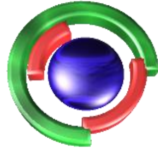
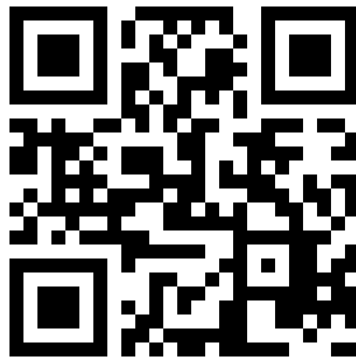
FUTURE VISION BIE

One Stop for All Study Materials & Lab Programs



By K B Hemanth Raj

Scan the QR Code to Visit the Web Page



Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

Table of Contents

1. Computer Graphics Hardware
Donald D. Hearn/M. Pauline Baker, Warren Carithers **1**

Computer Graphics Hardware Color Plates
Donald D. Hearn/M. Pauline Baker, Warren Carithers **27**

2. Computer Graphics Software
Donald D. Hearn/M. Pauline Baker, Warren Carithers **29**

3. Graphics Output Primitives
Donald D. Hearn/M. Pauline Baker, Warren Carithers **45**

4. Attributes of Graphics Primitives
Donald D. Hearn/M. Pauline Baker, Warren Carithers **99**

5. Implementation Algorithms for Graphics Primitives and Attributes
Donald D. Hearn/M. Pauline Baker, Warren Carithers **131**

6. Two-Dimensional Geometric Transformations
Donald D. Hearn/M. Pauline Baker, Warren Carithers **189**

7. Two-Dimensional Viewing
Donald D. Hearn/M. Pauline Baker, Warren Carithers **227**

8. Three-Dimensional Geometric Transformations
Donald D. Hearn/M. Pauline Baker, Warren Carithers **273**

9. Three-Dimensional Viewing
Donald D. Hearn/M. Pauline Baker, Warren Carithers **301**

Three-Dimensional Viewing Color Plate
Donald D. Hearn/M. Pauline Baker, Warren Carithers **353**

10. Hierarchical Modeling
Donald D. Hearn/M. Pauline Baker, Warren Carithers **355**

11. Computer Animation
Donald D. Hearn/M. Pauline Baker, Warren Carithers **365**

https://hemanthrajhemu.github.io

12. Three-Dimensional Object Representations
Donald D. Hearn/M. Pauline Baker, Warren Carithers **389**

Three-Dimensional Object Representations Color Plate
Donald D. Hearn/M. Pauline Baker, Warren Carithers **407**

13. Spline Representations
Donald D. Hearn/M. Pauline Baker, Warren Carithers **409**

14. Visible-Surface Detection Methods
Donald D. Hearn/M. Pauline Baker, Warren Carithers **465**

15. Illumination Models and Surface-Rendering Methods
Donald D. Hearn/M. Pauline Baker, Warren Carithers **493**

Illumination Models and Surface-Rendering Methods Color Plates
Donald D. Hearn/M. Pauline Baker, Warren Carithers **541**

16. Texturing and Surface-Detail Methods
Donald D. Hearn/M. Pauline Baker, Warren Carithers **543**

Texturing and Surface-Detail Methods Color Plates
Donald D. Hearn/M. Pauline Baker, Warren Carithers **567**

17. Color Models and Color Applications
Donald D. Hearn/M. Pauline Baker, Warren Carithers **569**

Color Models and Color Applications Color Plate
Donald D. Hearn/M. Pauline Baker, Warren Carithers **589**

18. Interactive Input Methods and Graphical User Interfaces
Donald D. Hearn/M. Pauline Baker, Warren Carithers **591**

Interactive Input Methods and Graphical User Interfaces Color Plates
Donald D. Hearn/M. Pauline Baker, Warren Carithers **631**

19. Global Illumination
Donald D. Hearn/M. Pauline Baker, Warren Carithers **633**

Global Illumination Color Plates
Donald D. Hearn/M. Pauline Baker, Warren Carithers **659**

20. Programmable Shaders
Donald D. Hearn/M. Pauline Baker, Warren Carithers **663**

Programmable Shaders Color Plates
Donald D. Hearn/M. Pauline Baker, Warren Carithers **693**

21. Algorithmic Modeling
Donald D. Hearn/M. Pauline Baker, Warren Carithers **695**

Algorithmic Modeling Color Plates
Donald D. Hearn/M. Pauline Baker, Warren Carithers **725**

https://hemanthrajhemu.github.io

Three-Dimensional Object Representations

1 Polyhedra
2 OpenGL Polyhedron Functions
3 Curved Surfaces
4 Quadric Surfaces
5 Superquadrics
6 OpenGL Quadric-Surface and Cubic-Surface Functions
7 Summary

Graphics scenes can contain many different kinds of objects and material surfaces: trees, flowers, clouds, rocks, water, bricks, wood paneling, rubber, paper, marble, steel, glass, plastic, and cloth, just to mention a few. So it may not be surprising that there is no single method that we can use to describe objects that will include all the characteristics of these different materials.

Polygon and quadric surfaces provide precise descriptions for simple Euclidean objects such as polyhedrons and ellipsoids. They are examples of **boundary representations** (**B-reps**), which describe a three-dimensional object as a set of surfaces that separate the object interior from the environment. In this chapter, we consider the features of these types of representation schemes and how they are used in computer-graphics applications.

From Chapter 13 of *Computer Graphics with OpenGL®*, Fourth Edition, Donald Hearn, M. Pauline Baker, Warren R. Carithers.
Copyright © 2011 by Pearson Education, Inc. Published by Pearson Prentice Hall. All rights reserved.

https://hemanthrajhemu.github.io

1 Polyhedra

The most commonly used boundary representation for a three-dimensional graphics object is a set of surface polygons that enclose the object interior. Many graphics systems store all object descriptions as sets of surface polygons. This simplifies and speeds up the surface rendering and display of objects because all surfaces are described with linear equations. For this reason, polygon descriptions are often referred to as *standard graphics objects*. In some cases, a polygonal representation is the only one available, but many packages also allow object surfaces to be described with other schemes, such as spline surfaces, which are usually converted to polygonal representations for processing through the viewing pipeline.

To describe an object as a set of polygon facets, we give the list of vertex coordinates for each polygon section over the object surface. The vertex coordinates and edge information for the surface sections are then stored in tables along with other information, such as the surface normal vector for each polygon. Some graphics packages provide routines for generating a polygon-surface mesh as a set of triangles or quadrilaterals. This allows us to describe a large section of an object's bounding surface, or even the entire surface, with a single command. And some packages also provide routines for displaying common shapes, such as a cube, sphere, or cylinder, represented with polygon surfaces. Sophisticated graphics systems use fast hardware-implemented polygon renderers that have the capability for displaying a million or more shaded polygons (usually triangles) per second, including the application of surface texture and special lighting effects.

2 OpenGL Polyhedron Functions

We have two methods for specifying polygon surfaces in an OpenGL program. Using the polygon primitives we can generate a variety of polyhedron shapes and surface meshes. In addition, we can use GLUT functions to display the five regular polyhedra.

