An Introduction to 3D Computer Graphics, Stereoscopic Image, and Animation in OpenGL and C/C++

Fore June

Chapter 13 Curves and Surfaces

Curves and surfaces are important in graphics design. We can create an interesting graphics object by simply revolving a simple curve around an axis. Skies and terrains, important features of a video game, can be created using surfaces. Historically, ship builders used mathematical models of surfaces to model ships. In the 1960s, automobile and aircraft industries began to apply curve and surface math models to design vehicles. Such techniques are rapidly borrowed by other areas of computer graphics applications such as video games and animated movies. We discuss in this chapter the application of some of these models in computer graphics. The terminology of some terms used in this field is quite confusing. Different authors may use different terms to refer to the same subject or they may define the same thing in different ways. We simply follow the terminology that a majority of people use in the Internet.

13.1 Representation of Curves and Surfaces

There are 3 common ways to represent curves and surfaces, namely, explicit, implicit and parametric representations. Each representation has some advantages and disadvantages. In computer graphics, the parametric representation is more commonnly used as it can generate a curve or a surface by varying one or two parameters.

13.1.1 Explicit Representation

In the explicit representation, the coordinates of a point on a curve is represented as a funtcion of the other coordinate. In general, if $P = (x, y)$ is a point on a 2D curve, the y coordinate is expressed as a function of the x coordinate:

$$\textbf{2D Curve}: \quad y = f(x) \tag{13.1}$$

Similarly, the y and z coordinates of a point $P = (x, y, z)$ on a 3D curve is expressed as functions of the x coordinate:

$$\textbf{3D Curve}: \quad \begin{aligned} y &= f(x) \\ z &= g(x) \end{aligned} \tag{13.2}$$

For example, a 2D line and a 2D circle are given by:

$$\begin{aligned} &\text{2D line:} && y = mx + b \\ &\text{2D circle segment:} && y = \sqrt{r^2 - x^2} \qquad\qquad 0 \le |x| < r \end{aligned} \tag{13.3}$$

An explicit 3D surface is represented as a function of two variables:

$$\textbf{3D Surface}: \quad z = f(x, y) \tag{13.4}$$

For example, the upper hemisphere surface of a sphere centered at the origin is given by:

$$\text{3D hemisphere surface:} \quad z = \sqrt{r^2 - x^2 - y^2} \qquad\qquad x^2 + y^2 \le r \tag{13.5}$$

In this representation, it is simple to compute the points and plot them and it is also easy to check whether a point lies on the curve. However, the representation has a few disadvantages. Firstly, it is not possible to get multiple values for a single x. In many situations, we need to break curves like circles and ellipses into segments. For example, a complete circle centered at the origin is composed of two segments:

$$y = \sqrt{r^2 - x^2} \qquad y = -\sqrt{r^2 - x^2} \tag{13.6}$$

Secondly, the form has problem representing curves with infinite slope (i.e. vertical tangent). In particular, if $f(x)$ is a polynomial, it is impossible to represent such a curve with infinite slope. Thirdly, it is not invariant under an affine transformation; its form may be changed under such a transformation. A curve or a surface may not have an explicit representation. This representation is rarely used in computer graphics.

13.1.2 Implicit Representation

In the implicit representation, the (x, y) coordinates of a point on a 2D curve satisfy an equation of the form

$$\textbf{2D Curve}: \quad F(x, y) = 0 \tag{13.7}$$

For example, a 2D line and a 2D circle can be represented as:

$$\begin{array}{ll} \text{2D line:} & ax + by + c = 0 \\ \text{2D circle:} & x^2 + y^2 - r^2 = 0 \end{array} \tag{13.8}$$

An implicit surface in 3D space has the form

$$\textbf{3D Surface}: \quad F(x, y, z) = 0 \tag{13.9}$$

For example, a plane and a spherical surface can be expressed as:

$$\begin{array}{ll} \text{A plane:} & ax + by + cz + d = 0 \\ \text{A spherical surface:} & x^2 + y^2 + z^2 - r^2 = 0 \end{array} \tag{13.10}$$

Actually, we can represent a curve as the intersection of two surfaces:

$$\begin{array}{l} F(x, y, z) = 0 \\ G(x, y, z) = 0 \end{array} \tag{13.11}$$

If we can solve a variable, say z of the implicit equation of (13.9) as a function of the other two variables, then we obtain an explicit representation of the surface, $z = f(x, y)$. This is always possible at least locally when $\dfrac{\partial F}{\partial z} \neq 0$.

Sometimes the function F is referred to as an inside-outside function because we can easily determine whether a point is inside or outside the curve or the surface. In 3D space, we have

1. $F(x, y, z) = 0$ for all (x, y, z) on the surface,
2. $F(x, y, z) > 0$ for all (x, y, z) outside the surface, and
3. $F(x, y, z) < 0$ for all (x, y, z) insdie the surface.

In implicit form, we can represent curves with infinite slope, as well as closed and multivalued curves such as a circle or an ellipse. On the other hand, when we join curve segments together, it is difficult to determine whether their tangent directions agree at the joint points. Like explicit representation, an implicit function is not invariant under an affine transformation. Also, some implicit curves are difficult to trace.

If the function $F(x, y, z)$ of (13.9) is a polynomial of the x, y, z, then the function represents algebraic surfaces. Of particular importance are the quadratic surface or quadrics where each term in $F(x, y, z)$ can have degree up to 2 (e.g. x, xy, or z^2 but not xy^2):

$$ax^2 + by^2 + cz^2 + dxy + eyz + hxz + kx + ly + mz + n = 0 \tag{13.12}$$

The web site at *http://wims.unice.fr/gallery/* has an interesting gallery of animated algebraic surfaces. Ellipsoid, elliptic cone and elliptic cylinder with special cases of sphere, circular cone and circular cylinder respectively are examples of quadrics:

$$\begin{array}{ll} \text{Ellipsoid:} & \dfrac{x^2}{a^2} + \dfrac{y^2}{b^2} + \dfrac{z^2}{c^2} - 1 = 0 \\ \text{Elliptic Cylinder:} & \dfrac{x^2}{a^2} + \dfrac{y^2}{b^2} - 1 = 0 \\ \text{Elliptic Cone:} & \dfrac{x^2}{a^2} + \dfrac{y^2}{b^2} - z^2 = 0 \end{array} \tag{13.13}$$

Quadrics are useful in modeling objects. Some researchers claim that 85% of manufactured objects can be modeled using quadrics. An implicit curve or surface in general is hard to render as we have to solve a set of non-linear equations but an algebraic surface usually can be rendered more efficiently than an arbitrary surface.

13.1.3 Parametric Representation

In this representation, we express each coordinate of a point on a curve in terms of an independent variable, u, the parameter. A point in 3D space is expressed as

$$P(u) = \begin{pmatrix} x(u) \\ y(u) \\ z(u) \end{pmatrix} \qquad u \in [u_1, u_2]. \tag{13.14}$$

The form is the same for two and three dimensions. A useful interpretation of the parametric form is to visualize the locus of points $P(u)$ being drawn as the parameter u varies. Sometimes we may even interpret u as a time variable t. The spatial variables $x(u), y(u)$, and $z(u)$ are usually polynomial or rational functions in u and $u \in [u_1, u_2]$, where u_1, and u_2 are real numbers.

For example, a 3D helix curve with radius a and rises $2\pi b$ units per turn can be described by the parametric equations,

$$\begin{array}{ll} x(u) = & a \cos(u) \\ y(u) = & a \sin(u) \\ z(u) = & bu \end{array} \tag{13.15}$$

In this example, $u \in [0, 2\pi]$. Figure 13-1 below shows a helix curve.

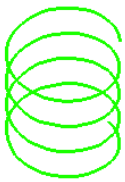

Figure 13-1 Parametric Helix Curve

To describe a 3D surface, we need two parameters, u and v. We express the coordinates of a point $P = (x, y, z)$ of a surface patch as a function of parameters u and v in a closed rectangle:

$$\begin{aligned} x &= x(u, v) \\ y &= y(u, v) \\ z &= z(u, v) \end{aligned} \tag{13.16}$$

with $u_1 \leq u \leq u_2$, and $v_1 \leq v \leq v_2$. For example, we can express a torus with major radius R and minor radius r parametrically as

$$\begin{aligned} x &= cos(u)(R + rcos(v)) \\ y &= sin(u)(R + rcos(v)) \\ z &= rsin(v) \end{aligned} \tag{13.17}$$

for $u, v \in [0, 2\pi]$. Figure 13-2 below shows a torus with $R = 2$ and $r = 0.6$; the diagram on the right side of the figure shows the relations between parameters (u, v) and the (x, y, z) coordinates.

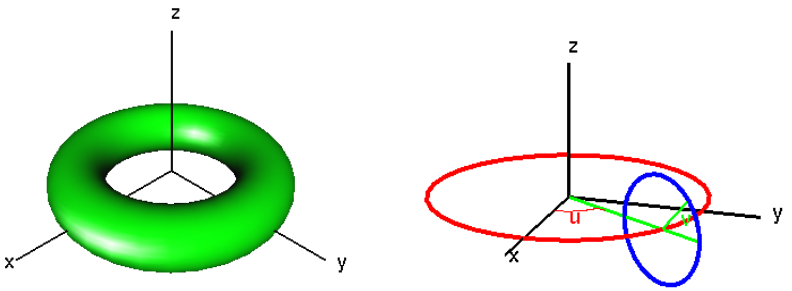

Figure 13-2 Torus with $R = 2, r = 0.6$.

13.1.4 Geometric Continuity

The smoothness of a parametric curve relates to its continuity. We say that a parametric curve $f(u)$ has C^k continuity if its $k\text{-}th$ derivative,

$$\frac{d^k f(u)}{du^k}$$

exists and is continuous throughout the curve. For example, a curve that describes the trajectory of the motion of an object with a parameter of time, must have C^1 continuity for the object to have finite acceleration.

