means of a rational cyclic curve of order 6 (blue) and its pre-image (red).

- The control point based exact description of the Zhukovsky airfoil profile shown in Fig. 4 is based on the rational counter part of the cyclic basis functions introduced in the case study of Lecture 3 (pp. 55–62, cf. articles [Juhász, Róth, 2010, Róth et al., 2009]).

Fig. 4: Control point based exact description of a Zhukovsky airfoil profile by means of a rational cyclic curve of order 2.

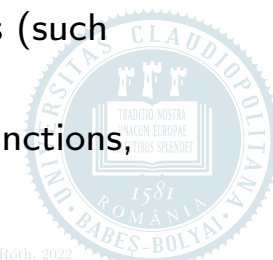
Reminder: in CAGD the general form of surface description is

$$\begin{cases} \mathbf{s} : [u_{\min}, u_{\max}] \times [v_{\min}, v_{\max}] \to \mathbb{R}^3, \\ \mathbf{s}(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} \mathbf{p}_{i,j} F_{n,i}(u) G_{m,j}(v), \end{cases} \tag{9}$$

where:

- the user defined net
$$[\mathbf{p}_{i,j}]_{i=0,j=0}^{n,m} \in \mathcal{M}_{n+1,m+1}\left(\mathbb{R}^3\right)$$
consists of vectors that usually represent position vectors (also called control points),
- while function systems
$$\mathcal{F}_n = \{F_{n,i} : [u_{\min}, u_{\max}] \to \mathbb{R}\}_{i=0}^{n}$$
and
$$\mathcal{G}_m = \{G_{m,j} : [v_{\min}, v_{\max}] \to \mathbb{R}\}_{j=0}^{m}$$
are bases of some vector spaces $\mathbb{F}_n$ and $\mathbb{G}_m$ of functions, respectively, that fulfill requirements detailed for curves of type (1).

- The vectors $\mathbf{p}_{i,j} \in \mathbb{R}^3$ may also correspond to (higher order mixed) partial derivatives (such as boundary tangent vectors in both directions and twist vectors).
- In principle function systems $\mathcal{F}_n$ and $\mathcal{G}_m$ can be bases of different vector spaces of functions, however, in practice they are almost always the same.



- The *u isoparametric lines* of the surface (9) are

$$\mathbf{c}_v(u) = \sum_{i=0}^{n} \mathbf{a}_i(v) F_{n,i}(u), \; u \in [u_{\min}, u_{\max}] \tag{10}$$

for all fixed values of $v \in [v_{\min}, v_{\max}]$, where

$$\mathbf{a}_i(v) = \sum_{j=0}^{m} \mathbf{p}_{i,j} G_{m,j}(v), \; i = 0, 1, \ldots, n.$$

- This means, that for all fixed values of $v \in [v_{\min}, v_{\max}]$ the u isoparametric line (10) of the surface (9) is in fact a curve of type (1) whose control points $[\mathbf{a}_i(v)]_{i=0}^{n}$ also move along curves of type (1) as the parameter v varies in the range $[v_{\min}, v_{\max}]$ (for an example consider Fig. 5).
- Naturally, the same phenomenon also occurs in the case of the v isoparametric lines of the surface (9). Hence, this type of surfaces is called *tensor product surface*, more precisely, tensor product of curves of type (1).

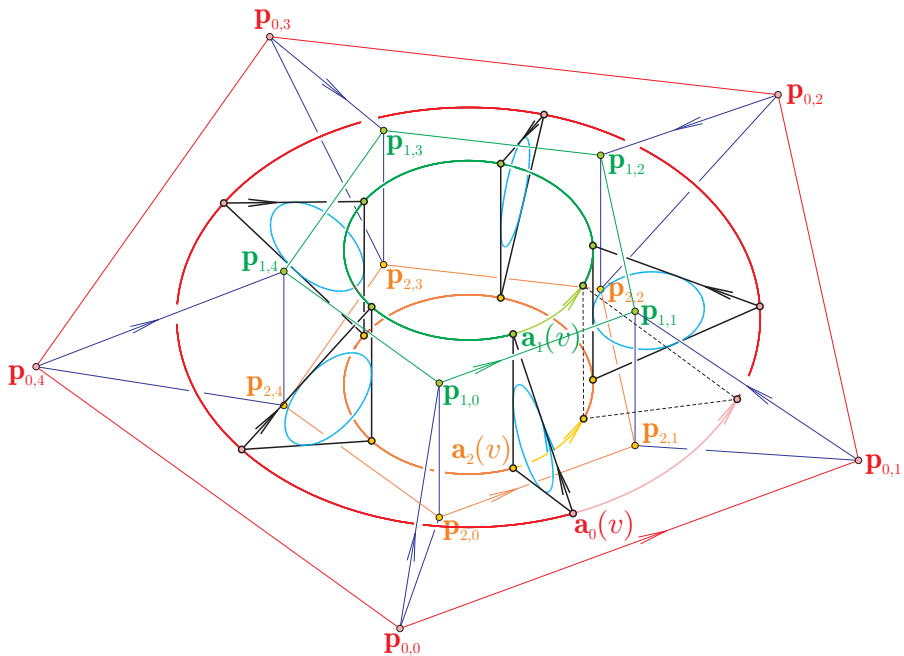


Fig. 5: Several u isoparametric lines (blue) of a cylic surface of order $(1, 2)$ along with its control net $\left[\mathbf{p}_{i,j}\right]_{i=0,j=0}^{2,4}$. As the parameter value v varies in the interval $[v_{\min}, v_{\max}] = [0, 2\pi]$, control points $\mathbf{a}_0\,(v)$, $\mathbf{a}_1\,(v)$ and $\mathbf{a}_2\,(v)$ of the first order u isoparametric lines move along quadratic cyclic curves determined by control polygons $\left[\mathbf{p}_{0,j}\right]_{j=0}^{4}$, $\left[\mathbf{p}_{1,j}\right]_{j=0}^{4}$ and $\left[\mathbf{p}_{2,j}\right]_{j=0}^{4}$, respectively.

- Observe, that formula (9) can be written into the matrix form

$$\mathbf{s}(u,v) = \begin{bmatrix} F_{n,0}(u) & F_{n,1}(u) & \cdots & F_{n,n}(u) \end{bmatrix} \begin{bmatrix} \mathbf{p}_{0,0} & \mathbf{p}_{0,1} & \cdots & \mathbf{p}_{0,m} \\ \mathbf{p}_{1,0} & \mathbf{p}_{1,1} & \cdots & \mathbf{p}_{1,m} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{p}_{n,0} & \mathbf{p}_{n,1} & \cdots & \mathbf{p}_{n,m} \end{bmatrix} \begin{bmatrix} G_{m,0}(v) \\ G_{m,1}(v) \\ \vdots \\ G_{m,m}(v) \end{bmatrix} \tag{11}$$

  for all parameter points $(u, v) \in [u_{\min}, u_{\max}] \times [v_{\min}, v_{\max}]$.

- If functions systems $\mathcal{F}_n$ and $\mathcal{G}_m$ are normalized, then the surface (9) will be in the *convex hull* of its control net.
- Moreover, the set of this type of surfaces is *closed under the affine transformations of its control points*, i.e., the surface determined by the transformed control points coincides with the transformed surface.

- By using the notations above, the *interpolation problem* by means of surfaces of type (9) can be formulated as follows.
- Consider the parameter values

$$u_{\min} \leq u_0 < u_1 < \ldots < u_n \leq u_{\max}$$

and

$$v_{\min} \leq v_0 < v_1 < \ldots < v_m \leq v_{\max},$$

and also define the net of data points

$$[\mathbf{d}_{k,\ell}]_{k=0,\ell=0}^{n,m} \in \mathcal{M}_{n+1,m+1}\left(\mathbb{R}^3\right),$$

such that the *interpolation conditions*

$$\mathbf{s}\left(u_k, v_\ell\right) = \mathbf{d}_{k,\ell}$$

are satisfied for all $k = 0, 1, \ldots, n$ and $\ell = 0, 1, \ldots, m$.

- Observe, that the surface interpolation problem can be reduced to several curve interpolation problems in directions u and v wich can be solved simultaneously.

- First, we determine the control points $[\mathbf{a}_i(v_\ell)]_{i=0}^n$ for all fixed parameter values $\{v_\ell\}_{\ell=0}^m$ such that

$$M \begin{pmatrix} F_{n,0} & F_{n,1} & \cdots & F_{n,n} \\ u_0 & u_1 & \cdots & u_n \end{pmatrix} \begin{bmatrix} \mathbf{a}_0(v_\ell) \\ \mathbf{a}_1(v_\ell) \\ \vdots \\ \mathbf{a}_n(v_\ell) \end{bmatrix} = \begin{bmatrix} \mathbf{c}_{v_\ell}(u_0) \\ \mathbf{c}_{v_\ell}(u_1) \\ \vdots \\ \mathbf{c}_{v_\ell}(u_n) \end{bmatrix} = \begin{bmatrix} \mathbf{d}_{0,\ell} \\ \mathbf{d}_{1,\ell} \\ \vdots \\ \mathbf{d}_{n,\ell} \end{bmatrix}, \qquad (12)$$

  and after that we obtain the control net $[\mathbf{p}_{i,j}]_{i=0,j=0}^{n,m}$ by solving the linear system of equations

$$M \begin{pmatrix} G_{m,0} & G_{m,1} & \cdots & G_{m,m} \\ v_0 & v_1 & \cdots & v_m \end{pmatrix} \begin{bmatrix} \mathbf{p}_{i,0} \\ \mathbf{p}_{i,1} \\ \vdots \\ \mathbf{p}_{i,m} \end{bmatrix} = \begin{bmatrix} \mathbf{a}_i(v_0) \\ \mathbf{a}_i(v_1) \\ \vdots \\ \mathbf{a}_i(v_m) \end{bmatrix} \qquad (13)$$

  for all $i = 0, 1, \ldots, n$.

- Observe, that the collocation matrices which appear in matrix equations (12) and (13) are fixed.
- Thus, once we have computed, e.g., the *LU decomposition* [Press et al., 2007] of these collocation matrices, linear systems (12) and (13) can be solved simultaneously with a parallel algorithm for all $\ell = 0, 1, \ldots, m$ and $i = 0, 1, \ldots, n$, respectively.
- Even the LU decomposition of regular matrices can also be parallelized [Kaya (2005)Kaya, Wright].