OpenGL Polygon Fill-Area Functions

A set of polygon patches for a section of an object surface, or a complete description for a polyhedron, can be given using the OpenGL primitive constants `GL_POLYGON`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_QUADS`, and `GL_QUAD_STRIP`. For example, we could tessellate the lateral (axial) surface of a cylinder using a quadrilateral strip. Similarly, all faces of a parallelogram can be described with a set of rectangles, and all faces of a triangular pyramid could be specified using a set of connected triangular surfaces.

GLUT Regular Polyhedron Functions

Some standard shapes—the five regular polyhedra—are predefined by routines in the GLUT library. These polyhedra, also called the *Platonic solids*, are distinguished by the fact that all the faces of any regular polyhedron are identical regular polygons. Thus, all edges in a regular polyhedron are equal, all edge angles are equal, and all angles between faces are equal. Polyhedra are named according to the number of faces in each of the solids, and the five regular polyhedra are the regular tetrahedron (or triangular pyramid, with 4 faces), the regular hexahedron (or cube, with 6 faces), the regular octahedron (8 faces), the regular dodecahedron (12 faces), and the regular icosahedron (20 faces).

https://hemanthrajhemu.github.io

Ten functions are provided in GLUT for generating these solids: five of the functions produce wire-frame objects, and five display the polyhedra facets as shaded fill areas. The displayed surface characteristics for the fill areas are determined by the material properties and the lighting conditions that we set for a scene. Each regular polyhedron is described in modeling coordinates, so that each is centered at the world-coordinate origin.

We obtain the four-sided, regular triangular pyramid using either of these two functions:

```
glutWireTetrahedron ( );
```

or

```
glutSolidTetrahedron ( );
```

This polyhedron is generated with its center at the world-coordinate origin and with a radius (distance from the center of the tetrahedron to any vertex) equal to $\sqrt{3}$.

The six-sided regular hexahedron (cube) is displayed with

```
glutWireCube (edgeLength);
```

or

```
glutSolidCube (edgeLength);
```

Parameter `edgeLength` can be assigned any positive, double-precision floating-point value, and the cube is centered on the coordinate origin.

To display the eight-sided regular octahedron, we invoke either of the following commands:

```
glutWireOctahedron ( );
```

or

```
glutSolidOctahedron ( );
```

This polyhedron has equilateral triangular faces, and the radius (distance from the center of the octahedron at the coordinate origin to any vertex) is 1.0.

The twelve-sided regular dodecahedron, centered at the world-coordinate origin, is generated with

```
glutWireDodecahedron ( );
```

or

```
glutSolidDodecahedron ( );
```

Each face of this polyhedron is a pentagon.

The following two functions generate the twenty-sided regular icosahedron:

```
glutWireIcosahedron ( );
```

or

```
glutSolidIcosahedron ( );
```

Default radius (distance from the polyhedron center at the coordinate origin to any vertex) for the icosahedron is 1.0, and each face is an equilateral triangle.

https://hemanthrajhemu.github.io

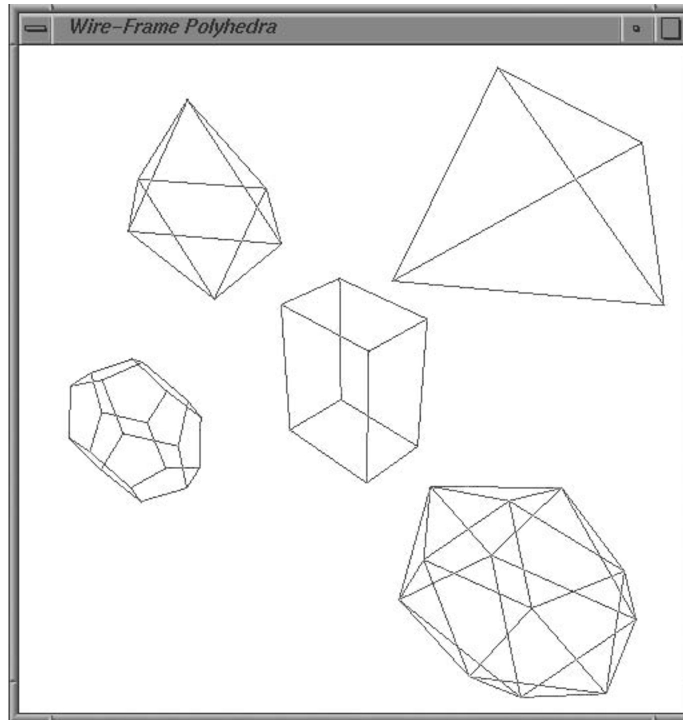


FIGURE 1
A perspective view of the five GLUT polyhedra, scaled and positioned within a display window by procedure `displayWirePolyhedra`.

Example GLUT Polyhedron Program

Using the GLUT functions for the Platonic solids, the following program generates a transformed, wire-frame perspective display of these polyhedrons. All five solids are positioned within one display window (shown in Figure 1).

```
#include <GL/glut.h>

GLsizei winWidth = 500, winHeight = 500;  // Initial display-window size.

void init (void)
{
    glClearColor (1.0, 1.0, 1.0, 0.0);  // White display window.
}

void displayWirePolyhedra (void)
{
    glClear (GL_COLOR_BUFFER_BIT);  // Clear display window.

    glColor3f (0.0, 0.0, 1.0);      // Set line color to blue.

    /*  Set viewing transformation.  */
    gluLookAt (5.0, 5.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

    /*  Scale cube and display as wire-frame parallelepiped. */
    glScalef (1.5, 2.0, 1.0);
    glutWireCube (1.0);
```


https://hemanthrajhemu.github.io

```
    /*  Scale, translate, and display wire-frame dodecahedron. */
    glScalef (0.8, 0.5, 0.8);
    glTranslatef (-6.0, -5.0, 0.0);
    glutWireDodecahedron ( );

    /*  Translate and display wire-frame tetrahedron. */
    glTranslatef (8.6, 8.6, 2.0);
    glutWireTetrahedron ( );

    /*  Translate and display wire-frame octahedron. */
    glTranslatef (-3.0, -1.0, 0.0);
    glutWireOctahedron ( );

    /*  Scale, translate, and display wire-frame icosahedron. */
    glScalef (0.8, 0.8, 1.0);
    glTranslatef (4.3, -2.0, 0.5);
    glutWireIcosahedron ( );

    glFlush ( );
}

void winReshapeFcn (GLint newWidth, GLint newHeight)
{
    glViewport (0, 0, newWidth, newHeight);

    glMatrixMode (GL_PROJECTION);
    glFrustum (-1.0, 1.0, -1.0, 1.0, 2.0, 20.0);

    glMatrixMode (GL_MODELVIEW);

    glClear (GL_COLOR_BUFFER_BIT);
}

void main (int argc, char** argv)
{
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition (100, 100);
    glutInitWindowSize (winWidth, winHeight);
    glutCreateWindow ("Wire-Frame Polyhedra");

    init ( );
    glutDisplayFunc (displayWirePolyhedra);
    glutReshapeFunc (winReshapeFcn);

    glutMainLoop ( );
}
```

3 Curved Surfaces

Equations for objects with curved boundaries can be expressed in either a parametric or a nonparametric form. The various objects that are often useful in graphics

https://hemanthrajhemu.github.io

applications include quadric surfaces, superquadrics, polynomial and exponential functions, and spline surfaces. These input object descriptions typically are tessellated to produce polygon-mesh approximations for the surfaces.