The continuity of the piecewise curve determines how the curve segments join at the joints. We can describe the various order of parametric continuity as follows:

- C^{-1}: Curves consist of discontinuities. They may not be joined.
- C^0: Curves are joined.
- C^1: First derivatives are continuous.
- C^2: First and second derivatives are continuous.
- C^k: First through k-th derivatives are continuous.

In real applications, we may have to join a number of small curves to form a long curve to represent the profile of an object. The point where two curve segments meet within a piecewise curve is referred to as a *breakpoint*. If the curve segments have the same k-th derivative at the breakpoint, then the curve has C^k continuity. While computer graphics has relied heavily on mathematical descriptions of point sets based on parametric functions, we need a different notion, *geometric continuity* to describe the smoothness of curves and surfaces.

We have learned that two C^k functions join smoothly at a boundary to form a joint C^k function if, at all common points, their i-th derivatives agree for $i = 0, 1, ..., k$. However, it is neither necessary nor sufficient to characterize the smoothness of curves or surfaces by the continuities of the derivatives of the component functions. As an example, consider the piecewise curve formed by three curve segments as shown in Figure 13-3 below:

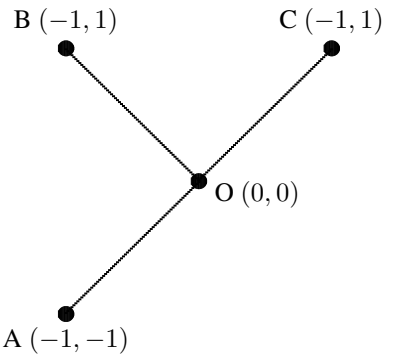

Figure 13-3 Geometric Continuity

We can parameterize the curve segments OB and OC with two parabolic arcs with equal derivatives at the point O:

$$\begin{aligned} P_{BO}(u) &= (1-u)^2 B + 2(1-u)uO + u^2 O \qquad u \in [0,1] \\ P_{OC}(u) &= (1-u)^2 O + 2(1-u)uO + u^2 C \qquad u \in [0,1] \end{aligned} \tag{13.18}$$

We see that the curve is C^1 continuous at the breakpoint O with $P_{BO}(1) = P_{OC}(0) = O$ and $\frac{dP_{BO}(u)}{du}\Big|_{u=1} = \frac{dP_{OC}(u)}{du}\Big|_{u=0} = O$ However, the two segments are not joining smoothly at the joint point O, showing that matching derivatives do not always imply smoothness. On the other hand, smoothness does not necessarily imply matching derivatives. We can similarly parameterize AO and OC with two parabolic arcs with unequal derivatives at the joint point O, even though the shape is geometrically continuous.

To study the smoothness of curves and surfaces, people define k-th order geometric continuity, G^k, as agreement of derivatives after suitable reparametrization. The geometric continuity can be considered as a relaxation of parametrization but not as a relaxation of smoothness. For $k \leq 2$, it has the following properties:

- G^0: Curves are joined. There may be a sharp turn at where they meet.
- G^1: First derivatives are continuous. Two curve segments have identical tangents at the breakpoint; they join smoothly.
- G^2: First and second derivatives are continuous. Two curve segments have identical curvature at the breakpoint. (Curvature is the rate of change of the tangents.)

In general, G^k continuity exists if we can reparameterize the curves to have C^k continuity. A reparametrization of a curve only affects the parameters but the reparametrized curve is geometrically identical to the original.

13.2 Interpolation

13.2.1 Polynomial Parametric Curves

In parametric representation, a popular approach is to represent a curve or a surface as a polynomial of the parameters. A polynomial parametric curve of degree n (= $order - 1$ in OpenGL), is of the form

$$p(u) = \sum_{k=0}^{n} u^k c_k \tag{13.19}$$

where each c_k has independent x, y, z components. That is,

$$p(u) = \begin{pmatrix} x(u) \\ y(u) \\ z(u) \end{pmatrix}, \quad c_k = \begin{pmatrix} c_{kx} \\ c_{ky} \\ c_{kz} \end{pmatrix} \tag{13.20}$$

A parametric polynomial surface is similarly defined by two parameters, u and v in the form

$$p(u) = \begin{pmatrix} x(u,v) \\ y(u,v) \\ z(u,v) \end{pmatrix} = \sum_{i=0}^{n}\sum_{j=0}^{m} c_{ij} u^i v^j \tag{13.21}$$

In particular, the commonly used parametric cubic polynomial curves have the form

$$p(u) = \sum_{k=0}^{3} c_k u^k = c_0 + c_1 u + c_2 u^2 + c_3 u^3 \tag{13.22}$$

We can express this in matrix form

$$p(u) = \begin{pmatrix} x(u) \\ y(u) \\ z(u) \end{pmatrix} = \begin{pmatrix} c_{0x} & c_{1x} & c_{2x} & c_{3x} \\ c_{0y} & c_{1y} & c_{2y} & c_{3y} \\ c_{0z} & c_{1z} & c_{2z} & c_{3z} \end{pmatrix} \begin{pmatrix} 1 \\ u \\ u^2 \\ u^3 \end{pmatrix} \tag{13.23a}$$

Or equivalently,

$$p(u) = \begin{pmatrix} x(u) & y(u) & z(u) \end{pmatrix} = \begin{pmatrix} 1 & u & u^2 & u^3 \end{pmatrix} \begin{pmatrix} c_{0x} & c_{0y} & c_{0z} \\ c_{1x} & c_{1y} & c_{1z} \\ c_{2x} & c_{2y} & c_{2z} \\ c_{3x} & c_{3y} & c_{3z} \end{pmatrix} \tag{13.23b}$$

13.2.2 Interpolation Polynomial

An interpolation polynomial is of the form

$$f(x) = \sum_{k=0}^{n} a_k x^k = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x_1 + a_0 \tag{13.24}$$

If $f(x)$ interpolates the data points (x_i, y_i), then

$$f(x_i) = y_i \qquad \text{for all } i \in 0, 1, ..., n$$

That is, the polynomial curve passes through the data points. We can find the coefficients a_i using a Vandermonde matrix, which is a matrix with the terms of a geometric progression in each row.

Cubic parametric interpolating polynomial is a commonly used parametric curve that interpolates four control points ($n = 4$). By adjusting the four control points, we obtain different curves. Suppose the four control points are P_0, P_1, P_2, and P_3, where

$$P_k = \begin{pmatrix} x_k & y_k & z_k \end{pmatrix} \tag{13.25}$$

In this case, the curve is described by

$$p(u) = c_0 + c_1 u + c_2 u^2 + c_3 u^3 \qquad 0 \leq u \leq 1 \tag{13.26}$$

To solve for (u), we have to find the coefficients c_i so that the polynomial $p(u)$ passes through (interpolates) the four control points. An easy way to find $p(u)$ is to specify the four control points, P_0, P_1, P_2, and P_3 at $u = 0, 1/3, 2/3$, and 1 respectively. That is,

$$\begin{aligned} P_0 &= p(0) = c_0 \\ P_1 &= p(\tfrac{1}{3}) = c_0 + c_1(\tfrac{1}{3}) + c_2(\tfrac{1}{3})^2 + c_3(\tfrac{1}{3})^3 \\ P_2 &= p(\tfrac{2}{3}) = c_0 + c_1(\tfrac{2}{3}) + c_2(\tfrac{2}{3})^2 + c_3(\tfrac{2}{3})^3 \\ P_3 &= p(1) = c_0 + c_1 + c_2 + c_3 \end{aligned} \tag{13.27}$$

We can express this in matrix form:

$$P = AC \tag{13.28}$$

where

$$\begin{pmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{pmatrix} = \begin{pmatrix} x_0 & y_0 & z_0 \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{pmatrix} \tag{13.29}$$

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & \frac{1}{3} & (\frac{1}{3})^2 & (\frac{1}{3})^3 \\ 1 & \frac{2}{3} & (\frac{2}{3})^2 & (\frac{2}{3})^3 \\ 1 & 1 & 1 & 1 \end{pmatrix} \qquad C = \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} c_{0x} & c_{0y} & c_{0z} \\ c_{1x} & c_{1y} & c_{1z} \\ c_{2x} & c_{2y} & c_{2z} \\ c_{3x} & c_{3y} & c_{3z} \end{pmatrix} \tag{13.31}$$

The inverse of matrix A can be calculated and is

$$A^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -5.5 & 9 & -4.5 & 1 \\ 9 & -22.5 & 18 & -4.5 \\ -4.5 & 13.5 & -13.5 & 4.5 \end{pmatrix} \tag{13.32}$$

As P in (13.28) consists of control points which are known, we can solve for the coefficients C by

$$C = A^{-1}P \tag{13.33}$$

The curve $p(u)$ of (13.26) can be expressed as

$$p(u) = UC \tag{13.34}$$

where

$$U = \begin{pmatrix} 1 & u & u^2 & u^3 \end{pmatrix} \tag{13.35}$$

Example

Find the cubic polynomial curve $p(u),\ u \in [0, 1]$ that interpolates the points $(0, 0, 0), (1, 2, 2), (2, 3, 4), (4, 5, 3)$. What is $p(0.8)$?

Solutions:

We let the curve $p(u)$ pass through the points at $u = 0, 1/3, 2/3, 1$. Then

$$C = A^{-1}P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -5.5 & 9 & -4.5 & 1 \\ 9 & -22.5 & 18 & -4.5 \\ -4.5 & 13.5 & -13.5 & 4.5 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 \\ 1 & 2 & 2 \\ 2 & 3 & 4 \\ 4 & 5 & 3 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 4 & 9.5 & 3 \\ -4.5 & -13.5 & 13.5 \\ 4.5 & 9 & -13.5 \end{pmatrix}$$

Thus

$$p(u) = UC = \begin{pmatrix} 1 & u & u^2 & u^3 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 \\ 4 & 9.5 & 3 \\ -4.5 & -13.5 & 13.5 \\ 4.5 & 9 & -13.5 \end{pmatrix} \tag{13.36}$$

When $u = 0.8$, we have

$$\begin{aligned} p(0.8) &= \begin{pmatrix} 1 & 0.8 & 0.64 & 0.512 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 \\ 4 & 9.5 & 3 \\ -4.5 & -13.5 & 13.5 \\ 4.5 & 9 & -13.5 \end{pmatrix} \\ &= \begin{pmatrix} 2.624 & 3.568 & 4.128 \end{pmatrix} \end{aligned}$$

The cubic polynomial curve of this example is shown in Figure 13-4 below.

