Individual Assignment 6

Introduction to Computer Graphics

Casper B. Hansen
University of Copenhagen
Department of Computer Science
fvx507@alumni.ku.dk

last revision April 4, 2014

Abstract

We discuss the concept of parametric surfaces, taking particular interest in Bezier patch surfaces and general surfaces.

The reader is expected to have read the previous assignment, in which many mathematical definitions used are presented, but it not required to have any prior knowledge of the material presented, which builds onto to that. Some parts require knowledge of partial differential equations and derivatives. For the code the reader is expected to have some basic understanding of linear algebra and the C++ language.

Contents

1 Theory

1.1 Parametric Surfaces

A parametric surface Q is — in 3-dimensional space — defined on $\mathbb{R}^2 \to \mathbb{R}^3$, taking a vector argument (s,t) for which $Q(s,t_0)$, where t_0 is a fixed value, is a parametric curve along s, and vice versa for $Q(s_0,t)$, where s_0 is a fixed value, is a parametric curve along t.

We know that a parametric curve can be expressed by $Q(t) = GMT$, where $G = \begin{bmatrix} G_1 & G_2 & G_3 & G_4 \end{bmatrix}$ is a vector of control points, M defines the basis matrix and $T = (t^3, t^2, t, 1)^T$ defines the parameter vector — all of which are discussed in the previous report.

As $Q(s_0,t)$ would define such a curve, let $G_i(s)$ be the ith control point of $Q(s_0,t)$, such that $G_i(s) = \begin{bmatrix} G_{1i} & G_{2i} & G_{3i} & G_{4i} \end{bmatrix}$ expresses the control points as a function of s. Then we can express G for parametric surfaces in much the same way, as we would parametric curves.

$$Q(s,t) = G(s)MT = \begin{bmatrix} G_1(s) & G_2(s) & G_3(s) & G_4(s) \end{bmatrix} MT \tag{1}$$

Unlike parametric curves, the matrix entries of G are then points on a parametric curve. That is, for a parametric curve, the points of that curve are now given as a function of s.

$$G_i(s) = \begin{bmatrix} g_{ix}(s) \\ g_{iy}(s) \\ g_{iz}(s) \end{bmatrix} = \begin{bmatrix} G_{1i} & G_{2i} & G_{3i} & G_{4i} \end{bmatrix} MS \qquad \text{where } S = \begin{bmatrix} s^3 \\ s^2 \\ s \\ 1 \end{bmatrix} \tag{2}$$

For each i we then have a row vector of control points. That is,

$$\begin{bmatrix} G_1(s) & G_2(s) & G_3(s) & G_4(s) \end{bmatrix} = S^T M^T \begin{bmatrix} G_{11} & G_{12} & G_{13} & G_{14} \\ G_{21} & G_{22} & G_{23} & G_{24} \\ G_{31} & G_{32} & G_{33} & G_{34} \\ G_{41} & G_{42} & G_{43} & G_{44} \end{bmatrix} \tag{3}$$

Substituting into (1) with the above, we get the parametric surface.

$$Q(s,t) = S^T M^T \begin{bmatrix} G_{11} & G_{12} & G_{13} & G_{14} \\ G_{21} & G_{22} & G_{23} & G_{24} \\ G_{31} & G_{32} & G_{33} & G_{34} \\ G_{41} & G_{42} & G_{43} & G_{44} \end{bmatrix} MT \tag{4}$$

The parametric surface is thus defined by the vector polynomials MT for t and $S^T M^T$ for s.

1.2 General Surfaces

A general surface is defined by a vector function; in our case, for the surfaces discussed we will constrain ourselves to vector functions f defined on $\mathbb{R}^2 \to \mathbb{R}^3$.

For a vertex $f(u,v)$ of any general surface, a vector which is perpendicular to the surface is given by the crossproduct of the partial derivatives of the vertex $f(u,v)$ with respect to u and v, respectively.

$$n(u,v) = \frac{\partial f(u,v)}{\partial u} \times \frac{\partial f(u,v)}{\partial v} \tag{5}$$

These are the normal vectors $n(u,v)$ of the surface at the vertex $f(u,v)$, effectively making them *vertex normals* which are the average of the normals of the faces sharing the vertex, as opposed to *face normals* which stand perpendicular to the faces.