4 Quadric Surfaces

A frequently used class of objects are the *quadric surfaces*, which are described with second-degree equations (quadratics). They include spheres, ellipsoids, tori, paraboloids, and hyperboloids. Quadric surfaces, particularly spheres and ellipsoids, are common elements of graphics scenes, and routines for generating these surfaces are often available in graphics packages. Also, quadric surfaces can be produced with rational spline representations.

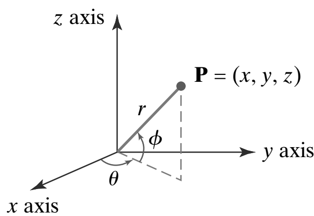


FIGURE 2
Parametric coordinate position (r, θ, ϕ) on the surface of a sphere with radius r.

Sphere

In Cartesian coordinates, a spherical surface with radius r centered on the coordinate origin is defined as the set of points (x, y, z) that satisfy the equation

$$x^2 + y^2 + z^2 = r^2 \tag{1}$$

We can also describe the spherical surface in parametric form, using latitude and longitude angles (Figure 2):

$$\begin{aligned} x &= r\cos\phi\cos\theta, & -\pi/2 \le \phi \le \pi/2 \\ y &= r\cos\phi\sin\theta, & -\pi \le \theta \le \pi \\ z &= r\sin\phi \end{aligned} \tag{2}$$

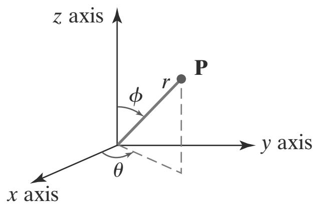


FIGURE 3
Spherical coordinate parameters (r, θ, ϕ), using colatitude for angle ϕ.

The parametric representation in Equations 2 provides a symmetric range for the angular parameters θ and ϕ. Alternatively, we could write the parametric equations using standard spherical coordinates, where angle ϕ is specified as the colatitude (Figure 3). Then, ϕ is defined over the range $0 \le \phi \le \pi$, and θ is often taken in the range $0 \le \theta \le 2\pi$. We could also set up the representation using parameters u and v defined over the range from 0 to 1 by substituting $\phi = \pi u$ and $\theta = 2\pi v$.

Ellipsoid

An ellipsoidal surface can be described as an extension of a spherical surface where the radii in three mutually perpendicular directions can have different values (Figure 4). The Cartesian representation for points over the surface of an ellipsoid centered on the origin is

$$\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1 \tag{3}$$

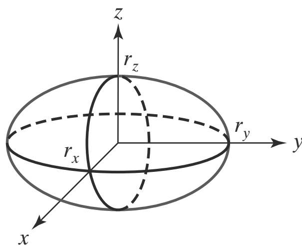


FIGURE 4
An ellipsoid with radii r_x, r_y, and r_z, centered on the coordinate origin.

And a parametric representation for the ellipsoid in terms of the latitude angle ϕ and the longitude angle θ in Figure 2 is

$$\begin{aligned} x &= r_x\cos\phi\cos\theta, & -\pi/2 \le \phi \le \pi/2 \\ y &= r_y\cos\phi\sin\theta, & -\pi \le \theta \le \pi \\ z &= r_z\sin\phi \end{aligned} \tag{4}$$

https://hemanthrajhemu.github.io

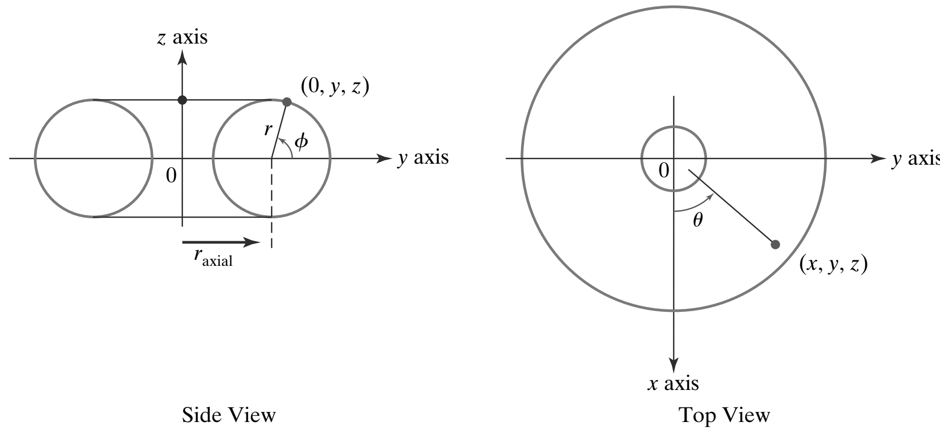


FIGURE 5
A torus, centered on the coordinate origin, with a circular cross-section and with the torus axis along the z axis.

Torus

A doughnut-shaped object is called a *torus* or *anchor ring*. Most often it is described as the surface generated by rotating a circle or an ellipse about a coplanar axis line that is external to the conic. The defining parameters for a torus are then the distance of the conic center from the rotation axis and the dimensions of the conic. A torus generated by the rotation of a circle with radius r in the yz plane about the z axis is shown in Figure 5. With the circle center on the y axis, the axial radius, r_{axial}, of the resulting torus is equal to the distance along the y axis to the circle center from the z axis (the rotation axis); and the cross-sectional radius of the torus is the radius of the generating circle.

The equation for the cross-sectional circle shown in the side view of Figure 5 is

$$(y - r_{\text{axial}})^2 + z^2 = r^2$$

Rotating this circle about the z axis produces the torus whose surface positions are described with the Cartesian equation

$$\left(\sqrt{x^2 + y^2} - r_{\text{axial}}\right)^2 + z^2 = r^2 \tag{5}$$

The corresponding parametric equations for the torus with a circular cross-section are

$$\begin{aligned} x &= (r_{\text{axial}} + r\cos\phi)\cos\theta, \qquad -\pi \le \phi \le \pi \\ y &= (r_{\text{axial}} + r\cos\phi)\sin\theta, \qquad -\pi \le \theta \le \pi \\ z &= r\sin\phi \end{aligned} \tag{6}$$

We could also generate a torus by rotating an ellipse, instead of a circle, about the z axis. For an ellipse in the yz plane with semimajor and semiminor axes denoted as r_y and r_z, we can write the ellipse equation as

$$\left(\frac{y - r_{\text{axial}}}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1$$

where r_{axial} is the distance along the y axis from the rotation z axis to the ellipse center. This generates a torus that can be described with the Cartesian equation

$$\left(\frac{\sqrt{x^2 + y^2} - r_{\text{axial}}}{r_y}\right) + \left(\frac{z}{r_z}\right)^2 = 1 \tag{7}$$

https://hemanthrajhemu.github.io

The corresponding parametric representation for the torus with an elliptical cross-section is

$$\begin{aligned} x &= (r_{\text{axial}} + r_y \cos\phi)\cos\theta, & -\pi \le \phi \le \pi \\ y &= (r_{\text{axial}} + r_y \cos\phi)\sin\theta, & -\pi \le \theta \le \pi \\ z &= r_z \sin\phi \end{aligned} \tag{8}$$

Other variations on the preceding torus equations are possible. For example, we could generate a torus surface by rotating either a circle or an ellipse along an elliptical path around the rotation axis.