- Fig. 6 presents different types of smooth interpolating surfaces.

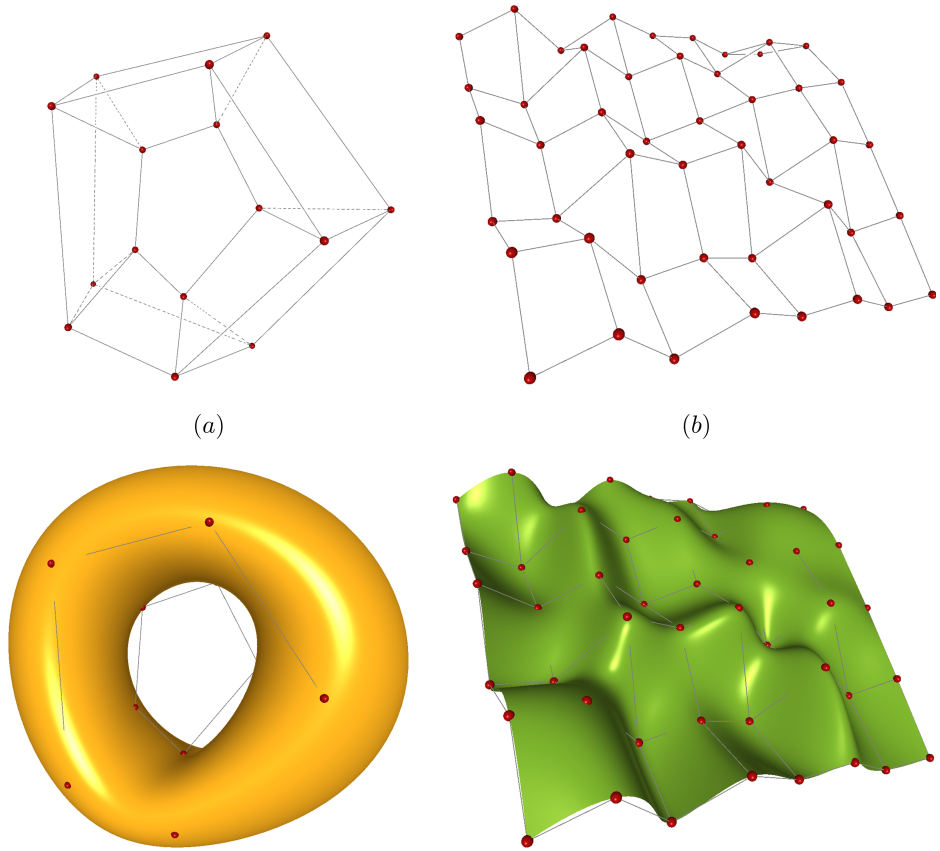


Fig. 6: Smooth interpolating closed and open surfaces along with their nets of data points. (*a*) A cyclic surface of order $(1, 2)$. (*b*) A non-uniform B-spline surface of order $(4, 4)$.

- As in the case of the curve interpolation problem, *the shape of an interpolating tensor product surface is globally controlled by the modifications of data points* $[\mathbf{d}_{k,\ell}]_{k=0,\ell=0}^{n,m}$ *or parameter values* $\{(u_k, v_\ell)\}_{k=0,\ell=0}^{n,m}$ assigned to them.
- This phenomenon is illustrated in Fig. 7 in case of which data points to be interpolated are fixed but the distribution of parameter values associated with them are different.

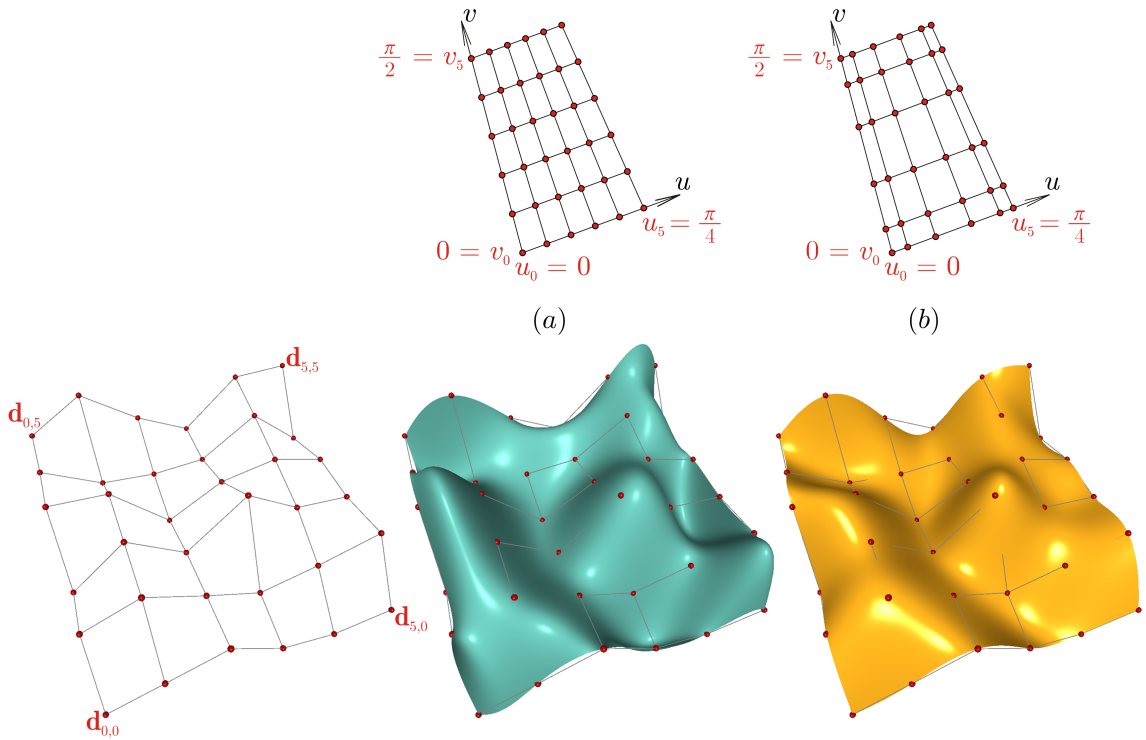


Fig. 7: Smooth algebraic trigonometric surfaces of order $(5, 5)$ defined in directions u and v by shape parameters $\alpha = \frac{\pi}{4}$ and $\beta = \frac{\pi}{2}$, respectively. Both surfaces interpolate the same net of data points $[\mathbf{d}_{k,\ell}]_{k=0,\ell=0}^{5,5}$. In cases (a) and (b) the pairwise distinct parameter values $\{(u_k, v_\ell)\}_{k=0,\ell=0}^{5,5}$ assigned to the data points follow uniform and non-uniform distribution, respectively.

- The *rational counterpart* of tensor product surface (9) can be formulated as follows.
- Consider the user defined non-negative fixed weight vectors $\mathbf{w}_u = \left[w_u^i\right]_{i=0}^n$ and $\mathbf{w}_v = \left[w_v^j\right]_{j=0}^m$ of rank 1 in directions u and v, respectively.
- Assume, that bases $\mathcal{F}_n$ and $\mathcal{G}_m$ are normalized, and define the normalized rational bases

$$\mathcal{R}_n^{\mathbf{w}u} = \left\{R_{n,i}^{\mathbf{w}u} : [u_{\min}, u_{\max}] \to \mathbb{R}\right\}_{i=0}^n$$

and

$$\mathcal{S}_m^{\mathbf{w}v} = \left\{S_{m,j}^{\mathbf{w}v} : [v_{\min}, v_{\max}] \to \mathbb{R}\right\}_{j=0}^m,$$

where

$$R_{i,n}^{\mathbf{w}u}(u) = \frac{w_u^i F_{n,i}(u)}{\sum_{k=0}^n w_u^k F_{n,k}(u)}, \quad u \in [u_{\min}, u_{\max}], \quad i = 0, 1, \ldots, n$$

and

$$S_{j,m}^{\mathbf{w}v}(v) = \frac{w_v^j G_{m,j}(v)}{\sum_{\ell=0}^m w_v^\ell G_{m,\ell}(v)}, \quad v \in [v_{\min}, v_{\max}], \quad j = 0, 1, \ldots, m.$$

- Then, given a control net $[\mathbf{p}_{i,j}]_{i=0,j=0}^{n,m}$, we can define the *rational surface*

$$\begin{aligned}\mathbf{s}^{\mathbf{w}_u,\mathbf{w}_v}(u,v) &= \sum_{i=0}^{n}\sum_{j=0}^{m}\mathbf{p}_{i,j}R_{n,i}^{\mathbf{w}_u}(u)\,S_{m,j}^{\mathbf{w}_v}(v)\\ &= \frac{\sum_{i=0}^{n}\sum_{j=0}^{m}w_u^i w_v^j\mathbf{p}_{i,j}F_{n,i}(u)\,G_{m,j}(v)}{\sum_{k=0}^{n}\sum_{\ell=0}^{m}w_u^k w_v^\ell F_{n,k}(u)\,G_{m,\ell}(v)},\ (u,v)\in[u_{\min},u_{\max}]\times[v_{\min},v_{\max}].\end{aligned} \tag{14}$$

- Note, that in formula (14) the products $w_u^i w_v^j$ of weights can also be associated with control points $\mathbf{p}_{i,j}$ $(i=0,1,\ldots,n,\ j=0,1,\ldots,m)$.
- This observation leads to the definition of a *more flexible tool for rational surface* modeling:
  - by considering a non-negative weight matrix

$$W_{n,m}=[w_{i,j}]_{i=0,j=0}^{n,m}\in\mathcal{M}_{n+1,m+1}(\mathbb{R}) \tag{15}$$

  of rank at least 1, we can generalize the rational surface (14) in the form

$$\mathbf{s}^{W_{n,m}}(u,v)=\frac{\sum_{i=0}^{n}\sum_{j=0}^{m}w_{i,j}\mathbf{p}_{i,j}F_{n,i}(u)\,G_{m,j}(v)}{\sum_{k=0}^{n}\sum_{\ell=0}^{m}w_{k,\ell}F_{n,k}(u)\,G_{m,\ell}(v)},\ (u,v)\in[u_{\min},u_{\max}]\times[v_{\min},v_{\max}]. \tag{16}$$

- Thus, similarly to the case of rational curves, by altering the weight of a selected control point we can either increase or decrease its effect on the shape of the rational surface (16).
- The rational surface (16) can also be considered as the *central projection* of the surface

$$\mathbf{s}_{\wp}^{W_{n,m}}(u,v)=\sum_{i=0}^{n}\sum_{j=0}^{m}\begin{bmatrix} w_{i,j}\mathbf{p}_{i,j} \\ w_{i,j}\end{bmatrix} F_{n,i}(u)\,G_{m,j}(v)\,,\ (u,v)\in[u_{\min},u_{\max}]\times[v_{\min},v_{\max}] \qquad (17)$$

  in the 4-dimensional Euclidean vector space $\mathbb{R}^4$ from its origin onto the 3-dimensional hyperplane $x_3=1$ (provided that the coordinates of $\mathbb{R}^4$ are labeled by x_0,x_1,x_2,x_3).
- The surface (17) is called the *pre-image* of the rational surface (16), while the vector space $\mathbb{R}^4$ is its *pre-image space*.
- Thus, the set of rational surfaces of type (16) is *closed under the projective transformations* of its control points, i.e., the surface determined by the transformed control points coincides with the transformed surface.