2 Implementation

2.1 Bezier Patch Models

A few classes were provided in the assignment handout, describing Bezier rows, columns and patches. These were used to ease the calculations of the patches. I will omit details of how buffers are created, reallocated and rendered, and focus on the subdivision algorithm.

Since the matrices M, D_B^L and D_B^L do not change, we do not want to keep calculating them. Therefore, they are declared static, inside the method — this increases efficiency.

```
78  void BezierPatchModel::subdivide(const BezierPatch &patch, int n)
79  {
80      static glm::mat4x4 M(glm::vec4(-1.0f,  3.0f, -3.0f, 1.0f),
81              glm::vec4( 3.0f, -6.0f,  3.0f, 0.0f),
82              glm::vec4(-3.0f,  3.0f,  0.0f, 0.0f),
83              glm::vec4( 1.0f,  0.0f,  0.0f, 0.0f));
84
85      static glm::mat4x4 DLB = glm::mat4x4(glm::vec4(8.0f, 0.0f, 0.0f, 0.0f),
86              glm::vec4(4.0f, 4.0f, 0.0f, 0.0f),
87              glm::vec4(2.0f, 4.0f, 2.0f, 0.0f),
88              glm::vec4(1.0f, 3.0f, 3.0f, 1.0f)) / 8.0f;
89
90      static glm::mat4x4 DRB = glm::mat4x4(glm::vec4(1.0f, 3.0f, 3.0f, 1.0f),
91              glm::vec4(0.0f, 2.0f, 4.0f, 2.0f),
92              glm::vec4(0.0f, 0.0f, 4.0f, 4.0f),
93              glm::vec4(0.0f, 0.0f, 0.0f, 8.0f)) / 8.0f;
```

Figure 1: Code excerpt of the static matrices M, D_B^L and D_B^R. (BezierPatchModel.cpp)

The method is passed a variable n, which determines the depth of subdivisions to be made. The recursion invariant is that as long as $n > 0$ we have yet to reach the desired subdivision depth, thus $n = 0$ defines the exit condition.

So, while this invariant holds, we must calculate each subdivision of the current patch, which was passed to the method. On each of these, we recurse decreasing the passed n.

```
142     else {
143         BezierPatch G11 = glm::transpose(DLB) * patch * DLB;
144         BezierPatch G12 = glm::transpose(DRB) * patch * DLB;
145         BezierPatch G21 = glm::transpose(DLB) * patch * DRB;
146         BezierPatch G22 = glm::transpose(DRB) * patch * DRB;
147
148         subdivide(G11, n - 1);
149         subdivide(G12, n - 1);
150         subdivide(G21, n - 1);
151         subdivide(G22, n - 1);
152     }
```

Figure 2: Code excerpt showing the recursive case of the algorithm. (BezierPatchModel.cpp)

When n becomes zero, we have arrived at a satisfactory depth of subdivision, and we can then calculate the vertices and normals for the patch at $Q(0.0, 0.0)$, $Q(1.0, 0.0)$, $Q(1.0, 1.0)$ and $Q(0.0, 1.0)$ — the extremities of the subdivided patch.

We do so by calculating the nondependent matrix $M^T GM$ first, on line 99, and then for each extremity we calculate the vertex at the parameters s and t, as shown on lines 101–119.

```
99          BezierPatch MTGM = (glm::transpose(M) * patch * M);
100
101         s = 0.0f;   t = 0.0f;
102         S = glm::vec4(s*s*s, s*s, s, 1.0f);
103         T = glm::vec4(t*t*t, t*t, t, 1.0f);
104         glm::vec3 P1 = S * MTGM * T;
105
106         s = 1.0f;   t = 0.0f;
107         S = glm::vec4(s*s*s, s*s, s, 1.0f);
108         T = glm::vec4(t*t*t, t*t, t, 1.0f);
109         glm::vec3 P2 = S * MTGM * T;
110
111         s = 1.0f;   t = 1.0f;
112         S = glm::vec4(s*s*s, s*s, s, 1.0f);
113         T = glm::vec4(t*t*t, t*t, t, 1.0f);
114         glm::vec3 P3 = S * MTGM * T;
115
116         s = 0.0f;   t = 1.0f;
117         S = glm::vec4(s*s*s, s*s, s, 1.0f);
118         T = glm::vec4(t*t*t, t*t, t, 1.0f);
119         glm::vec3 P4 = S * MTGM * T;
```