5 Superquadrics

The class of objects called **Superquadrics** is a generalization of the quadric representations. Superquadrics are formed by incorporating additional parameters into the quadric equations to provide increased flexibility for adjusting object shapes. One additional parameter is added to curve equations, and two additional parameters are used in surface equations.

Superellipse

We obtain a Cartesian representation for a superellipse from the corresponding equation for an ellipse by allowing the exponent on the x and y terms to be variable. One way to do this is to write the Cartesian superellipse equation in the form

$$\left(\frac{x}{r_x}\right)^{2/s} + \left(\frac{y}{r_y}\right)^{2/s} = 1 \tag{9}$$

where parameter s can be assigned any real value. When $s = 1$, we have an ordinary ellipse.

Corresponding parametric equations for the superellipse of Equation 9 can be expressed as

$$\begin{aligned} x &= r_x \cos^s\theta, & -\pi \le \theta \le \pi \\ y &= r_y \sin^s\theta \end{aligned} \tag{10}$$

Figure 6 illustrates superellipse shapes that can be generated using various values for parameter s.

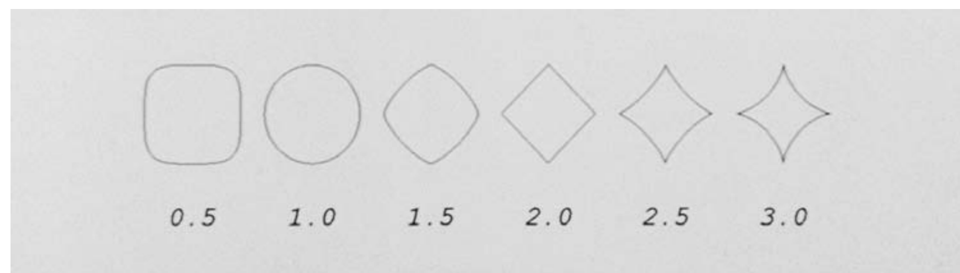


FIGURE 6
Superellipses plotted with values for parameter s ranging from 0.5 to 3.0 and with $r_x = r_y$.

https://hemanthrajhemu.github.io

Superellipsoid

A Cartesian representation for a superellipsoid is obtained from the equation for an ellipsoid by incorporating two exponent parameters as follows:

$$\left[\left(\frac{x}{r_x}\right)^{2/s_2} + \left(\frac{y}{r_y}\right)^{2/s_2}\right]^{s_2/s_1} + \left(\frac{z}{r_z}\right)^{2/s_1} = 1 \tag{11}$$

For $s_1 = s_2 = 1$, we have an ordinary ellipsoid.

We can then write the corresponding parametric representation for the superellipsoid of Equation 11 as

$$\begin{aligned} x &= r_x \cos^{s_1}\phi \cos^{s_2}\theta, && -\pi/2 \le \phi \le \pi/2 \\ y &= r_y \cos^{s_1}\phi \sin^{s_2}\theta, && -\pi \le \theta \le \pi \\ z &= r_z \sin^{s_1}\phi \end{aligned} \tag{12}$$

Color Plate 10 illustrates superellipsoid shapes that can be generated using various values for parameters s_1 and s_2. These and other superquadric shapes can be combined to create more complex structures, such as depictions of furniture, threaded bolts, and other hardware.

6 OpenGL Quadric-Surface and Cubic-Surface Functions

A sphere and a number of other three-dimensional quadric-surface objects can be displayed using functions that are included in the OpenGL Utility Toolkit (GLUT) and in the OpenGL Utility (GLU). In addition, GLUT has one function for displaying a teapot shape that is defined with bicubic surface patches. The GLUT functions, which are easy to incorporate into an application program, have two versions each. One version of each function displays a wire-frame surface, and the other displays the surface as a rendered set of fill-area polygon patches. With the GLUT functions, we can display a sphere, cone, torus, or the teapot. Quadric-surface GLU functions are a little more involved to set up, but they provide a few more options. With the GLU functions, we can display a sphere, cylinder, tapered cylinder, cone, flat circular ring (or hollow disk), and a section of a circular ring (or disk).

GLUT Quadric-Surface Functions

We generate a GLUT sphere with either of these two functions:

```
glutWireSphere (r, nLongitudes, nLatitudes);
```

or

```
glutSolidSphere (r, nLongitudes, nLatitudes);
```

where the sphere radius is determined by the double-precision floating-point number assigned to parameter `r`. Parameters `nLongitudes` and `nLatitudes` are used to select the integer number of longitude and latitude lines that will be used to approximate the spherical surface as a quadrilateral mesh. Edges of the quadrilateral surface patches are straight-line approximations of the longitude and latitude lines. The sphere is defined in modeling coordinates, centered at the world-coordinate origin with its polar axis along the z axis.

https://hemanthrajhemu.github.io

A GLUT cone is obtained with

```
glutWireCone (rBase, height, nLongitudes, nLatitudes);
```

or

```
glutSolidCone (rBase, height, nLongitudes, nLatitudes);
```

We set double-precision, floating-point values for the radius of the cone base and for the cone height using parameters `rbase` and `height`, respectively. As with a GLUT sphere, parameters `nLongitudes` and `nLatitudes` are assigned integer values that specify the number of orthogonal surface lines for the quadrilateral mesh approximation. A cone longitude line is a straight-line segment along the cone surface from the apex to the base that lies in a plane containing the cone axis. Each latitude line is displayed as a set of straight-line segments around the circumference of a circle on the cone surface that is parallel to the cone base and that lies in a plane perpendicular to the cone axis. The cone is described in modeling coordinates, with the center of the base at the world-coordinate origin and with the cone axis along the world z axis.

Wire-frame or surface-shaded displays of a torus with a circular cross-section are produced with

```
glutWireTorus (rCrossSection, rAxial, nConcentrics,
                 nRadialSlices);
```

or

```
glutSolidTorus (rCrossSection, rAxial, nConcentrics,
                  nRadialSlices);
```

The torus obtained with these GLUT routines can be described as the surface generated by rotating a circle with radius `rCrossSection` about the coplanar z axis, where the distance of the circle center from the z axis is `rAxial` (see Section 4). We select a size for the torus using double-precision, floating-point values for these radii in the GLUT functions. And the size of the quadrilaterals in the approximating surface mesh for the torus is set with integer values for parameters `nConcentrics` and `nRadialSlices`. Parameter `nConcentrics` specifies the number of concentric circles (with center on the z axis) to be used on the torus surface, and parameter `nRadialSlices` specifies the number of radial slices through the torus surface. These two parameters designate the number of orthogonal grid lines over the torus surface, with the grid lines displayed as straight-line segments (the boundaries of the quadrilaterals) between intersection positions. The displayed torus is centered on the world-coordinate origin, with its axis along the world z axis.

GLUT Cubic-Surface Teapot Function

During the early development of computer-graphics methods, sets of polygon-mesh data tables were constructed for the description of several three-dimensional objects that could be used to test rendering techniques. These objects included the surfaces of a Volkswagen automobile and a teapot, developed at the University of Utah. The data set for the Utah teapot, as constructed by Martin Newell in 1975, contains 306 vertices, defining 32 bicubic Bézier surface patches. Since determining the surface coordinates for a complex object is time-consuming,

https://hemanthrajhemu.github.io

these data sets, particularly the teapot surface mesh, became widely used.