- Fig. 8 shows several NURBS surfaces. Similar to the NURBS curves, in our days, NURBS surfaces form the de facto standard surface modeling tool in CAGD due to their advantageous geometric properties such as the projective (and implicitly affine) invariance, local convex hull and local controllability.

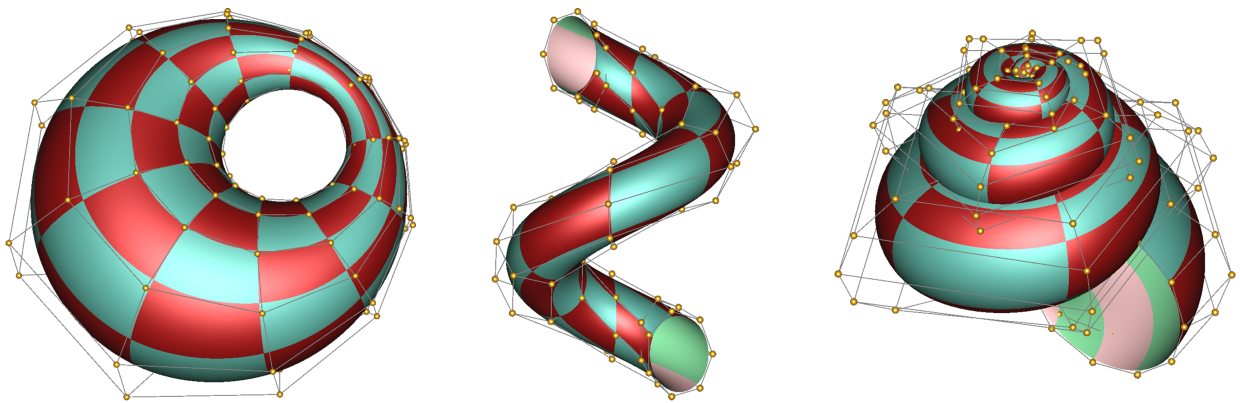


Fig. 8: Non-uniform rational B-spline surfaces of order $(4, 4)$.

- The control point based exact description of the ring Dupin cyclide shown in Fig. 9(b) is based on the rational counter part of the cyclic basis functions introduced in [Juhász, Róth, 2010, Róth et al., 2009].

(a) (b)

Fig. 9: Control point based exact description *(a)* of a Zhukovsky airfoil profile and *(b)* of a ring Dupin cyclid by means of rational cyclic curves and surfaces.

Color styles

- keywords, built-in types, enumerations, constants and namespaces of C++
- keywords, built-in types, enumerations, constants and functions of OpenGL
- our types, constants, enumerations and namespaces
- comments

Abstract tensor product surfaces

Implementation details: Class TensorProductSurface3

Description

- Class TensorProductSurface3 provides some functionality in order to evaluate and render surfaces of the form

$$\begin{cases} \mathbf{s} : [u_{\min}, u_{\max}] \times [v_{\min}, v_{\max}] \to \mathbb{R}^3 \\ \\ \mathbf{s}(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} \mathbf{p}_{i,j} F_{n,i}(u) G_{m,j}(v), \end{cases}$$

where the vector matrix

$$\left[\mathbf{p}_{i,j}\right]_{i=0,j=0}^{n,m} \in \mathcal{M}_{n+1,m+1}(\mathbb{R}^3)$$

usually represents a control net, while function systems

$$\mathcal{F}_n = \{F_{n,i} : [u_{\min}, u_{\max}] \to \mathbb{R}\}_{i=0}^{n}$$

and

$$\mathcal{G}_m = \{G_{m,j} : [v_{\min}, v_{\max}] \to \mathbb{R}\}_{j=0}^{m}$$

are linearly independent and, in general, normalized.

Description – continued

- Moreover, this class also solves the surface interpolation problem

$$\mathbf{s}(u_k, v_\ell) = \mathbf{d}_{k,\ell},\ k = 0, 1, \ldots, n,\ \ell = 0, 1, \ldots, m,$$

where

$$\left[\mathbf{d}_{k,\ell}\right]_{i=0,j=0}^{n,m} \in \mathcal{M}_{n+1,m+1}(\mathbb{R}^3)$$

is a matrix of data points, while

$$u_{\min} \leq u_0 < u_1 < \cdots < u_n \leq u_{\max}$$

and

$$v_{\min} \leq v_0 < v_1 < \cdots < v_m \leq v_{\max}$$

are strictly increasing knot vectors in u- and v-directions, respectively.

- The class does not know in advance the bases $\mathcal{F}_n$ and $\mathcal{G}_m$ nor the efficient evaluation methods of surface properties. Thus, the class has pure virtual methods which must be implemented by the user in a derived class.

Abstract tensor product surfaces – header file, part I

Implementation details: TensorProductSurfaces3.h

```
1  #pragma once

2  #include "DCoordinates3.h"
3  #include <GL/glew.h>
4  #include <iostream>
5  #include "Matrices.h"
6  #include "GenericCurves3.h"
7  #include "TriangulatedMeshes3.h"
8  #include <vector>

9  namespace cagd
10 {
11     class TensorProductSurface3
12     {
13     public:
14         // a nested class the stores the zeroth and higher order partial derivatives associated with a
15         // surface point s(u, v), i.e.,

                  s(u, v)

               ∂/∂u s(u, v)    ∂/∂v s(u, v)
16         //  ∂²/∂u² s(u, v)   ∂²/∂u∂v s(u, v)   ∂²/∂v² s(u, v)
               .               .                 .            .
               .               .                 .              .
               .               .                 .                .

17         class PartialDerivatives: public TriangularMatrix<DCoordinate3>
18         {
19         public:
20             PartialDerivatives(GLuint maximum_order_of_partial_derivatives = 1);

21             // homework: initializes all partial derivatives to the origin
22             GLvoid LoadNullVectors();
23         };
```

Abstract tensor product surfaces – header file, part II

Implementation details: TensorProductSurfaces3.h

```
24      protected:
25          GLboolean               _u_closed, _v_closed; // is the surface closed in direction u or v?
26          GLuint                  _vbo_data;            // vertex buffer object of the control net
27          GLdouble                _u_min, _u_max;       // definition domain in direction u
28          GLdouble                _v_min, _v_max;       // definition domain in direction v
29          Matrix<DCoordinate3> _data;                   // the control net (usually stores position vectors)

30      public:
31          // homework: special constructor
32          TensorProductSurface3(
33                  GLdouble u_min, GLdouble u_max,
34                  GLdouble v_min, GLdouble v_max,
35                  GLuint row_count = 4, GLuint column_count = 4,
36                  GLboolean u_closed = GL_FALSE, GLboolean v_closed = GL_FALSE);

37          // homework: copy constructor
38          TensorProductSurface3(const TensorProductSurface3& surface);

39          // homework: assignment operator
40          TensorProductSurface3& operator =(const TensorProductSurface3& surface);

41          // homework: set/get the definition domain of the surface
42          GLvoid SetUInterval(GLdouble u_min, GLdouble u_max);
43          GLvoid SetVInterval(GLdouble v_min, GLdouble v_max);

44          GLvoid GetUInterval(GLdouble& u_min, GLdouble& u_max) const;
45          GLvoid GetVInterval(GLdouble& v_min, GLdouble& v_max) const;

46          // homework: set coordinates of a selected data point
47          GLboolean SetData(GLuint row, GLuint column, GLdouble x, GLdouble y, GLdouble z);
48          GLboolean SetData(GLuint row, GLuint column, const DCoordinate3& point);

49          // homework: get coordinates of a selected data point
50          GLboolean GetData(GLuint row, GLuint column, GLdouble& x, GLdouble& y, GLdouble& z) const;
51          GLboolean GetData(GLuint row, GLuint column, DCoordinate3& point) const;
```

Abstract tensor product surfaces – header file, part III

Implementation details: TensorProductSurfaces3.h

```
52          // homework: get data by value
53          DCoordinate3 operator ()(GLuint row, GLuint column) const;

54          // homework: get data by reference
55          DCoordinate3& operator ()(GLuint row, GLuint column);

56          // blending function values in u- and v-directions (pure virtual methods)
57          virtual GLboolean UBlendingFunctionValues(
58                  GLdouble u_knot, RowMatrix<GLdouble>& blending_values) const = 0;

59          virtual GLboolean VBlendingFunctionValues(
60                  GLdouble v_knot, RowMatrix<GLdouble>& blending_values) const = 0;

61          // calculates the point and higher order (mixed) partial derivatives of the
62          // tensor product surface
63          //
64          // s(u, v) = sum_{i=0}^{n} sum_{j=0}^{m} p_{i,j} F_{n,i}(u) G_{m,j}(v),
65          //
66          // where n+1 and m+1 denote the row count and column count of the matrix _data, respectively,
67          // while (u, v) in [u_min, u_max] x [v_min, v_max] (pure virtual method)
68          virtual GLboolean CalculatePartialDerivatives(
69                  GLuint maximum_order_of_partial_derivatives,
70                  GLdouble u, GLdouble v, PartialDerivatives& pd) const = 0;

71          // generates a triangulated mesh that approximates the shape of the surface above
72          virtual TriangulatedMesh3* GenerateImage(
73                  GLuint u_div_point_count, GLuint v_div_point_count,
74                  GLenum usage_flag = GL_STATIC_DRAW) const;
75
```