Figure 3: Code excerpt showing the bottom-out case of the algorithm; vertex calculations. (BezierPatchModel.cpp)

Having the vertices of the patch, we can calculate the face normals of the patch.

$$\begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix} = P_1P_2 \times P_1P_3 \tag{6}$$

The formula for normal $(n_x, n_y, n_z)^T$ of a triangle given by vertices P_1, P_2 and P_3 is given above, and used in the implementation, as shown below. Note that a patch consists of two triangles, hence we must calculate both (lines 121 and 122). Furthermore, I chose to use the average of these to give the face normal of the quad (line 124), not each triangle.

```
121         glm::vec3 N1 = glm::cross(P1-P2, P1-P3);
122         glm::vec3 N2 = glm::cross(P1-P3, P1-P4);
123
124         glm::vec3 N  = glm::normalize( (N1 + N2) / 2.0f );
```

Figure 4: Code excerpt showing the bottom-out case of the algorithm; normal calculations. (BezierPatchModel.cpp)

Now that vertices and normals have been calculated, we can supply them to the buffer, the details of which are left out, but can be referenced in the source.

2.2 General Surfaces

The concept of a general surface is that its vertices, normals and texture coordinates are defined by formulae. Since the only difference between individual surfaces is the formulae from which it derives these data, we can make an abstract class which does not depend on any specific surface instance.

The abstract class function is to eliminate repeated code for actual surfaces; common data across all general surfaces, allocation and initialization, and even OpenGL calls.

```
27 class GeneralSurface : public Object
28 {
29 public:
30     GeneralSurface(int N, int M,
31                    float u_min, float u_max,
32                    float v_min, float v_max);
33     virtual ~GeneralSurface();
34
35     void initializeSurface(int parts);
36     virtual void render(int mask) const;
37
38     virtual glm::vec3 vertexAt(int part, float u, float v) = 0;
39     virtual glm::vec3 normalAt(int part, float u, float v) = 0;
40
41 protected:
42     int N, M;
43     float u_min, u_max;
44     float v_min, v_max;
45
46     GLuint vao;
47     GLuint buffers[BUFFER_COUNT];
48
49     int num_vertices;
50
51 private:
52 };
```

Figure 5: Code excerpt the definition of a `GeneralSurface`. (GeneralSurface.h)

It works by setting the basic properties in the constructor, where it also creates an OpenGL vertex array object and associated buffer objects. These are destroyed again in the destructor. Note specifically that the interface to fill these buffers comes in the form of a *part initialization* method, which will go through a number of parts, and for each part it will call the *pure virtual* methods[1] which retrieves the vertices and normals for any given part at surface coordinates u and v. This effectively keeps potentially harmful procedures out of reach to any subclass and makes it very extensible, new surfaces can be added it a matter of minutes.

[1] On a sidenote, since these two methods are pure virtual, that makes the class *pure abstract*, meaning that we cannot instiate a `GeneralSurface`, it must be a subclass of it.

2.2.1 Dini's Surface

The surface named after Italian mathematician Ulisse Dini consists of just one surface part. It dependents on two variables a and b, which are given in and set by the constructor. Predefined values for u_{min}, u_{max}, v_{min} v_{max} are given here as well. Once set, the constructor simply calls the superclass `initializeSurface` method. As Dini's surface is only composed of one single part, the argument is obsolete in this case, as it is ignored see figure 6. Note that $u \in [0; 6\pi]$ and $v \in [0.001; 2]$.

$$f(u,v) = \begin{bmatrix} a\cos u \sin v \\ a \sin u \sin v \\ a(\cos v + \log(\tan \frac{v}{2})) + bu; \end{bmatrix} \tag{7}$$