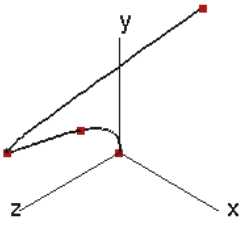

Figure 13-4 Cubic Polynomial Interpolation

13.2.3 Lagrange's Method

Langrange interpolation is a simplest form of numerical solution for polynomial interpolation. Though it is simple, the method is susceptible to Runge's phenomenon, which is a problem of oscillation at the edges of an interval caused by polynomial interpolation with high degree polynomials. Also, when an interpolation point is changed, the method requires recalculating the entire interpolant.

The Lagrange interpolating polynomial is the polynomial $y(x)$ of degree $\leq (n-1)$ passing through the set of n points $\{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$ with $y(x_i) = y_i$, and no two x_i are the same. It is given by

$$y(x) = \sum_{i=1}^{n} y_i \frac{\prod_{j \neq i}^{n} (x - x_j)}{\prod_{j \neq i}^{n} (x_i - x_j)} \tag{13.37}$$

The following function **polyint**() shows an implementation of this method. The class *Point3*, which has been discussed in Chapter 12, is a class that defines a 3D point.

```
//Langrange polynomial interpretation for N points
double polyint ( Point3  points[], double x, int N )
{
  double y;

  double num = 1.0;      //numerator
  double den = 1.0;      //denominator
  double sum = 0.0;

  for ( int i = 0; i < N; ++i ) {
    num = den = 1.0;
    for ( int j = 0; j < N; ++j ) {
      if ( j == i ) continue;
      num = num * ( x - points[j].x );              //x - xj
    }
    for ( int j = 0; j < N; ++j ) {
      if ( j == i ) continue;
      den = den * ( points[i].x - points[j].x );    //xi - xj
    }
    sum += num / den * points[i].y;
  }
  y = sum;

  return y;
}
```

13.2.4 Neville's Algorithm and Barycentric Formula

The Lagrange interpolation is simple and its implementation is straightforward but it is quite inefficient when the degree of the polynomial becomes large as it involves $O(n^2)$ multiplications. Neville's algorithm improves Lagrange method by calculating values recursively, which can be easily extended to calculate derivative values. The Barycentric formula extends the idea further, which only requires $O(n)$ multiplications in the evaluation of $y(x)$. In this method, we let

$$\begin{aligned} l(x) &= (x-x_0)(x-x_1)\cdots(x-x_n) \\ w_i &= \frac{1}{\prod_{j\neq i}^{n}(x_i-x_j)}, \quad i=0,\cdots,n \\ l_i(x) &= l(x)\frac{w_i}{(x-x_i)} \end{aligned} \tag{13.38}$$

where w_i are referred to as barycentric weights. Then equation (13.37) can be rewritten as

$$y(x) = \sum_{i=0}^{n} l_i y_i = l(x)\sum_{i=0}^{n}\frac{w_i}{(x-x_i)}y_i \tag{13.39}$$

The formula requires $O(n^2)$ operations for calculating some quantities and the numbers w_i, but once the quantities are obtained, it only requires $O(n)$ operations to evaluate $y(x)$. The following code segment shows an implementation of this method.

```
const int N = ..;
double w[N];

void calculate_weights( Point3 points[] )
{
  for ( int i = 0; i < N; i++ ) {
    w[i] = 1.0;
    for ( int j = 0; j < N; j++ ) {
      if ( j == i ) continue;
        w[i] /= points[i].x - points[j].x;
    }
  }
}

//Barycentric polynomial interpretation for N points
double barycentric ( Point3  points[], double x )
{
  double sum = 0.0, lx = 1.0, y;

  for ( int i = 0; i < N; i++ ) {
    if ( x == points[i].x )
      return points[i].y;
    lx *= x - points[i].x;
  }
```

```
    for ( int i = 0; i < N; i++ )
      sum += w[i] * points[i].y / ( x - points[i].x );

    y = lx * sum;
    return y;
}
```

Another simple method of interpolation based on polynomials is the Hermite interpolation which extends the basic polynomial interpolation to consider both the data points and the derivatives (slopes) at the the data points. Hermite interpolation is simple and efficient but it may not be very effective as sometimes a small change of the interpolating curve requires a large variation of the magnitude of the tangent vector.

In a curve interpolation, we generate a curve that passes through all of the control points. Under the constraint of restricting the curve to pass through a set of data points, we have too little local control of the curve. In many applications, we do not require the curve passing through all of the control points. All we need is that the curve passes through the first and the last control points; other points are used to shape the curve. This gives us a lot more control in shaping the curve by varying the control points. We can imagine that the shape of such as curve is obtained by fixing the two ends of an elastic magnetic string on a table. We then fix some small magnets at various positions of the table. The magnets attract the elastic string towards them and thus generate a curved shape. The magnets act like control points in a curve design algorithm. A curve generated by a set of control points but does not necessarily pass through all of them is called an *approximating curve*. In the following sections, we discuss some common algorithms used by designers to produce such a curve.

13.3 Bezier Curves and Surfaces

13.3.1 Bezier Curve

A commonly used family of approximating curves is the spline curves or splines for short. A spline curve is a smooth curve specified succinctly by a few control points. This term originates from traditional drafting design, where a spline is a thin strip, which is held in place by weights to create a curve for tracing. In a similar way we generate a curve using a set of control points.

Bezier curves and B-spline curves are two main classes of splines. Bezier curve is named after the French Pierre Bezier, who developed the method for the body design of the Renault car in the 1970s. Given a set of of $n+1$ points $P_0, P_1, ..., P_n$, the Bezier parametric curve is of the form

$$p(u) = \sum_{k=0}^{n} P_k B_{k,n}(u) \qquad u \in [0,1] \tag{13.40}$$

where $B_{k,n}(u)$ is a Bernstein polynomial given by

$$B_{k,n}(u) = \frac{n!}{k!(n-k)!} u^k (1-u)^{n-k} \qquad u \in [0,1] \tag{13.41}$$

Note that

$$\sum_{k=0}^{n} B_{k,n}(u) = \sum_{k=0}^{n} \binom{n}{k} u^k (1-u)^{n-k} = (u + (1-u))^n = 1^n = 1 \tag{13.42}$$

Thus Equation (13.40) is an affine combination of points and gives a valid point. The curve always passes through the first control point ($p(0) = P_0$) and the last control point ($p(1) = P_n$). Because $B_{k,n}(u) \geq 0$ for $u \in [0, 1]$, the curve always lies within the convex hull of the control points. Also, the curve is tangent to $P_1 - P_0$ and $P_n - P_{n-1}$ at the end points.

In general, $B_{k,n}$ is called a blending function as it 'blends' the control points to form the Bezier curve; its degree is always one less than the number of control points. We can generate closed curves by making the last control point P_n the same as the first one P_0.

A rational Bezier curve adds adjustable weights to provide better approximation to arbitrary shapes; it is defined by

$$p(u) = \frac{\sum_{k=0}^{n} B_{k,m}(u) w_k P_k}{\sum_{k=0}^{n} B_{k,m}(u) w_k} \qquad u \in [0, 1] \tag{13.43}$$

where $B_{k,m}$ is a Bernstein polynomial with degree m, P_k are the control points, and the weight w_k of P_k is the last ordinate (w) of the homogeneous point P_k^w.

13.3.2 Cubic Bezier Curve

The most commonly used Bezier curves are the degree three Bezier Curves, also known as cubic Bezier curves, which are defined by four control points.Two of these are the end points of the curve, while the other two effectively define the gradients at the end points, pulling the curve towards them.

From equations (13.40) and (13.41), a degree 3 (four control points) Bezier curve is given by

$$p(u) = B_0(u)P_0 + B_1(u)P_1 + B_2(u)P_2 + B_3(u)P_3 \qquad u \in [0, 1] \tag{13.44}$$

where

$$B_k(u) = B_{k,3}(u) = \binom{3}{k} u^k (1-u)^{3-k} = \frac{3!}{k!(3-k)!} u^k (1-u)^{3-k} \tag{13.45}$$

are the blending functions for the curve. In explicit form,

$$\begin{aligned} B_0 &= (1-u)^3 & B_1 &= 3u(1-u)^2 \\ B_2 &= 3u^2(1-u) & B_3 &= u^3 \end{aligned} \tag{13.46}$$

The figure below presents some examples of degree three Bezier curves, showing how the control points shape the curves.

