Individual Assignment 5

Introduction to Computer Graphics

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

last revision March 13, 2014

Abstract

We will discuss the concept of Bezier curves and how they can be described in mathematical form. A discussion of how the geometry vector and basis matrix affects the behaviour of the Bezier curve, and lastly we will discuss the of three different implementation techniques.

The reader is not expected to have any prior knowledge of the material presented. For the code the reader is expected to have some basic understanding of linear algebra and the C++ language.

Contents

1 Theory

The general form of a parametric curve is a vector function. That is, a scalar *parameter* $t \in \mathbb{R}$ is given as the argument of the vector function, which maps it to a vector $v \in \mathbb{R}^n$.

$$f : \mathbb{R} \to \mathbb{R}^n \tag{1}$$

We can rewrite this in 3-dimensional vector form, as a polynomial of degree 3 in 3 dimensions.

$$f(t) = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix} = \begin{bmatrix} a_x t^3 + b_x t^2 + c_x t + d_x \\ a_y t^3 + b_y t^2 + c_y t + d_y \\ a_z t^3 + b_z t^2 + c_z t + d_z \end{bmatrix} \quad 0 \le t \le 1 \tag{2}$$

For which the coordinate functions $x(t)$, $y(t)$ and $z(t)$ are regular polynomials of degree 3.

1.1 Coefficient Matrix

Expanding this further, we can rewrite it in matrix form.

$$f(t) = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix} = \begin{bmatrix} a_x + b_x + c_x + d_x \\ a_y + b_y + c_y + d_y \\ a_z + b_z + c_z + d_z \end{bmatrix} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} \quad 0 \le t \le 1 \tag{3}$$

This matrix is what we called the *coefficient matrix*, and we will denote it in vector form as $\mathbf{C}$. We then have that $f(t) = \mathbf{C}t$.

1.2 Geometry and Basis Matrices

Since the $\mathbf{C}$ gives us little information about the shape of the curve, we will expand $\mathbf{C}$ and write it as the product of two matrices; a *geometry matrix* which we will denote as $\mathbf{G}$ and a *basis matrix* which we will denote $\mathbf{M}$.

The geometry matrix contains information about the curve. That is, each column vector of the geometry matrix describes the curves control points. Since all information about the curve is contained within the geometry matrix, then the basis matrix simply needs to transform the geometry matrix into $\mathbf{C}$, making the basis matrix constant. The basis matrix determines the type of the curve (i.e. Bezier, Hermitian, etc.).

Since the basis matrix is constant, let us define it. A curves points are given by $f_H(t) = G_H M_H \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix}^T$, and its tangents are given by $f'_H(t) = G_H M_H \begin{bmatrix} 3t^2 & 2^t & 1 & 0 \end{bmatrix}^T$. We then have that

$$f_H(0) = G_1 = G_H M_H \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad \text{and} \quad f_H(1) = G_2 = G_H M_H \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \tag{4}$$

$$f'_H(0) = G_3 = G_H M_H \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad \text{and} \quad f'_H(1) = G_4 = G_H M_H \begin{bmatrix} 3 \\ 2 \\ 1 \\ 0 \end{bmatrix} \tag{5}$$

And thus M_H is given by the aboves inverse.

$$M_H = \begin{bmatrix} 1 & 0 & -3 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & -3 \\ 0 & 1 & 0 & 3 \end{bmatrix}^{-1} = \begin{bmatrix} 2 & -3 & 0 & 1 \\ -2 & 3 & 0 & 0 \\ 1 & -2 & 0 & 0 \\ 1 & -1 & 1 & 0 \end{bmatrix} \tag{6}$$

This defines the basis matrix of Hermitian curves M_H. We can use a relation between the Hermitian basis matrix to produce the Bezier basis matrix M_B by recognizing that $\begin{bmatrix} G_1 & G_2 & G_3 & G_4 \end{bmatrix}_H = \begin{bmatrix} G_1 & G_4 & 3(G_2 - G_1) & 3(G_4 - G_3) \end{bmatrix}_B$, giving us the following basis conversion matrix.