We can display the teapot, as a mesh of over 1,000 bicubic surface patches, using either of the following two GLUT functions:

```
glutWireTeapot (size);
```

or

```
glutSolidTeapot (size);
```

The teapot surface is generated using OpenGL Bézier curve functions. Parameter `size` sets the double-precision floating-point value for the maximum radius of the teapot bowl. The teapot is centered on the world-coordinate origin coordinate origin with its vertical axis along the y axis.

GLU Quadric-Surface Functions

To generate a quadric surface using GLU functions, we need to assign a name to the quadric, activate the GLU quadric renderer, and designate values for the surface parameters. In addition, we can set other parameter values to control the appearance of a GLU quadric surface.

The following statements illustrate the basic sequence of calls for displaying a wire-frame sphere centered on the world-coordinate origin:

```
GLUquadricObj *sphere1;

sphere1 = gluNewQuadric ( );
gluQuadricDrawStyle (sphere1, GLU_LINE);

gluSphere (sphere1, r, nLongitudes, nLatitudes);
```

A name for the quadric object is defined in the first statement, and, for this example, we have chosen the name `sphere1`. This name is then used in other GLU functions to reference this particular quadric surface. Next, the quadric renderer is activated with the `gluNewQuadric` function, and then the display mode `GLU_LINE` is selected for `sphere1` with the `gluQuadricDrawStyle` command. Thus, the sphere is displayed in a wire-frame form with a straight-line segment between each pair of surface vertices. Parameter `r` is assigned a double-precision value for the sphere radius, and the sphere surface is divided into a set of polygon facets by the equally spaced longitude and latitude lines. We specify the integer number of longitude lines and latitude lines as values for parameters `nLongitudes` and `nLatitudes`.

Three other display modes are available for GLU quadric surfaces. Using the symbolic constant `GLU_POINT` in the `gluQuadricDrawStyle`, we display a quadric surface as a point plot. For the sphere, a point is displayed at each surface vertex formed by the intersection of a longitude line and a latitude line. Another option is the symbolic constant `GLU_SILHOUETTE`. This produces a wire-frame display without the shared edges between two coplanar polygon facets. And with the symbolic constant `GLU_FILL`, we display the polygon patches as shaded fill areas.

https://hemanthrajhemu.github.io

We generate displays of the other GLU quadric-surface primitives using the same basic sequence of commands. To produce a view of a cone, cylinder, or tapered cylinder, we replace the `gluSphere` function with

```
gluCylinder (quadricName, rBase, rTop, height, nLongitudes,
                 nLatitudes);
```

The base of this object is in the xy plane ($z = 0$), and the axis is the z axis. We assign a double-precision radius value to the base of this quadric surface using parameter `rBase`, and we assign a radius to the top of the quadric surface using parameter `rTop`. If `rTop` = 0.0, we get a cone; if `rTop` = `rBase`, we obtain a cylinder. Otherwise, a tapered cylinder is displayed. A double-precision height value is assigned to parameter `height`, and the surface is divided into a number of equally spaced vertical and horizontal lines as determined by the integer values assigned to parameters `nLongitudes` and `nLatitudes`.

A flat, circular ring or solid disk is displayed in the xy plane ($z = 0$) and centered on the world-coordinate origin with

```
gluDisk (ringName, rInner, rOuter, nRadii, nRings);
```

We set double-precision values for an inner radius and an outer radius with parameters `rInner` and `rOuter`. If `rInner` = 0, the disk is solid. Otherwise, it is displayed with a concentric hole in the center of the disk. The disk surface is divided into a set of facets with integer parameters `nRadii` and `nRings`, which specify the number of radial slices to be used in the tessellation and the number of concentric circular rings, respectively. Orientation for the ring is defined with respect to the z axis, with the front of the ring facing in the $+z$ direction and the back of the ring facing in the $-z$ direction.

We can specify a section of a circular ring with the following GLU function:

```
gluPartialDisk (ringName, rInner, rOuter, nRadii, nRings,
                     startAngle, sweepAngle);
```

The double-precision parameter `startAngle` designates an angular position in degrees in the xy plane measured clockwise from the positive y axis. Similarly, parameter `sweepAngle` denotes an angular distance in degrees from the `startAngle` position. Thus, a section of a flat, circular disk is displayed from angular position `startAngle` to `startAngle` + `sweepAngle`. For example, if `startAngle` = 0.0 and `sweepAngle` = 90.0, then the section of the disk lying in the first quadrant of the xy plane is displayed.

Allocated memory for any GLU quadric surface can be reclaimed and the surface eliminated with

```
gluDeleteQuadric (quadricName);
```

Also, we can define the front and back directions for any quadric surface with the following orientation function:

```
gluQuadricOrientation (quadricName, normalVectorDirection);
```

Parameter `normalVectorDirection` is assigned either `GLU_OUTSIDE` or `GLU_INSIDE` to indicate a direction for the surface normal vectors, where "outside" indicates the **front-face direction** and "inside" indicates the

https://hemanthrajhemu.github.io

back-face direction. The default value is GLU_OUTSIDE. For the flat, circular ring, the default front-face direction is in the direction of the positive z axis ("above" the disk). Another option is the generation of surface-normal vectors, as follows:

```
gluQuadricNormals (quadricName, generationMode);
```

A symbolic constant is assigned to parameter generationMode to indicate how surface-normal vectors should be generated. The default is GLU_NONE, which means that no surface normals are to be generated and no lighting conditions typically are applied to the quadric surface. For flat surface shading (a constant color value for each surface), we use the symbolic constant GLU_FLAT. This produces one surface normal for each polygon facet. When other lighting and shading conditions are to be applied, we use the constant GLU_SMOOTH, which generates a normal vector for each surface vertex position.

Other options for GLU quadric surfaces include setting surface-texture parameters. In addition, we can designate a function that is to be invoked if an error occurs during the generation of a quadric surface:

```
gluQuadricCallback (quadricName, GLU_ERROR, function);
```

Example Program Using GLUT and GLU Quadric-Surface Functions

Three quadric-surface objects (sphere, cone, and cylinder) are displayed in a wire-frame representation by the following example program. We set the view-up direction as the positive z axis so that the axis for all displayed objects is vertical. The three objects are positioned at different locations within a single display window, as shown in Figure 7.