Abstract tensor product surfaces – header file, part IV

Implementation details: TensorProductSurfaces3.h

```
76          // ensures interpolation, i.e., updates the control net [p_{i,j}]_{i=0,j=0}^{n,m} stored by
77          // the matrix _data such that interpolation conditions s(u_k, v_ℓ) = d_{k,ℓ} hold for
78          // all k = 0, 1, ..., n and ℓ = 0, 1, ..., m
79          GLboolean UpdateDataForInterpolation(
80                  const RowMatrix<GLdouble>& u_knot_vector, const ColumnMatrix<GLdouble>& v_knot_vector,
81                  Matrix<DCoordinate3>& data_points_to_interpolate);

82          // homework: VBO handling methods
83          GLvoid     DeleteVertexBufferObjectsOfData();
84          GLboolean  RenderData(GLenum render_mode = GL_LINE_STRIP) const;
85          GLboolean  UpdateVertexBufferObjectsOfData(GLenum usage_flag = GL_STATIC_DRAW);

86          // homework: generate u-directional isoparametric lines
87          RowMatrix<GenericCurve3*>* GenerateUIsoparametricLines(GLuint iso_line_count,
88                                                                 GLuint maximum_order_of_derivatives,
89                                                                 GLuint div_point_count,
90                                                                 GLenum usage_flag = GL_STATIC_DRAW) const;

91          // homework: generate v-directional isoparametric lines
92          RowMatrix<GenericCurve3*>* GenerateVIsoparametricLines(GLuint iso_line_count,
93                                                                 GLuint maximum_order_of_derivatives,
94                                                                 GLuint div_point_count,
95                                                                 GLenum usage_flag = GL_STATIC_DRAW) const;

96          // homework: destructor
97          virtual ~TensorProductSurface3();
98      };
99  }
```

Abstract tensor product surfaces – source file, part I

Implementation details: TensorProductSurfaces3.cpp

```
1  #include "TensorProductSurfaces3.h"
2  #include "RealSquareMatrices.h"
3  #include <algorithm>

4  using namespace cagd;
5  using namespace std;

6  TriangulatedMesh3* TensorProductSurface3::GenerateImage(
7          GLuint u_div_point_count, GLuint v_div_point_count,
8          GLenum usage_flag) const
9  {
10     if (u_div_point_count <= 1 || v_div_point_count <= 1)
11         return GL_FALSE;

12     // calculating number of vertices, unit normal vectors and texture coordinates
13     GLuint vertex_count = u_div_point_count * v_div_point_count;

14     // calculating number of triangular faces
15     GLuint face_count = 2 * (u_div_point_count - 1) * (v_div_point_count - 1);

16     TriangulatedMesh3 *result = nullptr;
17     result = new TriangulatedMesh3(vertex_count, face_count, usage_flag);

18     if (!result)
19         return nullptr;

20     // u- and v-directional steps of the uniform subdivision grid in the definition domain
21     GLdouble du = (_u_max - _u_min) / (u_div_point_count - 1);
22     GLdouble dv = (_v_max - _v_min) / (v_div_point_count - 1);

23     // s- and t-directional steps of the uniform subdivision grid in the unit square
24     GLfloat ds = 1.0f / (u_div_point_count - 1);
25     GLfloat dt = 1.0f / (v_div_point_count - 1);
```

Abstract tensor product surfaces – source file, part II

Implementation details: TensorProductSurfaces3.cpp

```
26      // for face indexing
27      GLuint current_face = 0;

28      // partial derivatives up to order 1
29      PartialDerivatives pd;

30      for (GLuint i = 0; i < u_div_point_count; ++i)
31      {
32          GLdouble u = min(_u_min + i * du, _u_max);
33          GLfloat  s = min(i * ds, 1.0f);
34          for (GLuint j = 0; j < v_div_point_count; ++j)
35          {
36              GLdouble v = min(_v_min + j * dv, _v_max);
37              GLfloat  t = min(j * dt, 1.0f);
38              /*
39                  3--2
40                  |/|
41                  0--1
42              */
43              GLuint index[4];

44              index[0] = i * v_div_point_count + j;
45              index[1] = index[0] + 1;
46              index[2] = index[1] + v_div_point_count;
47              index[3] = index[2] - 1;

48              // calculating all needed surface data (i.e., zeroth and first order partial derivatives)
49              CalculatePartialDerivatives(1, u, v, pd);

50              // surface point
51              (*result)._vertex[index[0]] = pd(0, 0);  // surface point

52              // unit surface normal
53              (*result)._normal[index[0]] =  pd(1, 0); // cross product of u- and
54              (*result)._normal[index[0]] ^= pd(1, 1); // v-directional partial derivatives
55              (*result)._normal[index[0]].normalize(); // the obtained normal vector has to normalized
```

Abstract tensor product surfaces – source file, part III

Implementation details: TensorProductSurfaces3.cpp

```
56              // texture coordinates
57              (*result)._tex[index[0]].s() = s;
58              (*result)._tex[index[0]].t() = t;

59              // faces that store connectivity information
60              if (i < u_div_point_count - 1 && j < v_div_point_count - 1)
61              {
62                  (*result)._face[current_face][0] = index[0];
63                  (*result)._face[current_face][1] = index[1];
64                  (*result)._face[current_face][2] = index[2];
65                  ++current_face;

66                  (*result)._face[current_face][0] = index[0];
67                  (*result)._face[current_face][1] = index[2];
68                  (*result)._face[current_face][2] = index[3];
69                  ++current_face;
70              }
71          }
72      }

73      return result;
74  }

75  // ensures interpolation, i.e., s(u_k, v_l) = d_{k,l}, ∀k = 0, 1, ..., n, l = 0, 1, ..., m
76  GLboolean TensorProductSurface3::UpdateDataForInterpolation(
77          const RowMatrix<GLdouble>& u_knot_vector, const ColumnMatrix<GLdouble>& v_knot_vector,
78          Matrix<DCoordinate3>& data_points_to_interpolate)
79  {
80      GLuint row_count = _data.GetRowCount();
81      if (!row_count)
82          return GL_FALSE;

83      GLuint column_count = _data.GetColumnCount();
```

Abstract tensor product surfaces – source file, part IV

Implementation details: TensorProductSurfaces3.cpp

```
84      if (!column_count)
85          return GL_FALSE;
86      if (u_knot_vector.GetColumnCount() != row_count ||
87          v_knot_vector.GetRowCount() != column_count ||
88          data_points_to_interpolate.GetRowCount() != row_count ||
89          data_points_to_interpolate.GetColumnCount() != column_count)
90          return GL_FALSE;
91      // 1: calculate the u-collocation matrix and perfom LU-decomposition on it
92      RowMatrix<GLdouble> u_blending_values;
93      RealSquareMatrix u_collocation_matrix(row_count);
94      for (GLuint k = 0; k < row_count; ++k)
95      {
96          if (!UBlendingFunctionValues(u_knot_vector(k), u_blending_values))
97              return GL_FALSE;
98          u_collocation_matrix.SetRow(k, u_blending_values);
99      }
100     if (!u_collocation_matrix.PerformLUDecomposition())
101         return GL_FALSE;
102     // 2: calculate the v-collocation matrix and perform LU-decomposition on it
103     RowMatrix<GLdouble> v_blending_values;
104     RealSquareMatrix v_collocation_matrix(column_count);
105     for (GLuint l = 0; l < column_count; ++l)
106     {
107         if (!VBlendingFunctionValues(v_knot_vector(l), v_blending_values))
108             return GL_FALSE;
109         v_collocation_matrix.SetRow(l, v_blending_values);
110     }
111
```

Abstract tensor product surfaces – source file, part V

Implementation details: TensorProductSurfaces3.cpp

```
112     if (!v_collocation_matrix.PerformLUDecomposition())
113             return GL_FALSE;

114     // 3:   for all fixed ℓ ∈ {0, 1, ..., m} determine control points
115     //
116     //       a_i(v_ℓ) = Σ_{j=0}^{m} p_{i,j} G_{m,j}(v_ℓ), i = 0, 1, ..., n
117     //
118     //       such that
119     //
120     //       c_{v_ℓ}(u_k) = Σ_{i=0}^{n} a_i(v_ℓ) F_{n,i}(u_k) = d_{k,ℓ},
121     //
122     //       for all k = 0, 1, ..., n
123     Matrix<DCoordinate3> a(row_count, column_count);
124     if (!u_collocation_matrix.SolveLinearSystem(data_points_to_interpolate, a))
125         return GL_FALSE;

126     // 4:   for all fixed i ∈ {0, 1, ..., n} determine control points
127     //
128     //       p_{i,j}, j = 0, 1, ..., m
129     //
130     //       such that
131     //
132     //       a_i(v_ℓ) = Σ_{j=0}^{m} p_{i,j} G_{m,j}(v_ℓ)
133     //
134     //       for all ℓ = 0, 1, ..., m
135     if (!v_collocation_matrix.SolveLinearSystem(a, _data, GL_FALSE))
136         return GL_FALSE;

137     return GL_TRUE;
138 }
```

Example 1 (Trigonometric Bernstein-like basis functions)

- Let $\alpha \in (0, \pi)$ be an arbitrarily fixed shape parameter and consider the function system

$$\mathcal{A}_{2n}^{\alpha} = \left\{ A_{2n,i}^{\alpha}(u) = c_{2n,i}^{\alpha} \sin^{2n-i}\left(\frac{\alpha - u}{2}\right) \sin^{i}\left(\frac{u}{2}\right) : u \in [0, \alpha] \right\}_{i=0}^{2n}, \tag{18}$$

where the normalizing positive coefficients

$$c_{2n,i}^{\alpha} = \frac{1}{\sin^{2n}\left(\frac{\alpha}{2}\right)} \sum_{r=0}^{\left\lfloor \frac{i}{2} \right\rfloor} \binom{n}{i-r} \binom{i-r}{r} \left(2\cos\left(\frac{\alpha}{2}\right)\right)^{i-2r}, \quad i = 0, 1, \ldots, 2n$$

fulfill the symmetry

$$c_{2n,i}^{\alpha} = c_{2n,2n-i}^{\alpha}, \quad i = 0, 1, \ldots, n. \tag{19}$$

- Functions (18) form the basis of the vector space

$$\mathbb{T}_{2n}^{\alpha} = \langle 1, \cos(u), \sin(u), \ldots, \cos(nu), \sin(nu) : u \in [0, \alpha] \rangle$$

of truncated Fourier series of finite order n.