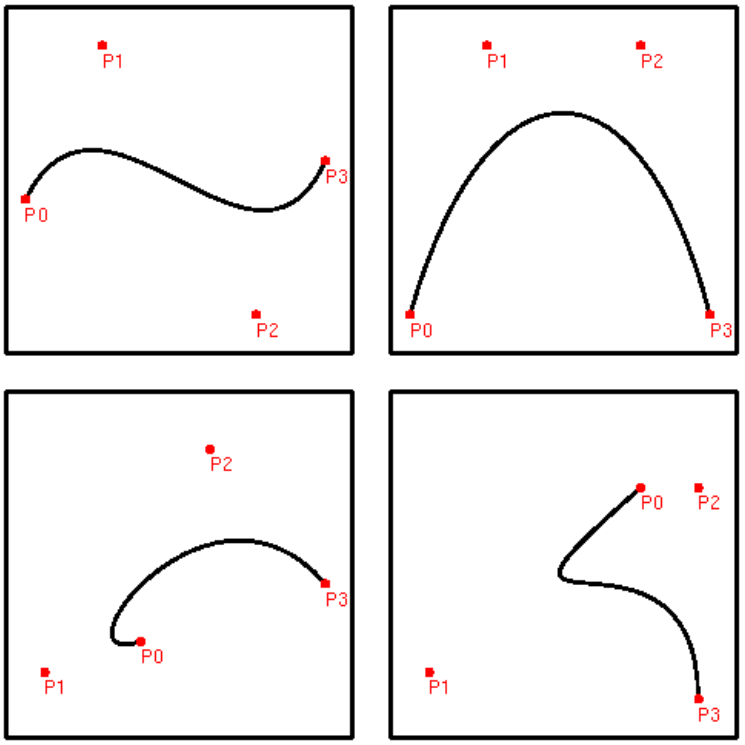

Figure 13-5 Cubic Bezier Curves (Degree 3)

Figure 13-6 below shows a plot of the blending functions for $u \in [0, 1]$; at any value of u, $B_0 + B_1 + B_2 + B_3 = 1$.

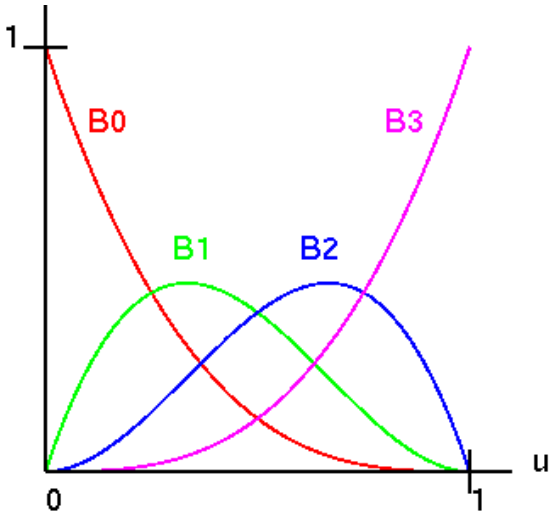

Figure 13-6 Blending Functions for Bezier Curves of Degree 3

If we expand (13.46) and substitute the expanded terms into (13.44), we can express

$p(u)$ explicitly as

$$p(u) = (1 - 3u + 3u^2 - u^3)P_0 + (3u - 6u^2 + 3u^3)P_1 + (3u^2 - 3u^3)P_2 + u^3 P_3 \quad (13.47)$$

or in matrix form

$$p(u) = \begin{pmatrix} 1 & u & u^2 & u^3 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{pmatrix} \begin{pmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{pmatrix} \quad (13.48)$$

The derivative of $p(u)$ is given by

$$p'(u) = \frac{dp(u)}{du} = \begin{pmatrix} 0 & 1 & 2u & 3u^2 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{pmatrix} \begin{pmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{pmatrix} \quad (13.49)$$

Therefore,

$$p'(0) = 3(P_1 - P_0) \qquad p'(1) = 3(P_3 - P_2) \quad (13.50)$$

This means that the curve $p(u)$ starts at $u = 0$, traveling in the direction of the vector from P_0 to P_1 and at the end, it travels in the direction of P_2 to P_3.

The following code segment shows an implementation of cubic Bezier curves.

```
//blending functions with degree 3; 0 <= k <=3, 0 <= u <= 1
float blend3 ( float u, int k )
{
  float b = 1;
  switch ( k ) {
    case 0:
        for ( int i = 0; i < 3; i++ )
          b *= ( 1 - u );
        break;
    case 1:
        b = 3 * u * ( 1 - u ) * ( 1 - u );
        break;
    case 2:
        b = 3 * u * u * ( 1 - u );
        break;
    case 3:
        b = u * u * u;
        break;
  }
  return b;
}

//blending functions with degree n (n+1 control points), 0 <= u <= 1
float blend ( float u, int n, int k )
{
  float b = 1;
  if ( n <= 0 ||  n < k )
```

```
    return b;

  int j = n - k;
  float u1 = 1 - u;
  for ( int i = 1; i <=n; i++ ) {
    b *= i;
    if ( k >= 1 ) {
      b *= u / k;
      k--;
    }
    if ( j >= 1 ) {
      b *= u1 / j;
      j--;
    }
  }

  return b;
}

//render the  curve
void display(void)
{
   float x, y, z;
   int i, j, k;

   Point3 data[4], *p[4];
   float B[4], u;
   ..............  //data[] contains four control points
   glBegin(GL_LINE_STRIP);
      for (i = 0; i <= 80; i++) {
         u = (float) i / 80.0;
         for ( k = 0; k < 4; k++ ) {
           //B[k] = blend3 ( u, k );
           B[k] = blend ( u, 3, k );  //degree 3 blending functions
           p[k] = (Point3 *) &data[k];
         }
         x = y = z = 0;
         for ( k = 0; k < 4; k++ ){
           x += B[k] * p[k]->x;
           y += B[k] * p[k]->y;
           z += B[k] * p[k]->z;
         }
         glVertex3f ( x, y, z );
      }
   glEnd();
   glFlush();
}
```

Although a cubic Bezier curve is simple and easy to compute, it is not possible to use it to closely approximate a curve that has many turns as a cubic Bezier curve only uses four control points. A common practice is to break a long curve into a number of segments,

each defined by a separate cubic Bezier curve and joined to another one to form a piecewise curve. Figure 13-7 shows an example of joining multiple cubic curves to approximate a curve fit to an arbitrary number of data points.

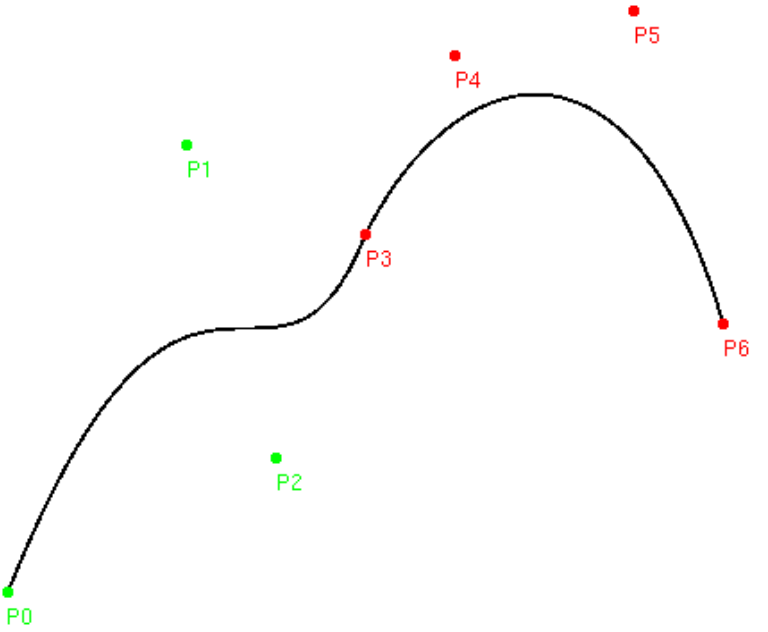

Figure 13-7 Joining Cubic Bezier Curves to Form Piecewise Curve

13.3.3 Bezier Surface

By blending functions of two orthogonal Bezier curves, we can form a Bezier surface, which can be also defined as a parametric surface. We can specify an order $(n+1)(m+1)$ Bezier surface using two parameters, u and v with $(n+1)(m+1)$ control points P_{ij}:

$$p(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_{i,n}(u) B_{j,m}(v) P_{ij} \qquad u,v \in [0,1] \tag{13.51}$$

where $B_{k,n}$ are the Bernstein polynomials given by (13.41).

The corresponding properties of Bezier curve apply to Bezier surface. For example, the surface is contained within the convex hull of the control points, and in general does not pass through all the control points except for the corners of the control point grid. We can form closed surfaces by setting the last control point to be the same point as the first. If the tangents of the first two control points are also equal to that of the last two control points then the closed surface will have first order continuity.

Using Bezier surfaces to construct patch meshes is better than constructing meshes using polygons such as triangles because Bezier surfaces are easier to manipulate and are much more compact. Moreover, they have more superior continuity properties. Some common parametric surfaces such as spheres and cylinders can be well approximated by relatively small numbers of cubic ($n = m = 3$) Bezier patches.

However, Bezier patch meshes are difficult to render directly because it is difficult to calculate their intersections with lines. It is also difficult to combine them directly with

perspective projection algorithms. Therefore, Bezier patch meshes are in general decomposed into meshes of flat triangles in the 3D graphics processing pipeline.

13.4 B-Splines

13.4.1 B-Spline Properties

In many computer applications such as computer aided design (CAD), we may need to construct curves which are long and have complicated shapes. One approach would be to use high-degree Bezier curves. However, a high-degree Bezier curve is expensive to compute and does not have good local control; each of the $n+1$ control points of a degree n Bezier curve affects the whole curve. When we change the position of one control point, the whole curve will be changed. Another approach is to construct the long curve by joining piecewise cubic Bezier curves with some constraints at the joints. In most cases, the constraint is that the curves are joined smoothly, which in turn requires at least C^1-continuity but desirably C^2-continuity. Piecewise Bezier curves interpolate the control points at the joints (which are end points for the segments) and have local control, but they need not be C^2-continuous. It turns out that piecewise cubic Bezier curves cannot be C^2-continuous and have local control at the same time. This is because C^2-continuous property implies second derivatives at the joined points are continuous; this requires that the two "interior" control points p_1^k, and p_2^k of segment k depend on $p_3^k (= p_0^{k+1})$, which in turn depend on the "interior" control points of the next segment, i.e. there is no local control.