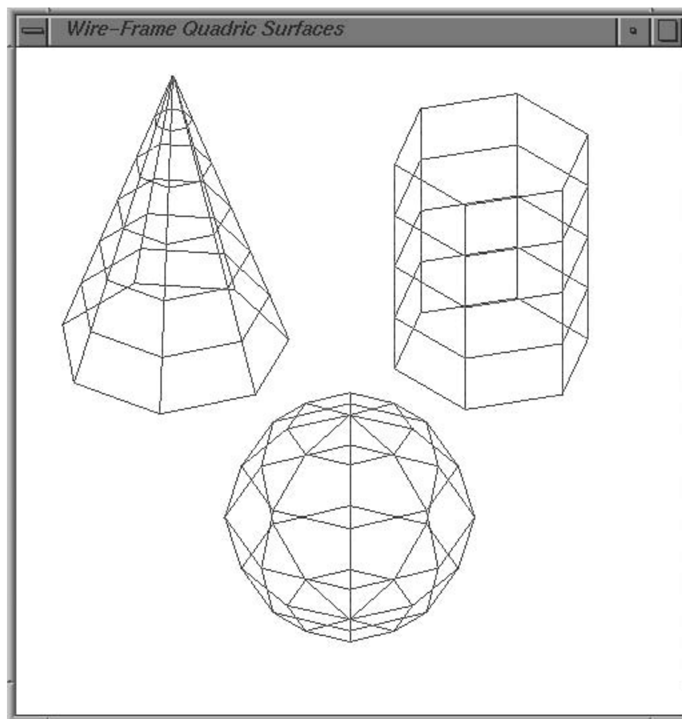


FIGURE 7
Display of a GLUT sphere, GLUT cone, and GLU cylinder, positioned within a display window by procedure wireQuadSurfs.

https://hemanthrajhemu.github.io

```
#include <GL/glut.h>

GLsizei winWidth = 500, winHeight = 500;    // Initial display-window size.

void init (void)
{
   glClearColor (1.0, 1.0, 1.0, 0.0);       // Set display-window color.
}

void wireQuadSurfs (void)
{
   glClear (GL_COLOR_BUFFER_BIT);           // Clear display window.

   glColor3f (0.0, 0.0, 1.0);               // Set line-color to blue.


   /*  Set viewing parameters with world z axis as view-up direction.  */
   gluLookAt (2.0, 2.0, 2.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);

   /*  Position and display GLUT wire-frame sphere.  */
   glPushMatrix ( );
   glTranslatef (1.0, 1.0, 0.0);
   glutWireSphere (0.75, 8, 6);
   glPopMatrix ( );

   /*  Position and display GLUT wire-frame cone.  */
   glPushMatrix ( );
   glTranslatef (1.0, -0.5, 0.5);
   glutWireCone (0.7, 2.0, 7, 6);
   glPopMatrix ( );

   /*  Position and display GLU wire-frame cylinder.  */
   GLUquadricObj *cylinder;   // Set name for GLU quadric object.
   glPushMatrix ( );
   glTranslatef (0.0, 1.2, 0.8);
   cylinder = gluNewQuadric ( );
   gluQuadricDrawStyle (cylinder, GLU_LINE);
   gluCylinder (cylinder, 0.6, 0.6, 1.5, 6, 4);
   glPopMatrix ( );

   glFlush ( );
}

void winReshapeFcn (GLint newWidth, GLint newHeight)
{
   glViewport (0, 0, newWidth, newHeight);

   glMatrixMode (GL_PROJECTION);
   glOrtho (-2.0, 2.0, -2.0, 2.0, 0.0, 5.0);

   glMatrixMode (GL_MODELVIEW);

   glClear (GL_COLOR_BUFFER_BIT);
}

void main (int argc, char** argv)
{
   glutInit (&argc, argv);
```


https://hemanthrajhemu.github.io

```
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition (100, 100);
    glutInitWindowSize (winWidth, winHeight);
    glutCreateWindow ("Wire-Frame Quadric Surfaces");

    init ( );
    glutDisplayFunc (wireQuadSurfs);
    glutReshapeFunc (winReshapeFcn);

    glutMainLoop ( );
}
```

7 Summary

Many representations have been developed for modeling the wide variety of objects and materials that we might want to display in a computer-graphics scene. In most cases, a three-dimensional object representation is rendered by a software package as a *standard graphics object*, whose surfaces are displayed as a polygon mesh.

Functions for displaying some common quadric surfaces, such as spheres and ellipsoids, are often available in graphics packages. Extensions of the quadrics, called *superquadrics*, provide additional parameters for creating a wider variety of object shapes.

Polygon surface facets for a standard graphics object can be specified in OpenGL using the polygon, triangle, or quadrilateral primitive functions. Also, GLUT routines are available for displaying the five regular polyhedra. Spheres, cones, and other quadric-surface objects can be displayed with GLUT and GLU functions, and a GLUT routine is provided for the generation of the cubic-surface Utah teapot. Tables 1 and 2 summarize the OpenGL polyhedron and quadric functions discussed in this chapter.

TABLE 1

Summary of OpenGL Polyhedron Functions

Function	Description
glutWireTetrahedron	Displays a wire-frame tetrahedron.
glutSolidTetrahedron	Displays a surface-shaded tetrahedron.
glutWireCube	Displays a wire-frame cube.
glutSolidCube	Displays a surface-shaded cube.
glutWireOctahedron	Displays a wire-frame octahedron.
glutSolidOctahedron	Displays a surface-shaded octahedron.
glutWireDodecahedron	Displays a wire-frame dodecahedron.
glutSolidDodecahedron	Displays a surface-shaded dodecahedron.
glutWireIcosahedron	Displays a wire-frame icosahedron.
glutSolidIcosahedron	Displays a surface-shaded icosahedron.

https://hemanthrajhemu.github.io

TABLE 2

Summary of OpenGL Quadric-Surface and Cubic-Surface Functions

Function	Description
`glutWireSphere`	Displays a wire-frame GLUT sphere.
`glutSolidSphere`	Displays a surface-shaded GLUT sphere.
`glutWireCone`	Displays a wire-frame GLUT cone.
`glutSolidCone`	Displays a surface-shaded GLUT cone.
`glutWireTorus`	Displays a wire-frame GLUT torus with a circular cross-section.
`glutSolidTorus`	Displays a surface-shaded, circular cross-section GLUT torus.
`glutWireTeapot`	Displays a wire-frame GLUT teapot.
`glutSolidTeapot`	Displays a surface-shaded GLUT teapot.
`gluNewQuadric`	Activates the GLU quadric renderer for an object name that has been defined with the declaration: `GLUquadricObj *nameOfObject;`
`gluQuadricDrawStyle`	Selects a display mode for a predefined GLU object name.
`gluSphere`	Displays a GLU sphere.
`gluCylinder`	Displays a GLU cone, cylinder, or tapered cylinder.
`gluDisk`	Displays a GLU flat, circular ring or solid disk.
`gluPartialDisk`	Displays a section of a GLU flat, circular ring or solid disk.
`gluDeleteQuadric`	Eliminates a GLU quadric object.
`gluQuadricOrientation`	Defines inside and outside orientations for a GLU quadric object.
`gluQuadricNormals`	Specifies how surface-normal vectors should be generated for a GLU quadric object.
`gluQuadricCallback`	Specifies a callback error function for a GLU quadric object.

REFERENCES

A detailed discussion of superquadrics is contained in Barr (1981). Programming techniques for various representations can be found in Glassner (1990), Arvo (1991), Kirk (1992), Heckbert (1994), and Paeth (1995). Kilgard (1996) discusses the GLUT functions for displaying polyhedrons, quadric surfaces, and the Utah teapot. And a complete listing of the OpenGL functions in the core library and in GLU is presented in Shreiner (2000).