Basis functions (18) fulfill the following properties:

- partition of unity:

$$\sum_{i=0}^{2n} A_{2n,i}^{\alpha}(u) \equiv 1, \forall u \in [0, \alpha], \tag{20}$$

- endpoint interpolation property:

$$A_{2n,i}^{\alpha}(0) = \begin{cases} 1, & i = 0, \\ 0, & \text{otherwise}, \end{cases} \tag{21}$$

$$A_{2n,i}^{\alpha}(\alpha) = \begin{cases} 1, & i = 2n, \\ 0, & \text{otherwise}. \end{cases} \tag{22}$$

- symmetry:

$$A_{2n,i}^{\alpha}(\alpha - u) = A_{2n,2n-i}^{\alpha}(u), \ \forall u \in [0, \alpha], \ i = 0, 1, \ldots, n, \tag{23}$$

- recursive property of derivatives:

$$\frac{\mathrm{d}}{\mathrm{d}u} A_{2n,i}^{\alpha}(u) \tag{24}$$

$$= \begin{cases} -\frac{n}{\tan\frac{\alpha}{2}} A_{2n,0}^{\alpha}(u) - \frac{c_{2n,0}^{\alpha}}{c_{2n,1}^{\alpha}} \frac{n}{\sin\frac{\alpha}{2}} A_{2n,1}^{\alpha}(u), & i = 0 \\ \frac{c_{2n,i}^{\alpha}}{c_{2n,i-1}^{\alpha}} \frac{i}{2\sin\frac{\alpha}{2}} A_{2n,i-1}^{\alpha}(u) - \frac{n-i}{\tan\frac{\alpha}{2}} A_{2n,i}^{\alpha}(u) - \frac{c_{2n,i}^{\alpha}}{c_{2n,i+1}^{\alpha}} \frac{2n-i}{2\sin\frac{\alpha}{2}} A_{2n,i+1}^{\alpha}(u), & i = \overline{1, 2n-1}, \\ \frac{c_{2n,2n}^{\alpha}}{c_{2n,2n-1}^{\alpha}} \frac{n}{\sin\frac{\alpha}{2}} A_{2n,2n-1}^{\alpha}(u) + \frac{n}{\tan\frac{\alpha}{2}} A_{2n,2n}^{\alpha}(u), & i = 2n, \end{cases}$$

Definition (Trigonometric Bernstein-like surfaces)

The tensor product

$$\mathbf{s}_{n,m}^{\alpha,\beta}(u,v)=\sum_{i=0}^{2n}\sum_{j=0}^{2m}\mathbf{p}_{i,j}A_{2n,i}^{\alpha}(u)\,A_{2m,j}^{\beta}(v),\quad (u,v)\in[0,\alpha]\times[0,\beta]$$

defines a trigonometric Bernstein-like surface of order (n,m) and of shape parameters $(\alpha,\beta)\in(0,\pi)\times(0,\pi)$.

Implementation details: header file, part – I

Trigonometric/TrigonometricBernsteinSurfaces3.h

```
1  #pragma once

2  #include "../Core/Matrices.h"
3  #include "../Core/RealSquareMatrices.h"
4  #include "../Core/TensorProductSurfaces3.h"

5  namespace cagd
6  {
7      class TrigonometricBernsteinSurface3: public TensorProductSurface3
8      {
9      protected:

10         // shape parameters in directions u and v, respectively
11         GLdouble             _alpha, _beta;

12         // orders of basis functions in directions u and v, respectively
13         GLuint               _n, _m;

14         // normalizing coefficients in directions u and v, respectively
15         RowMatrix<GLdouble> _u_c, _v_c;

16         // binomial coefficients
17         TriangularMatrix<GLdouble> _bc;

18         // auxiliar protected methods
19         GLvoid               _CalculateBinomialCoefficients(GLuint order, TriangularMatrix<GLdouble> &bc);
20         GLboolean            _CalculateNormalizingCoefficients(GLuint order, GLdouble alpha,
21                                                                RowMatrix<GLdouble> &c);

22     public:
23         // special constructor
24         TrigonometricBernsteinSurface3(GLdouble alpha, GLuint n, GLdouble beta, GLuint m);

25         // inherited pure virtual abstract methods that must be redeclared and defined
26         GLboolean UBlendingFunctionValues(GLdouble u, RowMatrix<GLdouble>& values) const;
```

Implementation details: header file, part – II

Trigonometric/TrigonometricBernsteinSurfaces3.h

```
27          GLboolean VBlendingFunctionValues(GLdouble v, RowMatrix<GLdouble>& values) const;
28          GLboolean CalculatePartialDerivatives(GLuint maximum_order_of_partial_derivatives,
29                                                GLdouble u, GLdouble v, PartialDerivatives& pd) const;

30          //...

31      };
32  }
```

Implementation details: source file, part – I

Trigonometric/TrigonometricBernsteinSurfaces3.cpp

```
1  #include "TrigonometricBernsteinSurfaces3.h"
2  #include <cmath>
3  using namespace std;
4  namespace cagd
5  {
6      // calculates binomial coefficients from 0 to up to the given order
7      GLvoid TrigonometricBernsteinSurface3::_CalculateBinomialCoefficients(
8              GLuint order, TriangularMatrix<GLdouble> &bc)
9      {
10         bc.ResizeRows(order + 1);
11         bc(0, 0) = 1.0;
12         for (GLuint r = 1; r <= order; ++r)
13         {
14             bc(r, 0) = 1.0;
15             bc(r, r) = 1.0;
16             for (GLuint i = 1; i <= r / 2; ++i)
17             {
18                 bc(r, i) = bc(r-1, i - 1) + bc(r - 1, i);
19                 bc(r, r - i) = bc(r, i);
20             }
21         }
22     }
23
```

Implementation details: source file, part – II

Trigonometric/TrigonometricBernsteinSurfaces3.cpp

```
24      // calculates the normalizing coefficients {c^α_{2n,i}}_{i=0}^{2n}
25      GLboolean TrigonometricBernsteinSurface3::_CalculateNormalizingCoefficients(
26              GLuint order, GLdouble alpha, RowMatrix<GLdouble> &c)
27      {
28          if (!order)
29          {
30              c.ResizeColumns(0);
31              return GL_FALSE;
32          }
33          GLuint size = 2 * order + 1;
34          c.ResizeColumns(size);
35          GLdouble sa = pow(sin(alpha / 2.0), (GLint)(2 * order));
36          GLdouble ca = 2.0 * cos(alpha / 2.0);
37          // c^α_{2n,i} = 1/sin^{2n}(α/2) Σ_{r=0}^{⌊i/2⌋} C(n, i-r) C(i-r, r) (2 cos(α/2))^{i-2r}, i = 0, 1, ..., 2n
38          // c^α_{2n,i} = c^α_{2n,2n-i}, i = 0, 1, ..., n
39          for (GLuint i = 0; i <= order; ++i)
40          {
41              c[i] = 0.0;
42              for (GLuint r = 0; r <= i/2; ++r)
43              {
44                  c[i] += _bc(order, i - r) * _bc(i - r, r) * pow(ca, (GLint)(i - 2 * r));
45              }
46              c[i] /= sa;
47              c[size - 1 - i] = c[i];
48          }
49          return GL_TRUE;
50      }
```

Implementation details: source file, part – III

Trigonometric/TrigonometricBernsteinSurfaces3.cpp

```
51      // special constructor
52      TrigonometricBernsteinSurface3::TrigonometricBernsteinSurface3(
53              GLdouble alpha, GLuint n, GLdouble beta, GLuint m):
54              TensorProductSurface3(0.0, alpha, 0.0, beta, 2 * n + 1, 2 * m + 1),
55              _alpha(alpha), _beta(beta),
56              _n(n), _m(m)
57      {
58          GLuint max_order = (_n > _m ? _n : _m);
59          _CalculateBinomialCoefficients(max_order, _bc);
60          _CalculateNormalizingCoefficients(_n, _alpha, _u_c);
61          _CalculateNormalizingCoefficients(_m, _beta, _v_c);
62      }
63      // calculates blending functions {A^α_{2n,i}(u) = c^α_{2n,i} sin^{2n-i}((α-u)/2) sin^i(u/2) : u ∈ [0, α]}_{i=0}^{2n}
64      GLboolean TrigonometricBernsteinSurface3::UBlendingFunctionValues(
65              GLdouble u, RowMatrix<GLdouble>& values) const
66      {
67          if (u < 0.0 || u > _alpha)
68          {
69              values.ResizeColumns(0);
70              return GL_FALSE;
71          }
72          GLuint size = 2 * _n + 1;
73          values.ResizeColumns(size);
74          for (GLuint i = 0; i < size; ++i)
75          {
76              values[i] = 0.0;
77          }
```

Implementation details: source file, part – IV

Trigonometric/TrigonometricBernsteinSurfaces3.cpp

```
78          if (u == 0.0)
79          {
80              values[0] = 1.0;
81          }
82          else
83          {
84              if (u == _alpha)
85              {
86                  values[size - 1] = 1.0;
87              }
88              else
89              {
90                  GLdouble sau = sin((_alpha - u) / 2.0), su = sin(u / 2.0);
91                  if (u < _alpha / 2.0)
92                  {
93                      GLdouble factor    = su / sau;
94                      GLdouble sau_order = pow(sau, static_cast<GLint>(size - 1));
95                      values[0] = sau_order;
96                      for (GLint i = 1; i < size; ++i)
97                      {
98                          values[i] = values[i - 1] * factor;
99                      }
100                 }
101                 else
102                 {
103                     GLdouble factor   = sau / su;
104                     GLdouble su_order = pow(su, static_cast<GLint>(size - 1));
105                     values[size - 1] = su_order;
106                     for (GLint i = size - 2; i >= 0; --i)
107                     {
108                         values[i] = values[i + 1] * factor;
```