$$G_H = \begin{bmatrix} G_1 & G_2 & G_3 & G_4 \end{bmatrix}_B \begin{bmatrix} 1 & 0 & -3 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & -3 \\ 0 & 1 & 0 & 3 \end{bmatrix}_{B \to H} \tag{7}$$

By this, we see that $G_H = G_B M_{B \to H}$, and following through on this we have that $f(t) = G_H M_H t = (G_B M_{B \to H}) M_H t = G_B (M_{B \to H} M_H) t$. From this, we see that

$$M_B = M_{B \to H} M_H = \begin{bmatrix} 1 & 0 & -3 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & -3 \\ 0 & 1 & 0 & 3 \end{bmatrix} \begin{bmatrix} 2 & -3 & 0 & 1 \\ -2 & 3 & 0 & 0 \\ 1 & -2 & 0 & 0 \\ 1 & -1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{8}$$

And thus, we have defined the basis matrix of Bezier curves. By inspection, we can see that the basis matrix of Bezier curves creates a weighted sum of the control points at the parameter t — in essence, the point on a Bezier curve is pulled in all control point directions and is put in place by the weight given for each control point at a given t.

This abstraction from **C** allows for easy interpretation and modification of the curve geometry and how it is being transformed.

2 Implementation

All of the algorithms are handled by the same `draw` method in the `BezierVec4` class. Therefore, we declare the common elements at the head of the function, including some helper macros to shorten the formulae.

```
    #define G1 points[0]
    #define G2 points[1]
    #define G3 points[2]
    #define G4 points[3]

    double t;
    double delta = 1.0 / n;
    glm::vec3 last = points[0];
    glm::vec3 point;
```

Figure 1: Code excerpt showing the common declarations and helper macros.

2.1 Sampled Curve

The most computationally expensive way of computing a Bezier curve we will look at is by way of *sampling*. We simply calculate each and every point of the segments using the regular polynomial functions, as can be seen on lines 79–82.

```
77              for (t = 0.0; t <= 1.0; t += delta)
78              {
79                  point = powf((1.0f - t), 3.0f) * points[0]
80                        + (float)(3.0f * t * powf((1 - t), 2.0f)) * points[1]
81                        + (float)(3.0f * (1.0f - t) * powf(t, 2.0f)) * points[2]
82                        + powf(t, 3.0f) * points[3];
83
84                  glBegin(GL_LINES);
85                      glVertex3f(last.x, last.y, last.z);
86                      glVertex3f(point.x, point.y, point.z);
87                  glEnd();
88                  last = point;
89              }
```

Figure 2: Code excerpt showing the sampled curve algorithm.

This method is not considered to be efficient in any way, but it is very simple.