To define the vertices of Dini's surface we follow the formula given above.

```
29      glm::vec3 ret;
30      ret.x = a * cosf(u) * sinf(v);
31      ret.y = a * sinf(u) * sinf(v);
32      ret.z = a * (cosf(v) + logf(tanf(0.5f * v))) + b * u;
33      return ret;
```

Figure 6: Code excerpt the overloaded `vertexAt` class method for `DiniSurface`. (DiniSurface.cpp)

Correspondingly, as discussed (see section 1.2 on page 2), the normals of Dini's surface are given by the following formula.

$$n(u,v) = \frac{\partial f(u,v)}{\partial u} \times \frac{\partial f(u,v)}{\partial v} = \begin{bmatrix} -a\sin u \sin v \\ a \cos u \sin v \\ b \end{bmatrix} \times \begin{bmatrix} a\cos u \cos v \\ a \sin u \sin v \\ a(\frac{0.5}{\sin(0.5v)\cos(0.5v)} - \sin v) \end{bmatrix} \tag{8}$$

Do note that we must normalize the resulting vector on line 50, as we are dealing with computer graphics, and although the vector may be perpendicular to the surface, if we did not scale it to unit length our shader would not look right.

```
40      glm::vec3 du, dv;
41
42      du.x = -a * sinf(u) * sinf(v);
43      du.y =  a * cosf(u) * sinf(v);
44      du.z = b;
45
46      dv.x = a * cosf(u) * cosf(v);
47      dv.y = a * sinf(u) * sinf(v);
48      dv.z = a * ( (0.5f / (sinf(0.5f * v) * cosf(0.5f * v))) - sinf(v));
49
50      return glm::normalize( glm::cross(du, dv) );
```

Figure 7: Code excerpt the overloaded `normalAt` class method for `DiniSurface`. (DiniSurface.cpp)

2.2.2 Klein's Bottle

This surface is interesting in that it has no backside. It consists of four surface parts, which are defined below. Note that $u, v \in [0; 2\pi]$.

$$f_{top}(u,v) = \begin{bmatrix} 2 + (2 + \cos u)\cos v \\ \sin u \\ 3\pi + (2 + \cos u)\cos v \end{bmatrix} \qquad f_{middle}(u,v) = \begin{bmatrix} (2.5 + 1.5\cos v)\cos u \\ (2.5 + 1.5\cos v)\sin u \\ 3v \end{bmatrix}$$

$$f_{bottom}(u,v) = \begin{bmatrix} (2.5 + 1.5\cos v)\cos u \\ (2.5 + 1.5\cos v)\sin u \\ -2.5\sin v \end{bmatrix} \qquad f_{handle}(u,v) = \begin{bmatrix} 2 - 2\cos v + \sin u \\ \cos u \\ 3v \end{bmatrix}$$

As can be seen, depending on which part we are currently processing, the corresponding formula is applied to the input vector (u, v) producing the vertex as mapped by $f_i(u, v)$.

```
27 glm::vec3 KleinBottle::vertexAt(int part, float u, float v)
28 {
29     glm::vec3 ret;
30
31     switch (part)
32     {
33         case PART_BOTTOM:
34             ret.x = (2.5f + 1.5f * cosf(v)) * cosf(u);
35             ret.y = (2.5f + 1.5f * cosf(v)) * sinf(u);
36             ret.z = -2.5f * sinf(v);
37             break;
38
39         case PART_HANDLE:
40             ret.x = 2.0f - 2.0f * cosf(v) + sinf(u);
41             ret.y = cosf(u);
42             ret.z = 3.0f * v;
43             break;
44
45         case PART_TOP:
46             ret.x = 2.0f + (2.0f + cosf(u)) * cosf(v);
47             ret.y = sinf(u);
48             ret.z = 3.0f * M_PI + (2.0f + cosf(u)) * sinf(v);
49             break;
50
51         case PART_MIDDLE:
52             ret.x = (2.5f + 1.5f * cosf(v)) * cosf(u);
53             ret.y = (2.5f + 1.5f * cosf(v)) * sinf(u);
54             ret.z = 3.0f * v;
55             break;
56     }
57
58     return ret;
59 }
```