Implementation details: source file, part – V

Trigonometric/TrigonometricBernsteinSurfaces3.cpp

```
109                     }
110                 }

111                 for (GLuint i = 0; i < size; ++i)
112                 {
113                     values[i] *= _u_c[i];
114                 }
115             }
116         }

117         return GL_TRUE;
118     }

119     // calculates blending functions {A^β_{2m,j}(v) = c^β_{2m,j} sin^{2m-j}((β-v)/2) sin^j(v/2) : v ∈ [0, β]}_{j=0}^{2m}
120     GLboolean TrigonometricBernsteinSurface3::VBlendingFunctionValues(
121             GLdouble v, RowMatrix<GLdouble>& values) const
122     {
123         if (v < 0.0 || v > _beta)
124         {
125             values.ResizeColumns(0);
126             return GL_FALSE;
127         }

128         GLuint size = 2 * _m + 1;

129         values.ResizeColumns(size);

130         for (GLuint j = 0; j < size; ++j)
131         {
132             values[j] = 0.0;
133         }


134         if (v == 0.0)
```

Implementation details: source file, part – VI

Trigonometric/TrigonometricBernsteinSurfaces3.cpp

```
135         {
136             values[0] = 1.0;
137         }
138         else
139         {
140             if (v == _beta)
141             {
142                 values[size - 1] = 1.0;
143             }
144             else
145             {
146                 GLdouble sbv = sin((_beta - v) / 2.0), sv = sin(v / 2.0);
147                 if (v < _beta / 2.0)
148                 {
149                     GLdouble factor    = sv / sbv;
150                     GLdouble sbv_order = pow(sbv, static_cast<GLint>(size - 1));
151                     values[0] = sbv_order;
152                     for (GLint j = 1; j < size; ++j)
153                     {
154                         values[j] = values[j - 1] * factor;
155                     }
156                 }
157                 else
158                 {
159                     GLdouble factor   = sbv / sv;
160                     GLdouble sv_order = pow(sv, static_cast<GLint>(size - 1));
161                     values[size - 1] = sv_order;
162                     for (GLint j = size - 2; j >= 0; --j)
163                     {
164                         values[j] = values[j + 1] * factor;
165                     }
```

Implementation details: source file, part – VII

Trigonometric/TrigonometricBernsteinSurfaces3.cpp

```
166                 }
167                 for (GLuint j = 0; j < size; ++j)
168                 {
169                     values[j] *= _v_c[j];
170                 }
171             }
172         }
173         return GL_TRUE;
174     }
175     // Calculates the surface point s_{n,m}^{α,β}(u,v) and
176     // the higher order partial derivatives ∂^d/(∂u^{d-r} ∂v^r) s_{n,m}^{α,β}(u,v), r = 0,1,...,d, d ≥ 1.
177     // The first and higher order derivatives of the blending functions are calculated
178     // by using the recursive formula (24)
179     GLboolean TrigonometricBernsteinSurface3::CalculatePartialDerivatives(
180             GLuint maximum_order_of_partial_derivatives,
181             GLdouble u, GLdouble v, PartialDerivatives& pd) const
182     {
183         if (u < _u_min || u > _u_max || v < _v_min || v > _v_max)
184         {
185             pd.ResizeRows(0);
186             return GL_FALSE;
187         }
188         pd.ResizeRows(maximum_order_of_partial_derivatives + 1);
189         pd.LoadNullVectors();
190         GLuint u_size = 2 * _n + 1, v_size = 2 * _m + 1;
191         RowMatrix<GLdouble> Au, Av;
192         if (!UBlendingFunctionValues(u, Au) || !VBlendingFunctionValues(v, Av))
```

Implementation details: source file, part – VIII

Trigonometric/TrigonometricBernsteinSurfaces3.cpp

```
193         {
194             pd.ResizeRows(0);
195             return GL_FALSE;
196         }

197         GLdouble sua = 2.0 * sin(_alpha / 2.0), tua = tan(_alpha / 2.0);
198         GLdouble sva = 2.0 * sin(_beta / 2.0), tva = tan(_beta / 2.0);

199         Matrix<GLdouble> dAu(maximum_order_of_partial_derivatives + 1, u_size);
200         Matrix<GLdouble> dAv(maximum_order_of_partial_derivatives + 1, v_size);

201         dAu.SetRow(0, Au);
202         dAv.SetRow(0, Av);
203
```

$$\frac{\mathrm{d}}{\mathrm{d}u} A^{\alpha}_{2n,i}(u) = \begin{cases} -\frac{n}{\tan\frac{\alpha}{2}} A^{\alpha}_{2n,0}(u) - \frac{c^{\alpha}_{2n,0}}{c^{\alpha}_{2n,1}} \frac{n}{\sin\frac{\alpha}{2}} A^{\alpha}_{2n,1}(u), & i = 0 \\ \frac{c^{\alpha}_{2n,i}}{c^{\alpha}_{2n,i-1}} \frac{i}{2\sin\frac{\alpha}{2}} A^{\alpha}_{2n,i-1}(u) - \frac{n-i}{\tan\frac{\alpha}{2}} A^{\alpha}_{2n,i}(u) - \frac{c^{\alpha}_{2n,i}}{c^{\alpha}_{2n,i+1}} \frac{2n-i}{2\sin\frac{\alpha}{2}} A^{\alpha}_{2n,i+1}(u), & i = \overline{1, 2n-1}, \\ \frac{c^{\alpha}_{2n,2n}}{c^{\alpha}_{2n,2n-1}} \frac{n}{\sin\frac{\alpha}{2}} A^{\alpha}_{2n,2n-1}(u) + \frac{n}{\tan\frac{\alpha}{2}} A^{\alpha}_{2n,2n}(u), & i = 2n, \end{cases}$$

```
204         for (GLuint d = 1; d <= maximum_order_of_partial_derivatives; d++)
205         {
206             for (GLuint i = 0; i < u_size; i++)
207             {
208                 dAu(d, i) =
209                   ((i > 0) ? _u_c[i] / _u_c[i-1] * i / sua * dAu(d - 1, i-1) : 0.0)
```

Implementation details: source file, part – IX

Trigonometric/TrigonometricBernsteinSurfaces3.cpp

```
210                     -((static_cast<GLint>(_n) - static_cast<GLint>(i)) / tua * dAu(d - 1, i)
211                     -((i < 2 * _n) ? (_u_c[i] / _u_c[i+1] * (2.0 * _n - i) / sua) * dAu(d - 1, i+1) : 0.0);
212                 }
213                 for (GLuint j = 0; j < v_size; j++)
214                 {
215                     dAv(d, j) =
216                     ((j > 0) ? (_v_c[j] / _v_c[j-1] * j / sva * dAv(d - 1, j-1)) : 0.0)
217                     -(static_cast<GLint>(_m) - static_cast<GLint>(j)) / tva * dAv(d - 1, j)
218                     -((j < 2 * _m) ? (_v_c[j] / _v_c[j+1] * (2.0 * _m - j) / sva) * dAv(d - 1, j+1) : 0.0);
219                 }
220             }
221             for (GLuint i = 0; i < u_size; ++i) {
222                 RowMatrix<DCoordinate3> diff_v(maximum_order_of_partial_derivatives + 1);
223                 for (GLuint j = 0; j < v_size; ++j) {
224                     for (GLuint d = 0; d <= maximum_order_of_partial_derivatives; d++)
225                     {
226                         diff_v[d] += _data(i, j) * dAv(d, j);
227                     }
228                 }
229                 for (GLuint d = 0; d <= maximum_order_of_partial_derivatives; d++) {
230                     for (GLuint r = 0; r <= d; r++)
231                     {
232                         pd(d, r) += diff_v[r] * dAu(d - r, i);
233                     }
234                 }
235             }
236             return GL_TRUE;
237         }
238 }
```

Project related example: bicubic Bézier patches

Matrix representation

Settings

- $[u_{\min}, u_{\max}] \times [u_{\min}, u_{\max}] = [0,1] \times [0,1]$;
- function systems F and G coincide and they correspond to the cubic Bernstein polynomials

$$\begin{aligned} B &= \left\{ B_i^3(t) = \binom{3}{i} t^i (1-t)^{3-i} : t \in [0,1] \right\}_{i=0}^{3} \\ &= \left\{ (1-t)^3, 3t(1-t)^2, 3t^2(1-t), t^3 : t \in [0,1] \right\}; \end{aligned}$$

- the information matrix $M = [\mathbf{p}_{i,j}]_{i=0,j=0}^{3,3} \in \mathcal{M}_{4,4}\left(\mathbb{R}^3\right)$ is a user-defined control net.