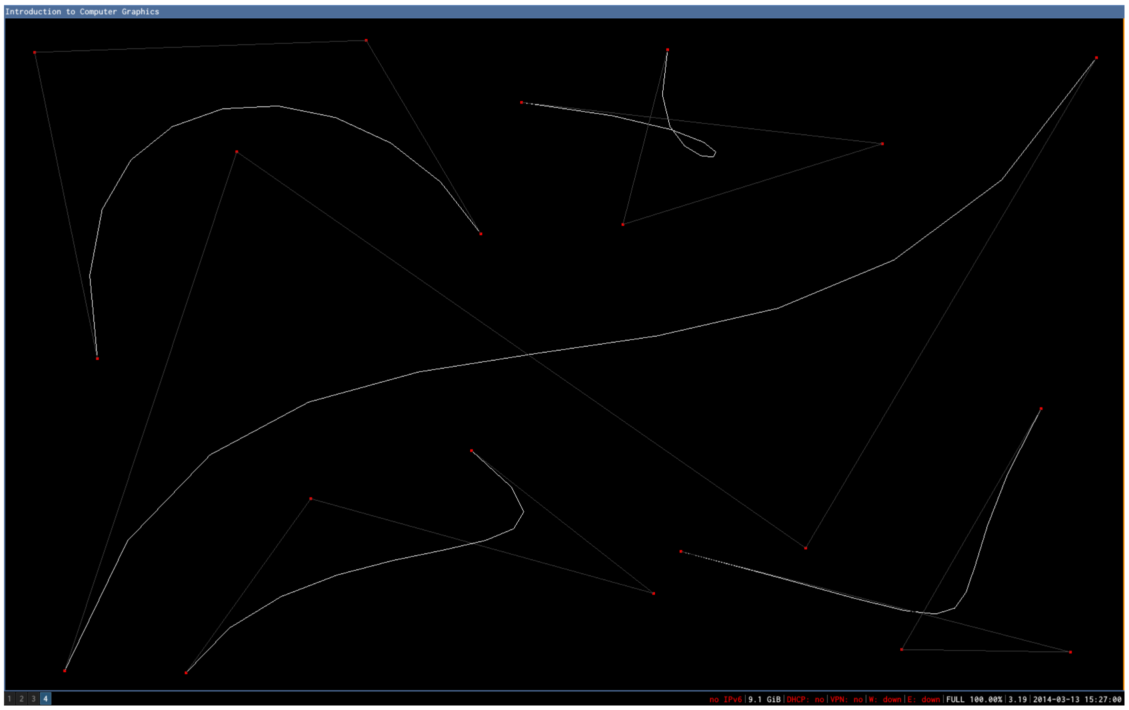
Introduction to Computer Graphics
1 2 3 4
no IPv6 | 9.1 GiB | DHCP: no | VPN: no | W: down | E: down | FULL 100.00% | 3.19 | 2014-03-13 15:27:00

Figure 3: A screenshot showing sampled Bezier curves.

The above screenshot shows the how the sampled Bezier curve implementation looks with 10 segments.

2.2 Forward Difference

This method is a lot more efficient than sampling discussed previously. It using a lot of pre-calculation to reduce overhead and minimizes computationally expensive operations.

Consider the polynomial coordinate function $f(t) = at^3 + bt^2 + ct + d$. We want to calculate, from the starting point, the next point at an offset into the curve of $\Delta t = \delta$.

We know that $\Delta f(t) = f(t + \delta) - f(t)$ and $f(t + \delta) = f(t) + \Delta f(t)$, so we have that $f_{n+1} = f_n + \Delta f_n$. Substituting this into the polynomial equation and reducing the expression, we have that

$$\Delta f(t) = 3a\Delta t^2 + (3a\Delta^2 + 2b\Delta)t + a\Delta^3 + b\Delta^2 + c\Delta \tag{9}$$

In which the highest order term is $3a\Delta t^2$, making it a polynomial of degree 2. So, the original polynomial degree was decreased by one. Repeating this procedure until we reach a constant expression yields

$$\Delta f(t) = 3a\Delta t^2 + (3a\Delta^2 + 2b\Delta)t + a\Delta^3 + b\Delta^2 + c\Delta \tag{10}$$

$$\Delta^2 f(t) = 6a\Delta^2 t + 6a\Delta^3 + 2b\Delta^2 \tag{11}$$

$$\Delta^3 f(t) = 6a\Delta^3 \tag{12}$$

This is what is done on lines 101–110. Note also, that for this we have to use the geometry vectors, which are calculated as discussed earlier on lines 95–99.