A B-spline, which can be considered as a generalization of the Bezier curve, addresses these problems by using a different approach to represent a complicated curve. A B-spline curve maintains local control but does not interpolate the "interior" control points. We can define a B-spline curve in the following way.

We first define a **knot vector** U (not a geometric 3D vector) as a nondecreasing sequence,

$$U = \{u_0, u_1, ..., u_T\} \tag{13.52}$$

and define n control points $p_0, p_1, ..., p_{n-1}$. We define the order as

$$m = T - n + 1 \qquad m \geq 1 \tag{13.53}$$

Degree d is defined as order minus one (i.e. $d = m - 1 = T - n$). Because $m \geq 1$, we have $T \geq n$. Also, $T = n + m - 1$. We call the entities $u_m, ..., u_T$ *internal knots*, and define the *basis* functions $N_{k,i}$ recursively as

$$N_{k,1}(u) = \begin{cases} 1 & \text{if } u_k < u \leq u_{k+1} \\ 0 & \text{otherwise} \end{cases} \tag{13.54}$$

$$N_{k,m}(u) = \left(\frac{u - u_k}{u_{k+m-1} - u_k}\right) N_{k,m-1}(u) + \left(\frac{u_{k+m} - u}{u_{k+m} - u_{k+1}}\right) N_{k+1,m-1}(u) \tag{13.55}$$

A Non-Uniform Rational B-spline (NURB) is defined by the curve

$$p(u) = \sum_{k=0}^{n-1} p_k N_{k,m}(u) \tag{13.56}$$

Note again that the basis functions sum up to 1 at any u and thus it is legitimate to use them to form a combination of points.

In (13.56), if the first m knots are 0 and the last m knots are equal to 1, then it defines a nonperiodic B-spline . If the internal knots are equally spaced, it is a uniform B-spline . If the B-spline has no internal nodes, it is reduced to a Bezier curve.

B-splines have a number of advantages in graphics design. The following list some of them:

1. A single B-spline can specify a long complicated curve.
2. B-splines can approximate curves with sharp bends and even "corners".
3. We can translate piecewise Bezier curves into B-splines.
4. B-splines act more flexibly and intuitively with a large number of control points.
5. We can have local control of B-splines. That is, changing the placement of a point only affects a small segment of the curve.
6. Compared to Bezier curves, B-splines are a lot more sensitive to the placement of control points.
7. A piecewise Bezier curve requires more control points than a corresponding B-spline curve.

13.4.2 Knot Vector and Basis Functions

As discussed above, an order m (degree $m-1$) B-spline curve with n control points is a continuous function defined by (13.56). In defining the curve, we must specify a knot vector $U = \{u_0, u_1, ..., u_{n+m-1}\}$. Each basis function $N_{k,j}(u)$ depends only on the $j+1$ knot values from u_k to u_{k+j}. $N_{k,j}(u) = 0$ for $u \leq u_k$ or $u > u_{k+j}$. So p_k only influences the curve segment for $u_k < u \leq u_{k+j}$. Actually, $p(u)$ is a polynomial of order j (degree $j-1$) on each interval $u_k < u \leq u_{k+1}$. Across the knots $p(u)$ is C^{j-2}-continuous and $p(u)$ is defined for $u_{min} < u \leq u_{max}$ where $u_{min} = u_j$ and $u_{max} = u_{n+2}$.

Knot vectors are generally classified into three categories: *uniform, open uniform*, and *non-uniform*.

If the knots u_k are equally spaced, the knot vector is a **uniform** knot vector. For example,

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$
$$\{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$$
$$\{-1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0\}$$

are uniform knot vectors.

Open Uniform knot vectors are uniform knot vectors which have j-equal knot values at each end. That is, knot values are equally spaced but the end values are repeated j times:

$$\begin{array}{ll} u_k = u_0, & 0 \leq k < j \\ u_{k+1} - u_k = constant, & j-1 \leq k < n+1 \\ u_k = u_{j+n}, & k \geq n+1, \end{array} \tag{13.57}$$

Examples:

$$\begin{array}{ll} \{0,0,0,1,2,3,4,4,4\} & k=3, n=5 \\ \{1,1,1,1,2,3,4,5,6,6,6\} & k=4, n=7 \\ \{0.1,0.1,0.1,0.1,0.1,0.3,0.5,0.7,0.7,0.7,0.7,0.7\} & k=5, n=7 \end{array}$$

Non-uniform knot vectors are general cases, where the only constraint is the standard $u_k \leq u_{k+1}$.

The shape of an $N_{k,j}$ basis function is determined entirely by the relative spacing between the knots. Scaling ($u'_k = \alpha u_k, \forall k$) or translating ($u'_k = u_k + \Delta u, \forall k$) the knot vector has no effect on the shape of the basis function $N_{k,j}$.

13.4.3 Cubic Uniform B-Splines

From discussions above, we see that when the knots are equidistant, the basis functions are just shifted copies of each other and we say that the B-spline is uniform. If the number of knots is the same as the degree, the B-spline degenerates into a Bezier curve.

Uniform B-splines of degree three, also known as cubic uniform B-splines, are one of the simplest and most useful classes of B-splines. A cubic B-Spline with $n+1$ control points with open uniform knots is given by:

$$p(u) = \sum_{k=0}^{n} p_k N_k(u) \qquad 3 \leq u \leq n+1 \tag{13.58}$$

where p_k's are the control points, and $N_k(u) = N_{k,4}(u)$ are the blending (or basis) functions of degree 3 (order 4). For degree 3, $N_k(u) = 0$ if $u \leq k$ or $u \geq k+4$. (i.e. For any given u value, only 4 basis functions are nonzero; see Figure 13.9 below.) Therefore, (13.58) can be expressed as

$$p(u) = \sum_{k=j-3}^{j} p_k N_k(u) \qquad u \in [j, j+1], \quad 3 \leq j \leq n \tag{13.59}$$

This equation implies that when a single control point p_k is moved, only the portion of the curve $p(u)$ with $k < u < k+4$ will be changed. In other words, we can have local control for such a curve. Another way of understanding (13.59) is that the curve $p(u)$ is composed of a number of curve segments and each segment is controlled by four control points. In general, the blending functions have the following properties:

1. They are translates of each other, i.e. $N_k(u) = N_0(u-k)$.
2. They are piecewise degree three polynomials.
3. They are C^2-continuous, i.e. $N_k(u)$'s have continuous second derivatives.
4. They are partitions of unity, i.e. $\sum N_k(u) = 1$, for $3 \leq u \leq n+1$. This is necessary as $p(u)$ is a valid point.
5. $N_k \geq 0$, (thus $N_k \leq 1$) for all u.
6. They have local properties, i.e. $N_k = 0$ for $u \leq k$ and $k+4 \leq u$.

Figure 13-8 shows how these basis functions are calculated recursively.

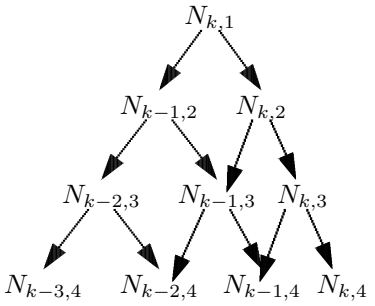

Figure 13-8 Degree 3 Blending Functions ($N_{k,1} = 1, N_i = N_{i,4}$)

Figure 13-9 shows a plot of the first five cubic blending functions with uniform knots. The knot vector is $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. (If open-uniform knots are used, $N_k(u) = 0$ when $u \leq 3$.) We can see that the functions are translates of each other ($N_k(u) = N_0(u + k)$) and $N_k(u)$ is nonzero only when $k < u < k + 4$.

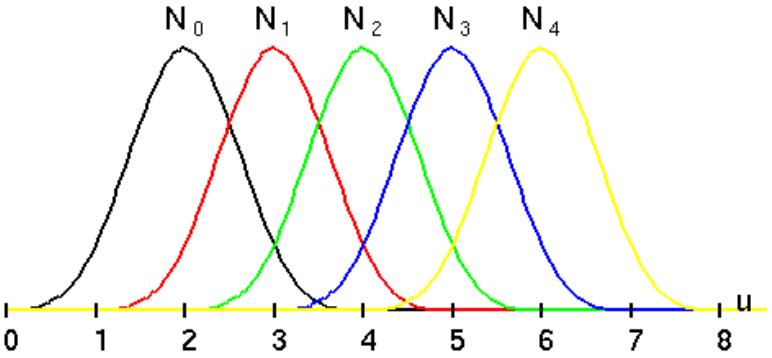

Figure 13-9 Cubic Blending Functions ($\sum_{i=j}^{j+3} N_i(u) = 1$)

13.4.4 Non Uniform Rational B-Splines (NURB)

The non uniform rational B-Spline (NURB) is the most general form of a B-Spline. An order m (or degree $m - 1$) NURB with n control points is given by (13.56). A uniform B-spline is simply a special form of NURB.

We denote the knot vector used in calculating the curve values of a NURB as $U = \{u_0, u_1, ..., u_{n+m-1}\}$. The knot vector divides the parametric space into intervals, which are usually referred to as knot spans. The number of knots in a knot vector is always equal to the number of control points plus the B-Spline order. The values in the knot vector should be in nondecreasing order. Each time the value of the parameter u gets to a new knot span, a new control point will be used and an old control point will be dropped in calculating the current $p(u)$. The steps below show a standard simple way of setting a knot vector of an order m NURB with n control points:

1. The knot vector U has totally $n + m$ knots, denoted as $U = \{u_0, ..., u_{n+m-1}\}$.
2. The values of the first m knots, $u_0, ..., u_{m-1}$ are all equal to 0.
3. The next $n - m$ knots $u_m, ..., u_{n-1}$ increments in 1, from 1 to $n - m$.