EXERCISES

1 Set up an algorithm for converting a given sphere to a polygon-mesh representation.

2 Set up an algorithm for converting a given ellipsoid to a polygon-mesh representation.

3 Set up an algorithm for converting a given cylinder to a polygon-mesh representation.

4 Set up an algorithm for converting a given superellipsoid to a polygon-mesh representation.

5 Set up an algorithm for converting a given torus with a circular cross section to a polygon mesh representation.

6 Set up an algorithm for converting a given torus with an ellipsoidal cross section to a polygon mesh representation.

https://hemanthrajhemu.github.io

7 Write a program that displays a sphere in the display window and allows the user to switch between solid and wire-frame views of the sphere, translate the sphere along any dimension, rotate the sphere around its center in any direction, and change the size of the sphere (i.e., its radius).

8 Write a program that displays a torus in the display window and allows the user to switch between solid and wire-frame views of the torus, translate the torus along any dimension, rotate the torus around its center in any direction, and change the sizes of the torus' defining properties (i.e., the radius of its cross section ellipse and its axial radius).

9 Write a program that displays a sphere of fixed radius at world coordinate origin and allows the user to adjust the number of longitude and latitude lines used to approximate the sphere's surface as a quadrilateral mesh. The user should also be able to switch between solid and wire-frame views of the sphere. Vary the resolution of the mesh approximation and observe the visual appearance of the sphere in both solid and wire-frame mode.

10 Write a program that displays a cylinder of fixed height and radius at world coordinate origin and allows the user to adjust the number of longitude and latitude lines used to approximate the cylinder's surface as a quadrilateral mesh. The user should also be able to switch between solid and wire-frame views of the cylinder. Vary the resolution of the mesh approximation and observe the visual appearance of the cylinder in both solid and wire-frame mode.

IN MORE DEPTH

1 The material presented in this chapter will allow you to increase the complexity of the representations of the objects in your application by constructing more complex three-dimensional shapes. Choose the most appropriate three-dimensional shapes introduced in this chapter to replace the polygonal approximations of the objects in your application with which you have been working so far. Be sure to include at least a few curved-surface objects, using the GLU and GLUT functions for generating spheres, ellipsoids, and other quadric and cubic surfaces. Use the shaded fill areas to render the objects, not wire-frame views. Choose a reasonable setting for the number of latitude and longitude lines used to generate the polygon mesh approximation to these curved-surface objects. Write routines to call the appropriate functions and display the shapes in the appropriate positions and orientations in the scene. Use techniques in hierarchical modeling to generate objects that are better approximated as a group of these more-primitive shapes if appropriate.

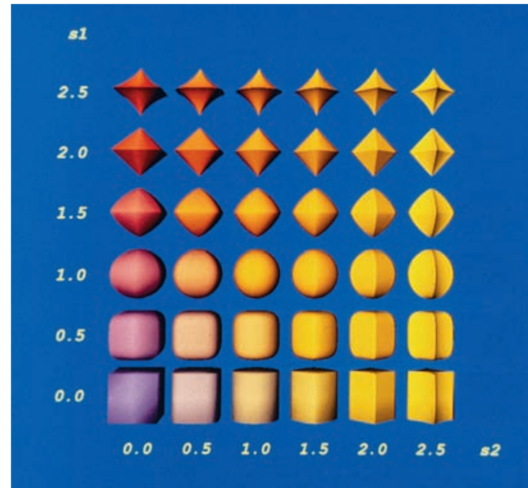
2 In this exercise, you will experiment with varying the resolution of the polygon meshes that serve as the approximations to the curved-surface objects specified in the previous exercise. Choose a minimum number of latitude and longitude lines at which the representation of the objects is minimally acceptable as far as visual appearance goes. Using this as a baseline, render the scene from the previous exercise several times, each time increasing the number of latitude and longitude lines that define the mesh approximations of the objects by some fixed amount. For each setting of resolution, record the amount of time that it takes to render the scene using shaded fill areas to render the objects. Continue doing this until the resolution produces little or no noticeable difference in approximation quality. Then, make a plot of rendering time as a function of resolution parameters (number of latitude and longitude lines) and discuss the properties of the plot. Is there an ideal setting for this scene that balances visual quality with performance?

https://hemanthrajhemu.github.io

This page intentionally left blank

https://hemanthrajhemu.github.io

Three-Dimensional Object Representations Color Plate



Color Plate 10
Superellipsoids plotted with values for parameters s_1 and s_2 ranging from 0.0 to 2.5 and with $r_x = r_y = r_z$.

From *Computer Graphics with OpenGL®*, Fourth Edition, Donald Hearn, M. Pauline Baker, Warren R. Carithers.
Copyright © 2011 by Pearson Education, Inc. Published by Pearson Prentice Hall. All rights reserved.

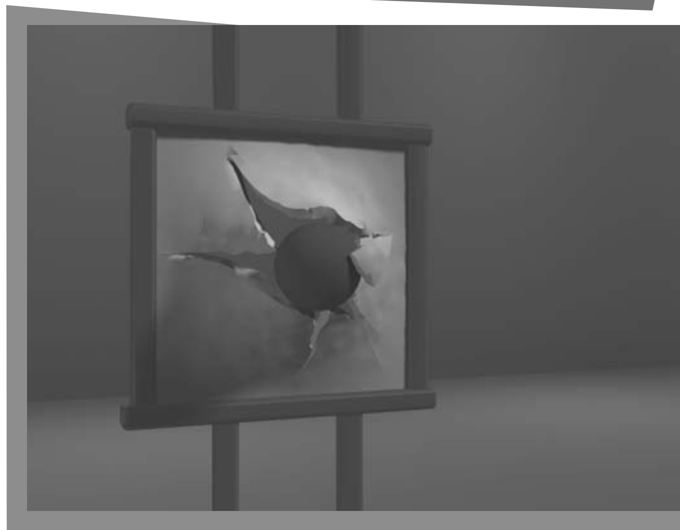
https://hemanthrajhemu.github.io

This page intentionally left blank

https://hemanthrajhemu.github.io

Spline Representations

1 Interpolation and Approximation Splines
2 Parametric Continuity Conditions
3 Geometric Continuity Conditions
4 Spline Specifications
5 Spline Surfaces
6 Trimming Spline Surfaces
7 Cubic-Spline Interpolation Methods
8 Bézier Spline Curves
9 Bézier Surfaces
10 B-Spline Curves
11 B-Spline Surfaces
12 Beta-Splines
13 Rational Splines
14 Conversion Between Spline Representations
15 Displaying Spline Curves and Surfaces
16 OpenGL Approximation-Spline Functions
17 Summary



Splines are another example of boundary representation modeling techniques. In drafting terminology, a spline is a flexible strip used to produce a smooth curve through a designated set of points. Several small weights are distributed along the length of the strip to hold it in position on the drafting table as the curve is drawn. The term *spline curve* originally referred to a curve drawn in this manner. We can mathematically describe such a curve with a piecewise cubic polynomial function whose first and second derivatives are continuous across the various curve sections. In computer graphics, the term **spline curve** now refers to any composite curve formed with polynomial sections satisfying any specified continuity conditions at the boundary of the pieces. A **spline surface** can be described with two sets of spline curves. There are several different kinds of spline specifications that are used in computer-graphics applications. Each individual specification simply refers to a particular type of polynomial with certain prescribed boundary conditions.