Now that we have the delta points, we can use them to incrementally calculate the next point in the Bezier curve. The advancement is dependent on Δ, which we define as $\Delta = \frac{1}{n}$, where n is the number of curve segments. In looking at the differences across the delta points we find that the initialization is then

$$\begin{bmatrix} f(0) \\ \Delta f(0) \\ \Delta^2 f(0) \\ \Delta^3 f(0) \end{bmatrix} = \begin{bmatrix} d \\ a\Delta^3 + b\Delta^2 + c\Delta \\ a\Delta^3 + 2b\Delta^2 \\ 6a\Delta^3 \end{bmatrix} \tag{13}$$

as adding $\Delta^n f$ and $\Delta^{n-1} f$ produces the advanced $\Delta^n f$ for the following point. Thus, we simply have to update these accordingly, while we render each point. This is done on lines 112–125.

```
 95         // calculate GM matrix vectors
 96         glm::vec3 a = -G1 + 3.0f * G2 - 3.0f * G3 + G4;
 97         glm::vec3 b = 3.0f * G1 - 6.0f * G2 + 3.0f * G3;
 98         glm::vec3 c = -3.0f * G1 + 3.0f * G2;
 99         glm::vec3 d = G1;
100 
101         // initialize deltas
102         glm::vec3 deltas[4];
103         float t1 = delta;
104         float t2 = t1 * t1;
105         float t3 = t2 * t1;
106 
107         deltas[0] = d;
108         deltas[1] = a * t3 + b * t2 + c * t1;
109         deltas[2] = 6.0f * a * t3 + 2.0f * b * t2;
110         deltas[3] = 6.0f * a * t3;
111 
112         last = deltas[0];
113         for (t = 0.0; t <= (1.0 - delta); t += delta)
114         {
115             deltas[0] += deltas[1];
116             deltas[1] += deltas[2];
117             deltas[2] += deltas[3];
118 
119             glBegin(GL_LINES);
120                 glVertex3f(last.x, last.y, last.z);
121                 glVertex3f(deltas[0].x, deltas[0].y, deltas[0].z);
122             glEnd();
123 
124             last = deltas[0];
125         }
```

Figure 4: Code excerpt showing the forward difference algorithm.

Note that on line 111 the condition of t is in the interval $[0; 1 - \frac{1}{n}]$. If we were to allow t to reach 1 in this loop, we would go one curve segment beyond the last control point. This is side effect of a poorly thought out loop invariant.

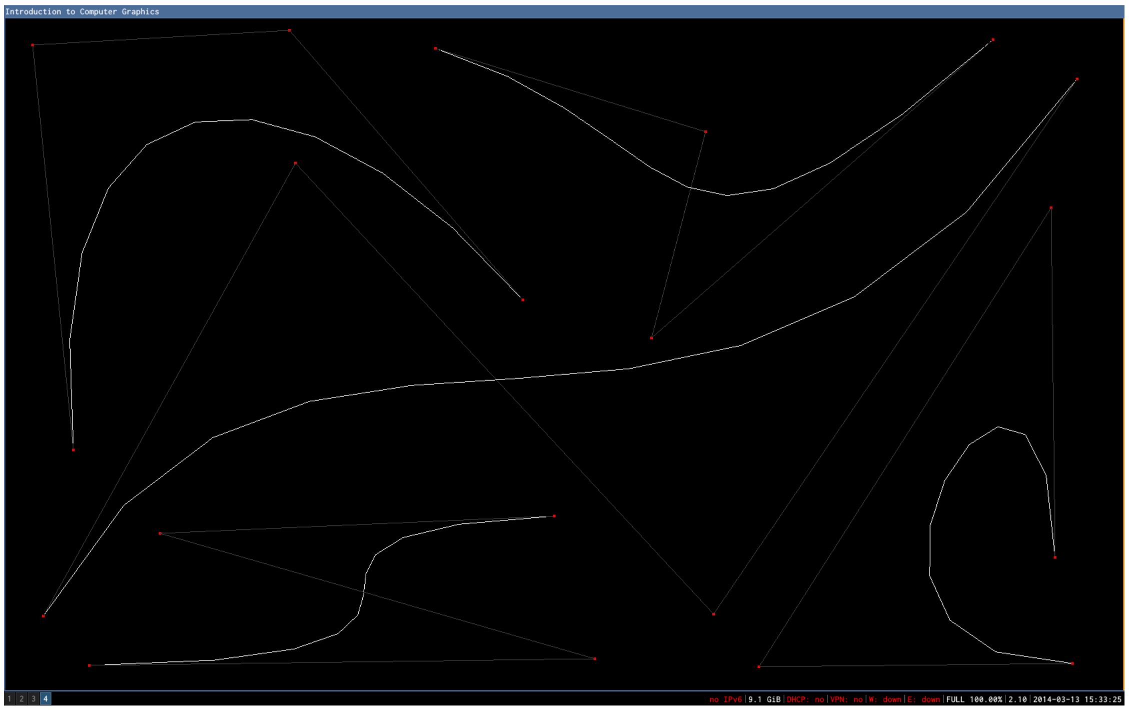
Introduction to Computer Graphics
1 2 3 4
no IPv6 | 9.1 GiB | DHCP: no | VPN: no | W: down | E: down | FULL 100.00% | 2.10 | 2014-03-13 15:33:25