4. The final m knots, $u_n, ..., u_{n+m-1}$ are all equal to $n - m + 1$.

Examples

1. Knot vector of 8 control points $(n = 8)$, order $4(m = 4)$:
   $u_0 = u_1 = u_2 = u_3 = 0$
   $u_4 = 1, u_5 = 2, u_6 = 3, u_7 = 4$
   $u_8 = u_9 = u_{10} = u_{11} = 5$

2. Knot vector of 7 control points $(n = 8)$, order $5(m = 5)$:
   $u_0 = u_1 = u_2 = u_3 = u_4 = 0$
   $u_5 = 1, u_6 = 2$
   $u_7 = u_8 = u_9 = u_{10} = u_{11} = 3$

The following code segment shows the function **setKnotVector** that builds a knot vector using this method, the function **N_k** that calculates the blending functions, and the function **nurb** that finds a point using NURB.

```
//order m; n control points; return knots in U[]
void setKnotVector ( int m, int n, float U[] )
{
    if ( n < m  ) return;     //not enough control points
    for ( int i = 0; i < n + m; ++i ){
       if (i < m) U[i] = 0.0;
       else if (i < n) U[i] = i-m+1;  //i is at least m here
       else U[i] = n - m + 1;
    }
}

//order m blending functions recurvsively, U[] holds knots
float N_k ( int k, int m, float u, float U[] )
{
  float d1, d2, sum = 0.0;

  if ( m == 1 )
  return ( U[k] < u &&  u <= U[k+1] );   //1 or 0

  //m larger than 1, so evaluate recursively
  d1 = U[k+m-1] - U[k];
  if ( d1 != 0 )
    sum = (u - U[k]) * N_k(k,m-1,u, U) / d1;
  d2 = U[k+m] - U[k+1];
  if ( d2 != 0 )
  sum += ( U[k+m] - u) * N_k(k+1, m-1, u, U ) / d2;

  return sum;
}

//non uniform rational B-splines, n control points, order m;
//  p is the output point
void nurb ( int n, int m, const Point3 control_points[],
                              float u, float U[],  Point3 &p )
{
```

```
    //sum control points,multiplied by respective basis functions
    Point3 p3;    //x, y, z components set to zero in constructor
    for ( int k = 0; k < n; ++k ){
      float Nk = N_k(k, m, u, U);  //blending (basis) function
      p3.x += Nk * control_points[k].x;
      p3.y += Nk * control_points[k].y;
      p3.z += Nk * control_points[k].z;
    }
    p = p3;              //= is a copy command in C++
  }
```

Figure 13-10 below shows a curve obtained using the **nurb** function shown above; 8 control points and order 4 are used in the calculations.

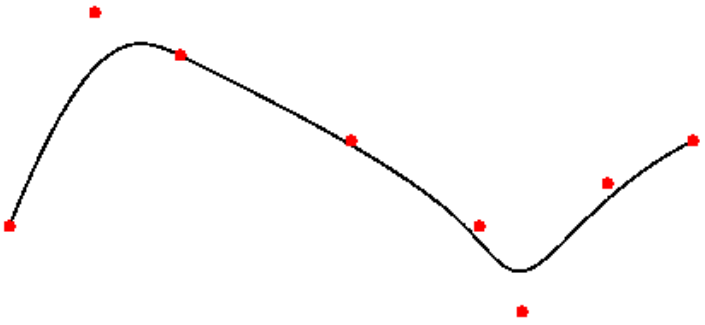

Figure 13-10 Curve Evaluated Using NURB with Order 4 and 8 Control Points

NURBS are just B-Splines with extensions made to accommodate points specified in homogeneous coordinates. They are invariant under the projective transformation. This means that if we draw two objects that are connected, they will be still connected when drawn in perspective. NURBS can also model conic sections precisely.

13.5 OpenGL Evaluators and NURBS

OpenGL provides functions to generate both Bezier (GL) and B-Spline (GLU) curves and surfaces. We can use the functions to interpolate vertices, normals, colors and textures.

13.5.1 OpenGL Evaluators for Bezier Curves and Surfaces

OpenGL supports the drawing of curves and surfaces through the use of evaluators. We use **glMap1***, which defines a one-dimensional evaluator to construct Bezier curves. This is done in the following steps, using **glMap1f** as example:

1. Specify the parameters with

```
void glMap1f(  GL_MAP1_VERTEX_3,
   float uMin,  float uMax,
   int stride,  int nPoints,
   const float *points  );
```

This assumes 3 coordinates per vertex, and *stride* is used if we have interleaved different kinds of data in the array *points*.

2. Activate Bezier curve display by

   glEnable(GL_MAP1_VERTEX_3);

3. Display the curve by putting

   glEvalCoord1f(float *uValue*);

   in a loop insdie **glBegin/glEnd**.

For uniform spacing of u values, we can also use

glMapGrid1f(int $nPartitions$, float $u1$, float $u2$);

to specify the number and range of u values, where $nPartitions$ is the number of partitions in the grid range interval $[u1, u2]$, $u1$ is a value used as the mapping for integer grid domain value $i = 0$, and $u2$ is a value used as the mapping for integer grid domain value $i = uPartitions$. The curve is actually drawn with

glEvalMesh1f(enum $mode$, int $n1$, int $n2$);

where $mode$ can be GL_LINE or GL_POINT and $n1$ and $n2$ specify the part of the curve that should be drawn.

Besides GL_MAP1_VERTEX_3, the commands **glMap1***() support other types of control points as listed in Table 13-1 below.

Table 13-1 Control Point Types for glMap1*()

Parameter	Data Types
GL_MAP1_VERTEX_3	vertex coordinates (x, y, z)
GL_MAP1_VERTEX_4	vertex coordinates (x, y, z, w)
GL_MAP1_INDEX	color index
GL_MAP1_COLOR_4	color components (R, G, B, A)
GL_MAP1_NORMAL	normal coordinates (x, y, z)
GL_MAP1_TEXTURE_COORD_1	texture coordinates s
GL_MAP1_TEXTURE_COORD_2	texture coordinates (s, t)
GL_MAP1_TEXTURE_COORD_3	texture coordinates (s, t, r)
GL_MAP1_TEXTURE_COORD_4	texture coordinates (s, t, r, q)

The following code segment shows an example of using the evaluator with 5 control points and the constructed Bezier curve is shown in Figure 13-11 below.

```
float controlPoints[Order][3] = {
    { -4.0, -1.0, 0.0}, { -2.0, 1.0, 0.0},
    { -3.0, -0.5, 0.0},
    {2.0, -1.0, 0.0}, {4.0, 1.0, 0.0}};

void init(void)
{
   glClearColor(1.0, 1.0, 1.0, 0.0);
   glShadeModel(GL_FLAT);
/*
 * GL_MAP1_VERTEX_3 -- specifies that 3-dimensional control points
 *        are provided and 3-D vertices should be produced
 * 0.0 -- low value of parmeter u
```

```
 * 1.0 -- high value of parmeter u
 * 3   -- number of floating-point values to advance in the data
 *        between two consecutive control points
 * 5   -- order of the spline (=degree+1) = number of control points
 */
   glMap1f(GL_MAP1_VERTEX_3,0.0,1.0, 3, Order, &controlPoints[0][0]);
   glEnable(GL_MAP1_VERTEX_3);
}

void display(void)
{
   int i;

   glClear( GL_COLOR_BUFFER_BIT );
   glColor3f( 0.0, 0.0, 0.0 );
   glBegin( GL_LINE_STRIP );
      for ( i = 0; i <= 30; i++ )
         glEvalCoord1f((float) i/30.0);
   glEnd();
   // The following code displays the control points as dots.
   glPointSize( 6.0 );
   glColor3f( 1.0, 0.0, 0.0 );
   glEnable ( GL_POINT_SMOOTH );
   glBegin(GL_POINTS);
     for ( i = 0; i < Order; i++ )
       glVertex3fv( &controlPoints[i][0] );
   glEnd();
   glFlush();
}
```

In the example, we use the command **glMap1f**() to define a one-dimensional evaluator that uses Equation (13.40) for Bezier curves to evaluate points. The output of the code is shown in Figure 13-11 below.

Figure 13-11 Curve Defined by glMap1f() in Above Code Segment

We can similarly construct two-dimensional Bezier surfaces or patches described by Equation (13.51); we summarize the procedures as follows, using **glMap2f**() as example:

1. Define the evaluator with

```
void glMap2f( GL_MAP2_VERTEX_3,
  float u1,    float u2,
  int ustride, int uorder,
  float v1,    float v2,
  int vstride, int vorder,
  const float *points );
```

2. Enable the evaluator by

glEnable(GL_MAP2_VERTEX_3);

3. Display the curve by putting

   glEvalCoord2f(float *uValue*, float *vValue*);

   in a loop insdie **glBegin/glEnd**. If we do not want to use loop, we can setup and evaluate a mesh with **glMapGrid2f**() and **glEvalMesh2f**().