Project related example: bicubic Bézier patches. Header file

BicubicBezierPatches.h

```
1  #pragma once

2  #include "../Core/TensorProductSurfaces3.h"

3  namespace cagd
4  {
5      class BicubicBezierPatch: public TensorProductSurface3
6      {
7          ...

8      public:
9          BicubicBezierPatch();

10         // we have to implement pure virtual methods introduced in class TensorProductSurface3
11         GLboolean UBlendingFunctionValues(GLdouble u_knot, RowMatrix<GLdouble>& blending_values) const;
12         GLboolean VBlendingFunctionValues(GLdouble v_knot, RowMatrix<GLdouble>& blending_values) const;
13         GLboolean CalculatePartialDerivatives(GLuint maximum_order_of_partial_derivatives,
14                                               GLdouble u, GLdouble v, PartialDerivatives& pd) const;

15         ...
16     };
17 }
```

Project related example: bicubic Bézier patches. Source file, part I

BicubicBezierPatches.cpp

```
1  #include "BicubicBezierPatches.h"

2  using namespace cagd;

3  BicubicBezierPatch::BicubicBezierPatch(): TensorProductSurface3(0.0, 1.0, 0.0, 1.0, 4, 4)
4  {

5  }

6  GLboolean BicubicBezierPatch::UBlendingFunctionValues(
7          GLdouble u_knot, RowMatrix<GLdouble>& blending_values) const
8  {
9      if (u_knot < 0.0 || u_knot > 1.0)
10         return GL_FALSE;

11     blending_values.ResizeColumns(4);

12     GLdouble u = u_knot, u2 = u * u, u3 = u2 * u, w = 1.0 - u, w2 = w * w, w3 = w2 * w;

13     blending_values[0] = w3;
14     blending_values[1] = 3.0 * w2 * u;
15     blending_values[2] = 3.0 * w * u2;
16     blending_values[3] = u3;

17     return GL_TRUE;
18 }

19 GLboolean BicubicBezierPatch::VBlendingFunctionValues(
20         GLdouble v_knot, RowMatrix<GLdouble>& blending_values) const
21 {
22     // homework: it is similar to UBlendingFunctionValues
23 }
```

Project related example: bicubic Bézier patches. Source file, part II

BicubicBezierPatches.cpp

```
24 GLboolean BicubicBezierPatch::CalculatePartialDerivatives(
25         GLuint maximum_order_of_partial_derivatives,
26         GLdouble u, GLdouble v, PartialDerivatives& pd) const
27 {
28     if (u < 0.0 || u > 1.0 || v < 0.0 || v > 1.0 || maximum_order_of_partial_derivatives > 1)
29         return GL_FALSE;

30     // blending function values and their derivatives in u-direction
31     RowMatrix<GLdouble> u_blending_values(4), d1_u_blending_values(4);

32     GLdouble u2 = u * u, u3 = u2 * u, wu = 1.0 - u, wu2 = wu * wu, wu3 = wu2 * wu;

33     u_blending_values[0] = wu3;
34     u_blending_values[1] = 3.0 * wu2 * u;
35     u_blending_values[2] = 3.0 * wu * u2;
36     u_blending_values[3] = u3;

37     d1_u_blending_values[0] = -3.0 * wu2;
38     d1_u_blending_values[1] = -6.0 * wu * u + 3.0 * wu2;
39     d1_u_blending_values[2] = -3.0 * u2 + 6.0 * wu * u;
40     d1_u_blending_values[3] = 3.0 * u2;

41     // blending function values and their derivatives in v-direction
42     RowMatrix<GLdouble> v_blending_values(4), d1_v_blending_values(4);

43     GLdouble v2 = v * v, v3 = v2 * v, wv = 1.0 - v, wv2 = wv * wv, wv3 = wv2 * wv;

44     // homework
45     v_blending_values[0] = ...;
46     v_blending_values[1] = ...;
47     v_blending_values[2] = ...;
48     v_blending_values[3] = ...;
```

Project related example: bicubic Bézier patches. Source file, part III

BicubicBezierPatches.cpp

```
49      // homework
50      d1_v_blending_values[0] = ...;
51      d1_v_blending_values[1] = ...;
52      d1_v_blending_values[2] = ...;
53      d1_v_blending_values[3] = ...;

54      pd.ResizeRows(2);
55      pd.LoadNullVectors();
56      for (GLuint row = 0; row < 4; ++row)
57      {
58          DCoordinate3 aux_d0_v, aux_d1_v;
59          for (GLuint column = 0; column < 4; ++column)
60          {
61              aux_d0_v += _data(row, column) * v_blending_values(column);
62              aux_d1_v += _data(row, column) * d1_v_blending_values(column);
63          }
64          pd(0, 0) += aux_d0_v * u_blending_values(row);     // surface point
65          pd(1, 0) += aux_d0_v * d1_u_blending_values(row); // first order u-directional partial derivative
66          pd(1, 1) += aux_d1_v * u_blending_values(row);     // first order v-directional partial derivative
67      }

68      return GL_TRUE;
69  }
```

Project related example: bicubic Bézier patches. Usage

GLWidget.h

```
#pragma once

#include <GL/glew.h>
#include <QOpenGLWidget>
#include "../Bezier/BicubicBezierPatch.h"
#include "../Core/TriangulatedMeshes3.h"

namespace cagd
{
    class GLWidget: public QOpenGLWidget
    {
        Q_OBJECT

    private:

        ...

        // a bicubic Bezier patches
        BicubicBezierPatch     _patch;

        // triangulated meshes
        TriangulatedMesh3      *_before_interpolation, *_after_interpolation;

        ...

    public:

        ...

        ~GLWidget();
    };
}
```

Example: bicubic Bézier patches. Usage, part I

GLWidget.cpp

```
#include "GLWidget.h"

#include "Core/Materials.h"
#include "Core/Matrices.h"

using namespace cagd;
using namespace std;

void GLWidget::initializeGL()
{
    ...

    // before any GLEW specific function call you have to initialize the GLEW library
    glewInit();

    // define the control net of the bicubic Bezier patch
    _patch.SetData(0, 0, -2.0, -2.0, 0.0);
    _patch.SetData(0, 1, -2.0, -1.0, 0.0);
    _patch.SetData(0, 2, -2.0,  1.0, 0.0);
    _patch.SetData(0, 3, -2.0,  2.0, 0.0);

    _patch.SetData(1, 0, -1.0, -2.0, 0.0);
    _patch.SetData(1, 1, -1.0, -1.0, 2.0);
    _patch.SetData(1, 2, -1.0,  1.0, 2.0);
    _patch.SetData(1, 3, -1.0,  2.0, 0.0);

    _patch.SetData(2, 0,  1.0, -2.0, 0.0);
    _patch.SetData(2, 1,  1.0, -1.0, 2.0);
    _patch.SetData(2, 2,  1.0,  1.0, 2.0);
    _patch.SetData(2, 3,  1.0,  2.0, 0.0);

    _patch.SetData(3, 0,  2.0, -2.0, 0.0);
    _patch.SetData(3, 1,  2.0, -1.0, 0.0);
    _patch.SetData(3, 2,  2.0,  1.0, 0.0);
    _patch.SetData(3, 3,  2.0,  2.0, 0.0);
```

Example: bicubic Bézier patches. Usage, part II

GLWidget.cpp

```
// generate the mesh of the surface patch
_before_interpolation = _patch.GenerateImage(30, 30, GL_STATIC_DRAW);

if (_before_interpolation)
    _before_interpolation->UpdateVertexBufferObjects();

// define an interpolation problem:
// 1: create a knot vector in u-direction
RowMatrix<GLdouble> u_knot_vector(4);
u_knot_vector(0) = 0.0;
u_knot_vector(1) = 1.0 / 3.0;
u_knot_vector(2) = 2.0 / 3.0;
u_knot_vector(3) = 1.0;

// 2: create a knot vector in v-direction
ColumnMatrix<GLdouble> v_knot_vector(4);
v_knot_vector(0) = 0.0;
v_knot_vector(1) = 1.0 / 3.0;
v_knot_vector(2) = 2.0 / 3.0;
v_knot_vector(3) = 1.0;

// 3: define a matrix of data points, e.g. set them to the original control points
Matrix<DCoordinate> data_points_to_interpolate(4, 4);
for (GLuint row = 0; row < 4; ++row)
    for (GLuint column = 0; column < 4; ++column)
        _patch.GetData(row, column, data_points_to_interpolate(row, column));

// 4: solve the interpolation problem and generate the mesh of the interpolating patch
if (_patch.UpdateDataForInterpolation(u_knot_vector, v_knot_vector, data_points))
{
    _after_interpolation = _patch.GenerateImage(30, 30, GL_STATIC_DRAW);

    if (_after_interpolation)
        _after_interpolation->UpdateVertexBufferObjects();
}
```

Example: bicubic Bézier patches. Usage, part III

GLWidget.cpp

```
    ...
}


void GLWidget::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // saving the current modelview matrix
    glPushMatrix();

        ...

        if (_before_interpolation)
        {
            MatFBRuby.Apply();
            _before_interpolation->Render();
        }

        ...

        if (_after_interpolation)
        {
            glEnable(GL_BLEND);
            glDepthMask(GL_FALSE);
            glBlendFunc(GL_SRC_ALPHA, GL_ONE);
                MatFBTurquoise.Apply();
                _after_interpolation->Render();
            glDepthMask(GL_TRUE);
            glDisable(GL_BLEND);
        }

        ...

    // restoring the former modelview matrix
    glPopMatrix();
```

Example: bicubic Bézier patches. Usage, part IV

GLWidget.cpp

```
}

GLWidget::~GLWidget()
{
    ...

    if (_before_interpolation)
        delete _before_interpolation, _before_interpolation = 0;

    if (_after_interpolation)
        delete _after_interpolation, _after_interpolation = 0;

    ...
}
```

Project related example: bicubic Bézier patches. Results

At run-time

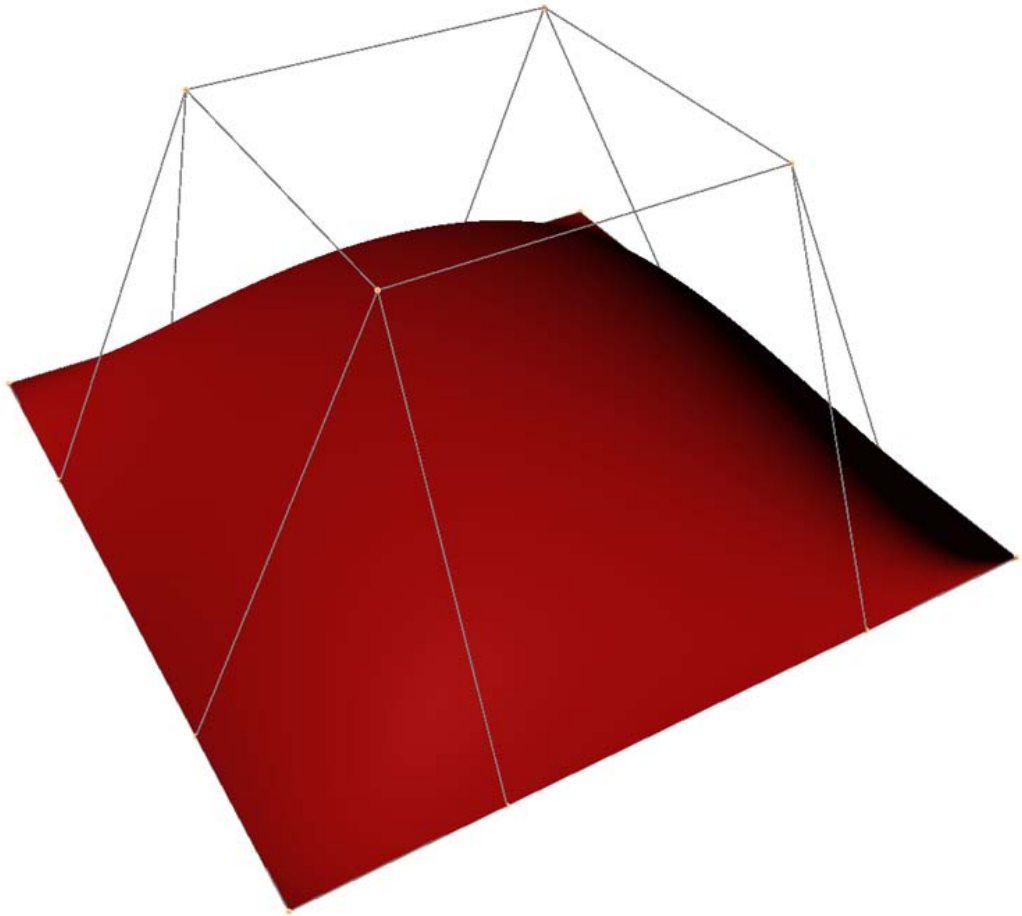


Fig. 10: A bicubic Bézier patch.