Figure 5: A screenshot showing forward differencing Bezier curves.

The above screenshot shows the how the forward difference Bezier curve implementation looks with 10 segments.

2.3 Subdivision

The algorithm with the highest quality, however, is the subdivision algorithm. It generates approximation points along the curve and in doing so it decreases the distance from the approximation to the actual curve. Once an decent distance is achieved it considers the points generated to be acceptable.

We consider a Bezier curve with control points G_1, G_2, G_3 and G_4. Let H be the midway point between G_2 and G_3, and let L_2 be the midway point between G_1 and G_2, and let R_2 be the midway point between G_3 and G_4. Further, let L_3 and R_3 be the midway points between H and L_2, and H and R_2, respectively. Finally, let $L_1 = G_1$ and $R_1 = G_4$, and let $L_4 = R_4$ be the midway point between L_3 and R_3.

We then have the following formulae at our disposal

$$H = (G_2 + G_3)/2 \tag{14}$$

$$L_1 = G_1 \tag{15}$$

$$L_2 = (G_1 + G_2)/2 \tag{16}$$

$$L_3 = (H + L_2)/2 \tag{17}$$

$$L_4 = (L_3 + R_3)/2 \tag{18}$$

$$R_1 = G_4 \tag{19}$$

$$R_2 = (G_3 + G_4)/2 \tag{20}$$

$$R_3 = (H + L_2)/2 \tag{21}$$

$$R_4 = (L_3 + R_3)/2 \tag{22}$$

These expressions constitute newly generated points on of the curve. In the code these are defined on lines 185–192.

Now, we have to determine whether or not these points are close enough approximations of the curve. This is done by the so-called *flatness test* for which we will need to determine the distance function.

$$d(x) = \left|(x - G_1) - \left[(x - G_1) \cdot \frac{G_4 - G_1}{|G_4 - G_1|}\right] \frac{G_4 - G_1}{|G_4 - G_1|}\right| \tag{23}$$

We use this to determine how far away the points G_2 and G_3 are on lines 162–168. The farthest point is then used for comparison with the ϵ on line 170. If the point exceeds the ϵ we recurse with the L and R points generated on lines 194–195, otherwise we simply draw the segments on lines 172–181.

```
162     glm::vec3 unitV4V1 = glm::normalize(V4 - V1);
163     glm::vec3 V2V1 = V2 - V1;
164     glm::vec3 V3V1 = V3 - V1;
165 
166     float dV2 = glm::length(V2V1 - glm::dot(V2V1, unitV4V1) * unitV4V1);
167     float dV3 = glm::length(V3V1 - glm::dot(V3V1, unitV4V1) * unitV4V1);
168     float d = (dV2 > dV3) ? dV2 : dV3;
169 
170     if ( d < epsilon )
171     {
172         glBegin(GL_LINES);
173             glVertex3f(V1.x, V1.y, V1.z);
174             glVertex3f(V2.x, V2.y, V2.z);
175 
176             glVertex3f(V2.x, V2.y, V2.z);
177             glVertex3f(V3.x, V3.y, V3.z);
178 
179             glVertex3f(V3.x, V3.y, V3.z);
180             glVertex3f(V4.x, V4.y, V4.z);
181         glEnd();
182     }
183     else
184     {
185         glm::vec3 H = (V2 + V3) / 2.0f;
186         glm::vec3 L[4];
187         glm::vec3 R[4];
188 
189         L[0] = V1;                      R[0] = V4;
190         L[1] = (V1 + V2) / 2.0f;        R[1] = (V3 + V4) / 2.0f;
191         L[2] = (H  + L[1]) / 2.0f;      R[2] = (H  + R[1]) / 2.0f;
192         L[3] = (L[2] + R[2]) / 2.0f;    R[3] = (L[2] + R[2]) / 2.0f;
193 
194         subdivide(L[0], L[1], L[2], L[3], epsilon);
195         subdivide(R[0], R[1], R[2], R[3], epsilon);
196     }
```