The following code segment presents an example of using OpenGL commands to construct a Bezier surface. The output of it is shown in Figure 13-12 below.

```
const int uOrder = 4;
const int vOrder = 4;
float controlPoints[uOrder][vOrder][3] = {
   {{-2, -1, 4.0}, {-1, -1, 3.0},
    {0, -1, -1.5}, {2, -1, 2.5}},
   {{-2, -0.5, 1.0}, {-1, -0.5, 3.0},
    {0, -0.5, 0.0}, {2, -0.5, -1.0}},
   {{-2, 0.5, 4.0}, {-1, 0.5, 1.0},
    {0, 0.5, 2.0}, {2, 0.5, 4.0}},
   {{-2, 1.5, -2.0}, {-1, 1.5, -2.0},
    {0, 1.5, 0.0}, {2, 1.5, -1.0}}
};

void init(void)
{
   glClearColor(1.0, 1.0, 1.0, 0.0);
   glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, uOrder,
        0, 1, 12, vOrder, &controlPoints[0][0][0]);
   glEnable(GL_MAP2_VERTEX_3);
   glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
   glEnable(GL_DEPTH_TEST);
}

void display(void)
{
  int i, j;
  float u, v;

  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glColor3f( 0.0, 0.0, 0.0 );
  gluLookAt ( 10, 10, 10, 0, 0.0, 0.0, 0.0, 1.0, 0.0);
  glEnable ( GL_LINE_SMOOTH );
  glLineWidth( 2 );

  const int n1 = 8, n2 = 30;
  for ( i = 0; i <= n1; i++ ) {
    v = (float) i / n1;
    glBegin(GL_LINE_STRIP);
    for (j = 0; j <= n2; j++){
      u = (float) j / n2;
      glEvalCoord2f( u, v );
    }
    glEnd();
  }
  for (i = 0; i <= n1; i++) {
    u = (float) i / n1;
    glBegin(GL_LINE_STRIP);
    for (j = 0; j <= n2; j++) {
      v = (float) j / n2;
```

```
        glEvalCoord2f( u, v );
      }
      glEnd();
    }

    glFlush();
  }
```

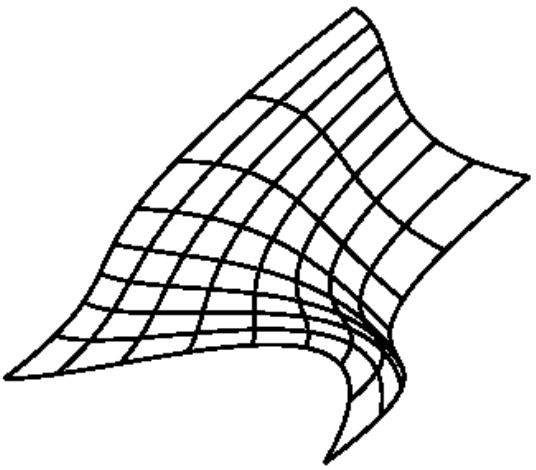

Figure 13-12 Bezier Surface Constructed using Evaluators

13.5.2 The GLU NURBS

Another way to draw Bezier curves and B-Splines is to use the NURBS interface. Internally, the NURBS interface is built on top of the evaluators discussed above. From the programmers point of view, the NURBS are simpler to use. It is also relatively easy to incorporate lighting effects and texture mapping in NURBS curves or surfaces. The following steps summarize the procedures to use NURBS:

1. We need to pass a NURBS context object into the NURBS interface each time we use the interface. Therefore, the first thing to do is to create a NURBS context object by

   GLUnurbs *nurbs = gluNewNurbsRenderer();

   This creates a pointer to a NURBS object, and we refer to this pointer when creating a NURBS curve or surface. A value of 0 is returned if there is not enough memory to allocate the object.

2. If we need to use lighting with a NURBS surface, we can enable the automatic normal-calculation feature by

   glEnable(GL_AUTO_NORMAL);

3. We may call **gluNurbsProperty**() to choose rendering values. The function has prototype,

   void gluNurbsProperty(GLUnurbs *_nurbs_, GLenum _property_, GLfloat _value_);

where

property specifies the property to be set, which can be GLU_SAMPLING_TOLERANCE, GLU_DISPLAY_MODE, GLU_CULLING, GLU_AUTO_LOAD_MATRIX, GLU_PARAMETRIC_TOLERANCE, GLU_SAMPLING_METHOD, GLU_U_STEP, GLU_V_STEP, or GLU_NURBS_MODE.

value specifies the value of the indicated property, which may be a numeric value or one of GLU_OUTLINE_POLYGON, GLU_FILL, GLU_OUTLINE_PATCH, GLU_TRUE, GLU_FALSE, GLU_PATH_LENGTH, GLU_PARAMETRIC_ERROR, GLU_DOMAIN_DISTANCE, GLU_NURBS_RENDERER, or GLU_NURBS_TESSELLATOR.

4. If we need notification when encountering an error, we can call

   void gluNurbsCallback (GLUnurbs **nurb*, GLenum *which* , _GLfuncptr *CallBackFunc*);

5. Start rendering our curve or surface by calling

   gluBeginCurve(*nurbs*) or **gluBeginSurface**(*nurbs*)

6. Generate and render our curve or surface by calling

   gluNurbsCurve() or **gluNurbsSurface**()

   at least once with the control points, knot sequence, and order of the blending functions for the NURBS object. We might further call these functions to specify surface normals and/or texture coordinates.

7. Finish rendering the curve or surface by calling

   gluEndCurve(*nurbs*) or **gluEndSurface**(*nurbs*).

The following code segment presents an example of constructing a NURBS surface with lighting. Same control points of the evaluator example above have been used. The output of it is shown in Figure 13-13 below.

```
GLUnurbsObj *nurbs;
void nurbsError( GLenum error )
{
  const GLubyte *s = gluErrorString ( error );
  printf ( "Nurbs Error: %s\n", s );  exit (0);
}

void init(void)
{ glClearColor(1.0, 1.0, 1.0, 0.0);
  GLfloat mat_ambient[] = { 0.5, 0.4, 0.3, 1.0 };
  GLfloat mat_diffuse[] = { 0.7, 0.8, 0.9, 1.0 };
  GLfloat mat_specular[] = { 0.9, 0.8, 0.7, 1.0 };
  GLfloat mat_shininess[] = { 50.0 };
  glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
  glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
  glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
  glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
```

```
  GLfloat light[] = { 1, 1, 1 };
  GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
  glLightfv(GL_LIGHT0, GL_DIFFUSE, light );
  glLightfv(GL_LIGHT0, GL_AMBIENT, light );
  glLightfv(GL_LIGHT0, GL_SPECULAR, light );
  glLightfv(GL_LIGHT0, GL_POSITION, light_position);
  glEnable(GL_LIGHTING);    glEnable(GL_LIGHT0);
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_AUTO_NORMAL);
  glEnable(GL_NORMALIZE);

  nurbs = gluNewNurbsRenderer();
  gluNurbsProperty( nurbs, GLU_SAMPLING_TOLERANCE, 25.0);
  gluNurbsProperty( nurbs, GLU_DISPLAY_MODE, GLU_FILL);
  gluNurbsCallback( nurbs, GLU_ERROR, (GLvoid (*)()) nurbsError);
}

void display(void)
{ glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  gluLookAt ( 10, 10, 10, 0, 0.0, 0.0, 0.0, 1.0, 0.0);
  const int uKnotCount = 8, vKnotCount = 8;
  float uKnots[uKnotCount] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
  float vKnots[vKnotCount] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
  int uStride = vOrder * 3, vStride = 3;
  gluBeginSurface ( nurbs );
    gluNurbsSurface ( nurbs,
                  uKnotCount, uKnots, vKnotCount, vKnots,
                  uStride, vStride, &controlPoints[0][0][0],
                  uOrder, vOrder, GL_MAP2_VERTEX_3);
  gluEndSurface ( nurbs );
  glFlush();
}
```

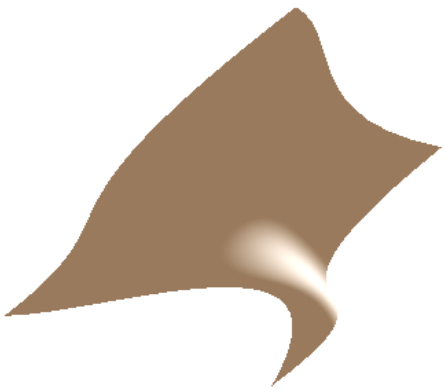

Figure 13-13 Surface Constructed by NURBS

13.6 Subdivision Surface

We discuss before that any surface can be approximated by a mesh of polygons, in particular triangles. Very often we can start from a coarse mesh and subdivide each polygon into smaller faces to obtain finer representation.

In general, we define the subdivision surfaces recursively. We start with a given polygonal mesh and apply a refinement scheme to subdivide the polygons of it, creating new

vertices and new faces. We compute the positions of the new vertices using affine combinations of nearby old vertices. The refinement process produces a denser mesh than the original one, containing more polygonal faces. We can apply the same refinement process to the resulting mesh repeatedly until we obtain a smooth surface we want.

We consider a simple example to illustrate this subdivision technique; we approximate a unit sphere centered at $O = (0, 0, 0)$ with a mesh of triangles. We first approximate the sphere using a mesh of 20 equivalent triangles defined by 12 vertices. Each vertice is on the sphere and its coordinates are defined by three values, X, Z, and 0 where

$$X = .525731112119133606, \quad Z = .850650808352039932$$

Note that the sum of the squares of X and Z is equal to 1. That is,

$$X^2 + Z^2 = 1 \tag{13.60}$$

Therefore, any vertice $P = (x, y, z)$ of any triangle of the mesh has a unit normal at it, which can be calculated by

$$\mathbf{n} = P - O = (x, y, z) - (0, 0, 0) = (x, y, z) \tag{13.61}$$

Obvisously, the magnitude of $\mathbf{n}$ is $|\mathbf{n}| = x^2 + y^2 + z^2 = X^2 + Z^2 + 0^2 = 1$.