Figure 8: Code excerpt the overloaded `vertexAt` class method for `KleinBottle`. (KleinBottle.cpp)

Likewise, calculated as we do any general surface (see section 1.2 on page 2), the normals are given by the formulae below.

$$n_{top}(u,v) = \begin{bmatrix} (2.5 + 1.5\cos u)\cos u \cos v \\ (2.5 + 1.5\cos u)\sin u \\ (2.5 + 1.5\cos u)\cos u \sin v \end{bmatrix} \qquad n_{middle}(u,v) = \begin{bmatrix} (7.5 + 4.5\cos v)\cos u \\ (7.5 + 4.5\cos v)\sin u \\ (3.75 + 2.25\cos v)\sin v \end{bmatrix}$$

$$n_{bottom}(u,v) = \begin{bmatrix} (-6.75 - 3.75\cos v)\cos u \cos v \\ (-6.75 - 3.75\cos v)\sin u \cos v \\ (3.75 + 2.25\cos v)\sin v \end{bmatrix} \qquad n_{handle}(u,v) = \begin{bmatrix} -3\sin u \\ -3\cos u \\ 2\sin u \sin v \end{bmatrix}$$

As we did the vertices, so do we do the surface normals, but with a small difference; once a surface normal has been calculated we need to scale it to unit length. That is, we have to normalize the returned vector on line 92.

```
61 glm::vec3 KleinBottle::normalAt(int part, float u, float v)
62 {
63     glm::vec3 ret;
64 
65     switch (part)
66     {
67         case PART_BOTTOM:
68             ret.x = (-6.25f - 3.75f * cosf(v)) * cosf(v) * cosf(u);
69             ret.y = (-6.25f - 3.75f * cosf(v)) * cosf(v) * sinf(u);
70             ret.z = (3.75f + 2.25f * cosf(v)) * sinf(v);
71             break;
72 
73         case PART_HANDLE:
74             ret.x = -3.0f * sinf(u);
75             ret.y = -3.0f * cosf(u);
76             ret.z = 2.0f * sinf(u) * sinf(v);
77             break;
78 
79         case PART_TOP:
80             ret.x = (2.0f + cosf(u)) * cosf(u) * cosf(v);
81             ret.y = (2.0f + cosf(u)) * sinf(u);
82             ret.z = (2.0f + cosf(u)) * cosf(u) * sinf(v);
83             break;
84 
85         case PART_MIDDLE:
86             ret.x = (7.5f + 4.5f * cosf(v)) * cosf(u);
87             ret.y = (7.5f + 4.5f * cosf(v)) * sinf(u);
88             ret.z = (3.75f + 2.25f * cosf(v)) * cosf(v);
89             break;
90     }
91 
92     return glm::normalize(ret);
93 }
```

Figure 9: Code excerpt the overloaded `normalAt` class method for `KleinBottle`. (KleinBottle.cpp)

2.3 Tests

The supplied program source code can render all of the above discussed objects. When running, pressing the space key will browse models and pressing pressing the enter key will browse model types (ie. patch models and general surfaces). Using the 1–3 numeric keys one can change the rendering type (ie. points, lines[2] and triangles) to verify the geometry.

2.3.1 Model screenshots

I present a few rendered screenshots of the models discussed. Both of these sequences were captured at subdivision intervals of two; the left-most picture is rendered at level 0, that is no subdivision at all, the following at level 2, and lastly the right-most at level 4.

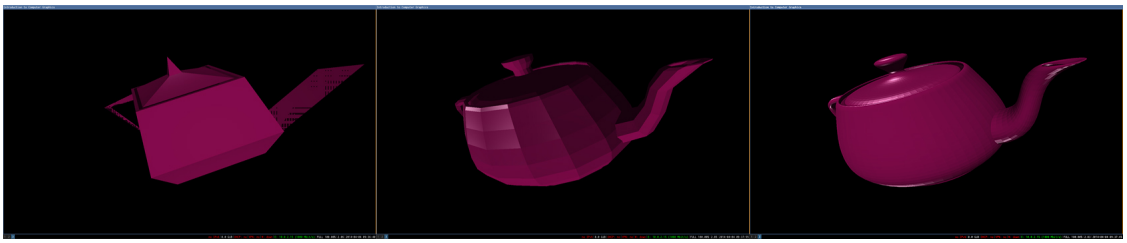

Figure 10: Renderings of the teapot at different subdivision levels.