Figure 6: Code excerpt showing the subdivision algorithm.

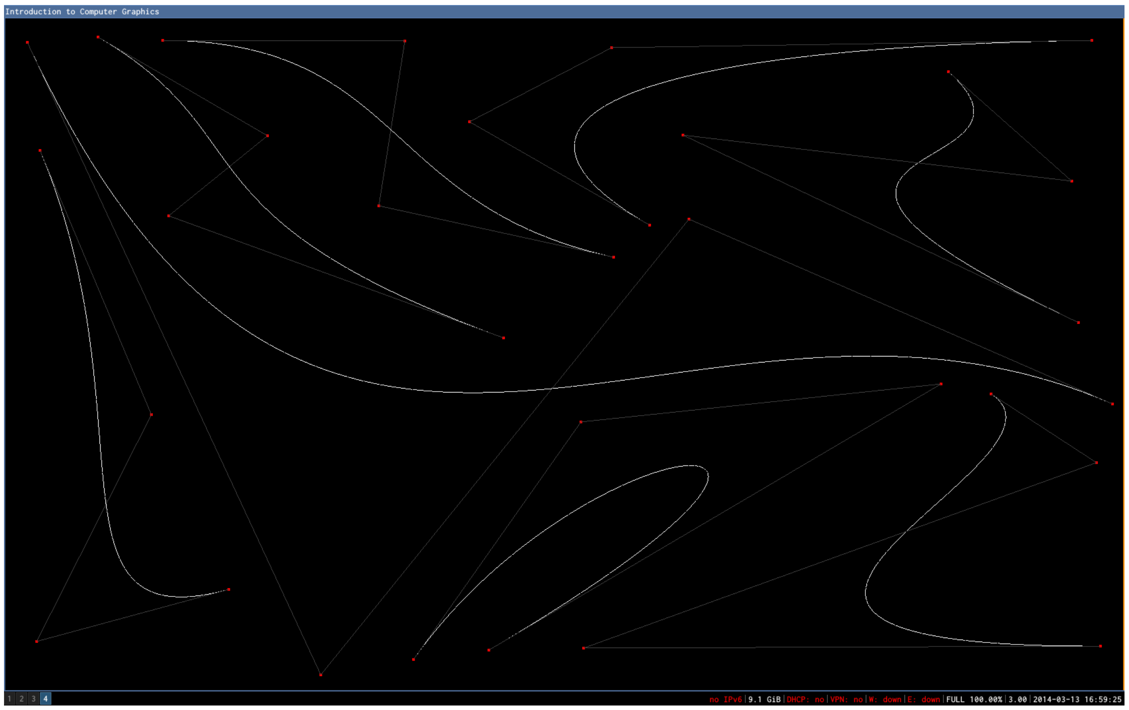
Introduction to Computer Graphics
1 2 3 4
no IPv6 | 9.1 GiB | DHCP: no | VPN: no | W: down | E: down | FULL 100.00% | 3.00 | 2014-03-13 16:59:25

Figure 7: A screenshot showing subdivision of Bezier curves.