Project related example: bicubic Bézier patches. Results

At run-time

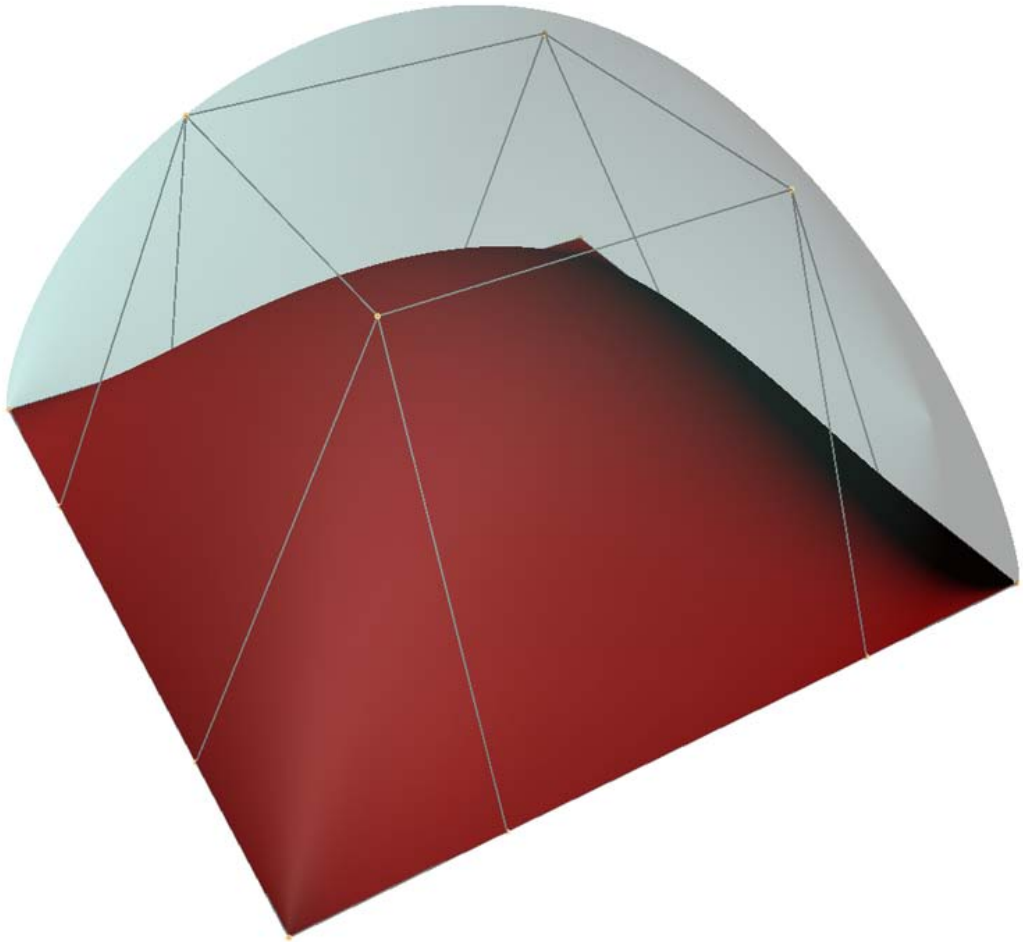


Fig. 11: Two bicubic Bézier patches: one of them interpolates the control net of the other patch.

Project related example: bicubic Bézier patches. Results

At run-time

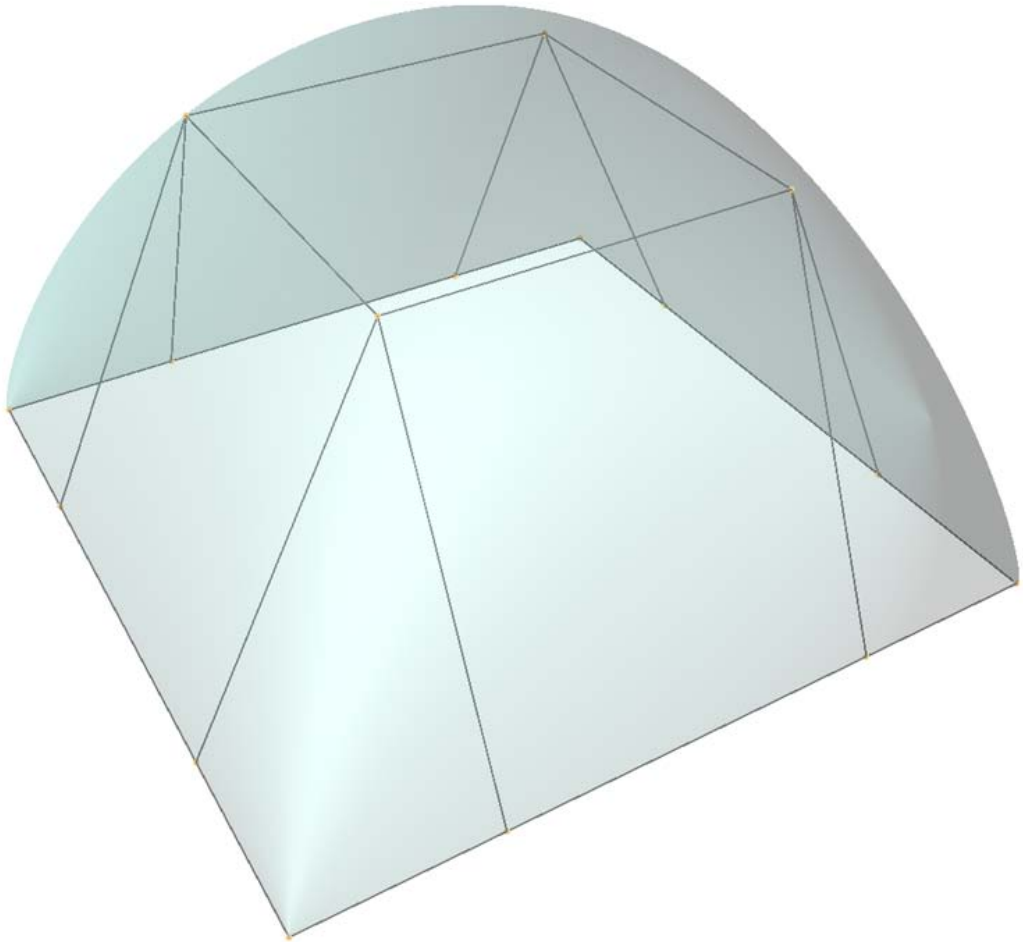


Fig. 12: An interpolating bicubic Bézier patch.

Bibliography – I


Coons, S., **1964**.
Surfaces for computer aided design,
Technical Report, MIT.


Coons, S., **1968**.
Rational cubic surface patches,
Technical Report, Project MAC, MIT.


Farin, G., **1997**.
Curves and surfaces for Computer-Aided Geometric Design, 4th ed.,
Academic Press, New York.


Forrest, A., 1968.
Curves and surfaces for computer-aided design,
Ph.D. Thesis, Cambridge.


Hoshek, J., Lasser, D., **1993**.
Fundamentals of Computer Aided Geometric Design,
A K Peters Ltd., Wellesley, Massachusetts, USA.


Juhász, I., Róth, Á., **2010**.
Closed rational trigonometric curves and surfaces,
Journal of Computational and Applied Mathematics, **234**(8):2390–2404.

Bibliography – II


Kaya, D., Wright, K., **2005**. *Parallel algorithms for LU decomposition on a shared memory multiprocessor*, Applied Mathematics and Computation, **163**(1):179–191.


Lyche, T., **1999**.
Trigonometric splines: a survey with new results,
In: Peña, J.M. (Ed.), Shape Preserving Representations in Computer-Aided Geometric Design. Nova Science Publishers Inc., New York, pp. 201–227.


Peña, J.M. (Ed.), **1999**.
Shape preserving representations in Computer-Aided Geometric Design,
Nova Science Publishers Inc.


Press, W.H., Teukolsky, S. A., Vetterling, W. T., Flannery, B. P., **2007**.
Numerical recipes. The art of scientific computing, 3rd ed., Cambridge University Press.


Róth, Á., Juhász, I., Schicho, J., Hoffmann, M., **2009**.
A cyclic basis for closed curve and surface modeling,
Computer Aided Geometric Design, **26**(5):528–546.


Sánchez-Reyes, J., **1998**.
Harmonic rational Bézier curves, p-Bézier curves and trigonometric polynomials,
Computer Aided Geometric Design, **15**(9):909–923.

Bibliography – III


Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis, **2005**.
OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2, 5th Edition,
Addison-Wesley Professional.


Versprille, K., J., **1975**.
Computer aided design applications of the rational B-spline approximation form,
Ph.D. Thesis, Syracuse University.