As can be seen, the algorithm works correctly, as each quad subdivided becomes four quads. As we increase the number of subdivisions the model becomes smoother and the light reflected off of it looks increasingly realistic.

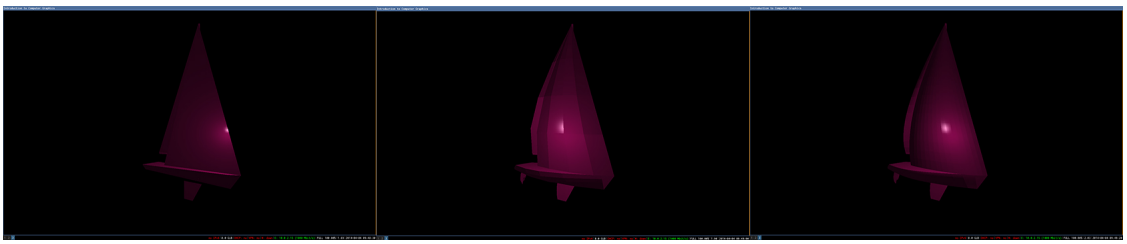

Figure 11: Renderings of the boat at different subdivision levels.

Do note that I decided not to put any effort into computing vertex normals, as opposed to face normals, which is what is used. Therefore, the model quads are still very visible. One could mend this by averaging the face normals who share a vertex, and aligning them with an index buffer object for the shader to interpolate these better across the surface. This is outside the scope of the curriculum, hence the decision of leaving it out.

2.3.2 Bezier patch surfaces screenshots

I present a few screenshots of the bezier patch surfaces discussed. Dini's surface, on the left, uses a two-sided shader which colors backfaces using a different color. This produces a very pretty surface model that shows both sides of the model.

[2]Lines will not render correctly for all models, as the vertex data is organized by triangle faces.

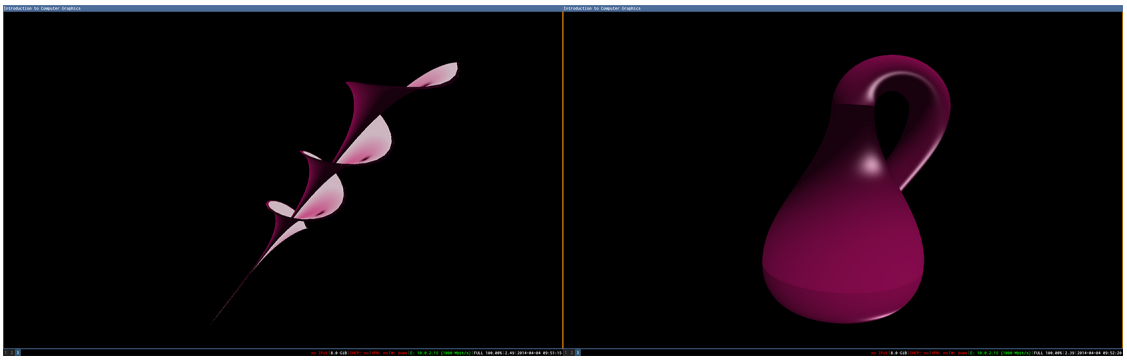

Figure 12: Renderings of (left) Dini's surface and (right) the Klein Bottle.

The Klein bottle, on the right, exposes a shader problem. Unfortunately, I did not get to fix it in time for the assignment delivery, but I believe I know of where the problem stems from. The shader makes use of an *eye* vector, which is not updated properly, as the camera moves about. Further, another source of this effect could be that the shader doesn't account properly for the nature of this model, namely that it has no backside. The way the shader is constructed, it tries to invert the normals of the surface if a triangle is facing backward.