We define the 12 vertices and the 20 faces of the mesh using two arrays:

```
//vertex data
static double vdata[12][3] = {
   {-X, 0.0, Z}, {X, 0.0, Z}, {-X, 0.0, -Z}, {X, 0.0, -Z},
   {0.0, Z, X}, {0.0, Z, -X}, {0.0, -Z, X}, {0.0, -Z, -X},
   {Z, X, 0.0}, {-Z, X, 0.0}, {Z, -X, 0.0}, {-Z, -X, 0.0}
};
//indices for triangles
static int tindices[20][3] = {
   {0,4,1}, {0,9,4}, {9,5,4}, {4,5,8}, {4,8,1},
   {8,10,1}, {8,3,10}, {5,3,8}, {5,2,3}, {2,7,3},
   {7,10,3}, {7,6,10}, {7,11,6}, {11,0,6}, {0,1,6},
   {6,1,10}, {9,0,11}, {9,11,2}, {9,2,5}, {7,2,11} };
```

For example, the first triangle is defined by the indices $\{0, 4, 1\}$ of the vertex array vdata[]; thus the coordinates of its vertices are given by vdata[0], vdata[4], and vdata[1], which have coordinate values

{-X, 0.0, Z}, {0.0, Z, X}, {X, 0.0, Z}

To refine the mesh, we subdivide a triangle into smaller triangles by making affine combinations of the original three vertices as shown in Figure 13-14 below. In the figure, the original triangle is defined by the three points P_1, P_2, and P_3. New triangles are formed by creating new vertices that are formed by affine combinations of the three vertices:

$$\begin{aligned} P_{12} &= \frac{1}{2}(P_1 + P_2) \\ P_{23} &= \frac{1}{2}(P_2 + P_3) \\ P_{31} &= \frac{1}{2}(P_3 + P_1) \end{aligned} \tag{13.62}$$

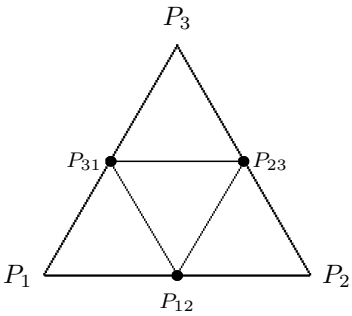

Figure 13-14 Subdividing a Triangle

Since a triangle always lies in a plane, the newly created triangles all lie in the same plane as the original one and they are beneath the the surface of the unit sphere. Obviously, these new triangles will not improve our approximation if we render them as is. To make the actual improvement, the trick is to move the new vertices P_{12}, P_{23}, and P_{31}, which are at a distance of less than 1 from the sphere center, to the surface of the sphere. This can be done by normalizing the normal to each point and moving the point to the tip of the unit normal. For example, we calculate the normal $\mathbf{N_{ij}}$ at P_{ij} by:

$$\mathbf{N_{ij}} = P_{ij} - O \tag{13.63}$$

We then normalize it by:

$$\mathbf{n_{ij}} = \frac{\mathbf{N_{ij}}}{|\mathbf{N_{ij}}|} \tag{13.64}$$

We recalculate the new position P'_{ij} of the vertex P_{ij} by:

$$P'_{ij} = \mathbf{n_{ij}} + O \tag{13.65}$$

Now P'_{ij} lies on the spherical surface as it is at a unit distance from the sphere center. The effect of the normalization process is to 'push out' the new vertices onto the spherical surface. (Keep in mind that the difference between two points is a vector and the sum of a vector and a point is a point. It is wrong and would be sloppy if we 'normalize' a point directly.)

As shown in Figure 13-14, we subdivide a triangle into four smaller ones and by 'pushing' the three new vertices onto the surface of the unit sphere, we have refined the mesh, which now has a total of $4 \times 20 = 80$ faces. The following code segment shows an implementation of this process and the output of it is shown in Figure 13-15 below.

```
Point3 vertices[12];

void midPoint(const Point3 &p1, const Point3 &p2, Point3  &p12)
{
  p12.x = ( p1.x + p2.x ) / 2.0;
  p12.y = ( p1.y + p2.y ) / 2.0;
  p12.z = ( p1.z + p2.z ) / 2.0;
}

void triangle(const Point3 &p1, const Point3 &p2, const Point3 &p3)
{
  glBegin(GL_TRIANGLES);
   glNormal3f( p1.x, p1.y, p1.z ); glVertex3f( p1.x, p1.y, p1.z );
   glNormal3f( p2.x, p2.y, p2.z ); glVertex3f( p2.x, p2.y, p2.z );
```

```
    glNormal3f( p3.x, p3.y, p3.z ); glVertex3f( p3.x, p3.y, p3.z );
   glEnd();
 }

 void subdivide(const Point3 &p1, const Point3 &p2, const Point3 &p3)
 {
   Point3 p12, p23, p31;

   midPoint ( p1, p2, p12 );
   midPoint ( p2, p3, p23 );
   midPoint ( p3, p1, p31 );

   Point3 O (0, 0, 0);          //center of sphere
   Vector3 v12, v23, v31;       //3D vectors
   v12 = p12 - O;
   v23 = p23 - O;
   v31 = p31 - O;
   //normalize the vectors
   v12.normalize();
   v23.normalize();
   v31.normalize();
   //find vertices at vector tips
   p12 = v12 + O;
   p23 = v23 + O;
   p31 = v31 + O;
   triangle ( p1, p12, p31 );
   triangle ( p2, p23, p12 );
   triangle ( p3, p31, p23);
   triangle ( p12, p23, p31);
 }

 void init(void)
 {
   glClearColor (1.0, 1.0, 1.0, 0.0);
   for ( int i = 0; i < 12; i++ )
     vertices[i] = Point3 ( vdata[i]  );
 }

 void display(void)
 {
   glClear (GL_COLOR_BUFFER_BIT);
   glLineWidth ( 2 );
   glColor3f (0.0, 0.0, 0.0);
   glLoadIdentity ();
   gluLookAt(2.2, 2.2, 2.2, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

   glPolygonMode( GL_FRONT_AND_BACK, GL_LINE );
  for (int i = 0; i < 20; i++) {
    int i0 = tindices[i][0];
    int i1 = tindices[i][1];
    int i2 = tindices[i][2];
    subdivide( vertices[i0], vertices[i1], vertices[i2] );
  }
  glFlush ();
}
```

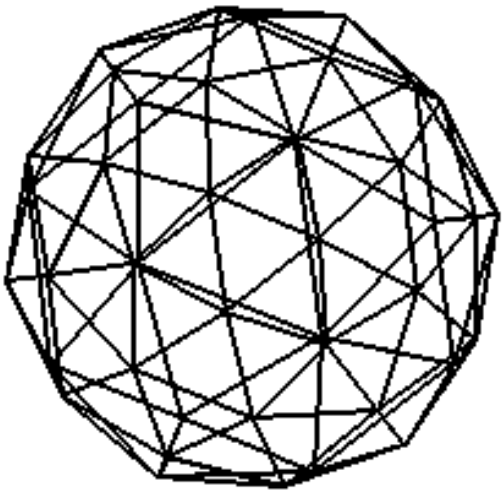

Figure 13-15 Sphere of 80 Triangles Constructed by Subdivision

We can continue to subdivide the triangles recursively until we are satisfied with the refinement of the mesh. To accomplish this recursive property, we just need to make slight modifications to the above **subdivide**() function:

```
void subdivide( const Point3 &p1, const Point3 &p2, const Point3 &p3,
                    int level)
{
  Point3 p12, p23, p31;

  midPoint ( p1, p2, p12 );
  midPoint ( p2, p3, p23 );
  midPoint ( p3, p1, p31 );

  Point3 O (0, 0, 0);          //sphere center
  Vector3 v12, v23, v31;       //3D vectors
  v12 = p12 - O;
  v23 = p23 - O;
  v31 = p31 - O;
  //normalize the vectors
  v12.normalize();
  v23.normalize();
  v31.normalize();
  //find vertices at vector tips
  p12 = v12 + O;
  p23 = v23 + O;
  p31 = v31 + O;
  if ( level > 0 ) {  //subdivide recursively
    --level;
    subdivide ( p1, p12, p31, level );
    subdivide ( p2, p23, p12, level );
    subdivide ( p3, p31, p23, level );
    subdivide ( p12, p23, p31, level );
  } else {
    triangle ( p1, p12, p31 );
    triangle ( p2, p23, p12 );
    triangle ( p3, p31, p23);
    triangle ( p12, p23, p31);
  }
```

```
}
```

Figure 13-16 shows a sphere constructed with one more level of subdivision, consisting of $4 \times 80 = 320$ triangles.

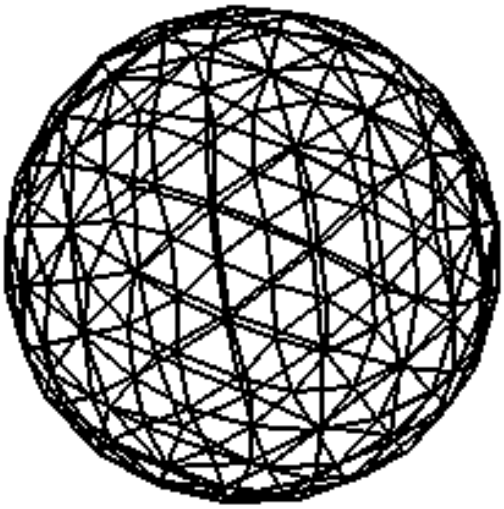

Figure 13-16 Sphere of 320 Triangles Constructed by Subdivision

Other books by the same author

Windows Fan, Linux Fan

by *Fore June*

Windws Fan, Linux Fan describes a true story about a spiritual battle between a Linux fan and a Windows fan. You can learn from the successful fan to become a successful Internet Service Provider (ISP) and create your own wealth. See *http://www.forejune.com/*

Second Edition, 2002.
ISBN: 0-595-26355-0 Price: $6.86

An Introduction to Digital Video Data Compression in Java

by *Fore June*

The book describes the the principles of digital video data compression techniques and its implementations in java. Topics covered include RBG-YCbCr conversion, macroblocks, DCT and IDCT, integer arithmetic, quantization, reorder, run-level encoding, entropy encoding, motion estimation, motion compensation and hybrid coding. See

http://www.forejune.com/

January 2011

ISBN-10: 1456570870

ISBN-13: 978-1456570873

An Introduction to Video Compression in C/C++

by *Fore June*

The book describes the the principles of digital video data compression techniques and its implementations in C/C++. Topics covered include RBG-YCbCr conversion, macroblocks, DCT and IDCT, integer arithmetic, quantization, reorder, run-level encoding, entropy encoding, motion estimation, motion compensation and hybrid coding.

January 2010
ISBN: 9781451522273