From Chapter 14 of *Computer Graphics with OpenGL®*, Fourth Edition, Donald Hearn, M. Pauline Baker, Warren R. Carithers.

Copyright © 2011 by Pearson Education, Inc. Published by Pearson Prentice Hall. All rights reserved.

https://hemanthrajhemu.github.io

Splines are used to design curve and surface shapes, to digitize drawings, and to specify animation paths for the objects or the camera position in a scene. Typical computer-aided design (CAD) applications for splines include the design of automobile bodies, aircraft and spacecraft surfaces, ship hulls, and home appliances.

1 Interpolation and Approximation Splines

We specify a spline curve by giving a set of coordinate positions, called **control points,** which indicate the general shape of the curve. These coordinate positions are then fitted with piecewise-continuous, parametric polynomial functions in one of two ways. When polynomial sections are fitted so that all the control points are connected, as in Figure 1, the resulting curve is said to **interpolate** the set of control points. On the other hand, when the generated polynomial curve is plotted so that some, or all, of the control points are not on the curve path, the resulting curve is said to **approximate** the set of control points (Figure 2). Similar methods are used to construct interpolation or approximation spline surfaces.

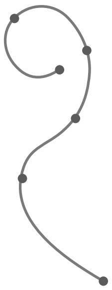


FIGURE 1
A set of six control points interpolated with piecewise continuous polynomial sections.

Interpolation methods are commonly used to digitize drawings or to specify animation paths. Approximation methods are used primarily as design tools to create object shapes. Figure 3 shows the screen display of an approximation spline surface for a design application. Straight lines connect the control-point positions above the surface.

A spline curve or surface is defined, modified, and manipulated with operations on the control points. By interactively selecting spatial positions for the control points, a designer can set up an initial shape. After the polynomial fit is displayed for a given set of control points, the designer can then reposition some of or all the control points to restructure the shape of the object. Geometric transformations (translation, rotation, and scaling) are applied to the object by transforming the control points. In addition, CAD packages sometimes insert extra control points to aid a designer in adjusting the object shapes.

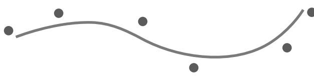


FIGURE 2
A set of six control points approximated with piecewise continuous polynomial sections.

A set of control points forms a boundary for a region of space that is called the **convex hull.** One way to envision the shape of a convex hull for a two-dimensional curve is to imagine a rubber band stretched around the positions of the control

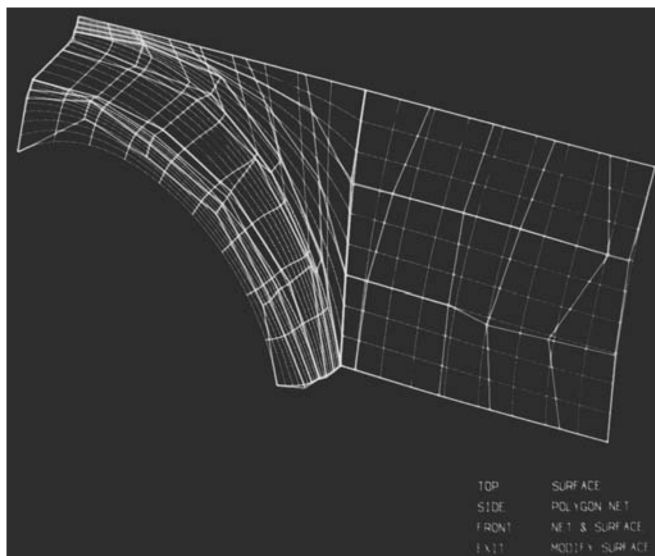


FIGURE 3
An approximation spline surface for a CAD application in automotive design. Surface contours are plotted with polynomial curve sections, and the surface control points are connected with straight-line segments. (*Courtesy of Evans & Sutherland.*)

https://hemanthrajhemu.github.io

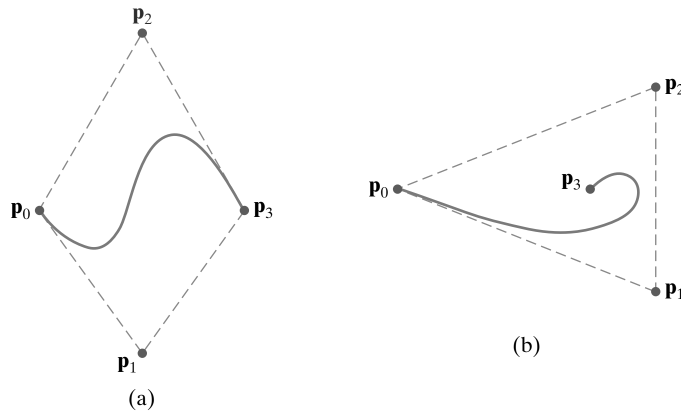


FIGURE 4 Convex-hull shapes (dashed lines) for two sets of control points in the xy plane.

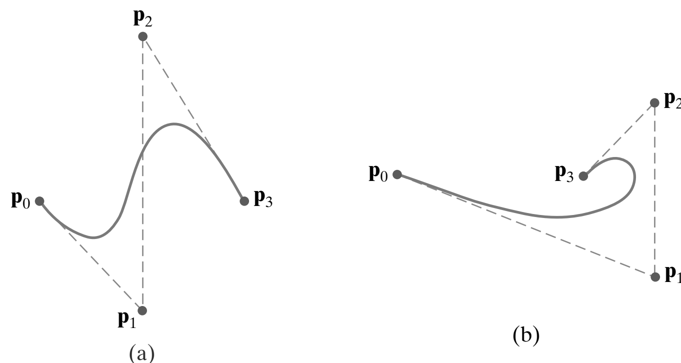


FIGURE 5 Control-graph shapes (dashed lines) for two sets of control points in the xy plane.

points so that each control point is either on the perimeter of this boundary or inside it (Figure 4). Thus, the convex hull for a two-dimensional spline curve is a convex polygon. In three-dimensional space, the convex hull for a set of spline control points forms a convex polyhedron. Convex hulls provide a measure for the deviation of a curve or surface from the region of space near the control points. In most cases, a spline is bounded by its convex hull, which ensures that the object shape follows the control points without erratic oscillations. Also, the convex hull provides a measure of the coordinate extents of a designed curve or surface, so it is useful in clipping and viewing routines.

A polyline connecting the sequence of control points for an approximation spline curve is usually displayed to remind a designer of the control-point positions and ordering. This set of connected line segments is called the **control graph** for the curve. Often the control graph is alluded to as the "control polygon" or the "characteristic polygon," even though the control graph is a polyline and not a polygon. Figure 5 shows the shape of the control graph for the control-point sequences in Figure 4. For a spline surface, two sets of polyline control-point connectors form the edges for the polygon facets in a quadrilateral mesh for the surface control graph, as in Figure 3.

2 Parametric Continuity Conditions

To ensure a smooth transition from one section of a piecewise parametric spline to the next, we can impose various **continuity conditions** at the connection points. If each section of a spline curve is described with a set of parametric coordinate

https://hemanthrajhemu.github.io

functions of the form

$$x = x(u), \qquad y = y(u), \qquad z = z(u), \qquad u_1 \le u \le u_2 \tag{1}$$

we set **parametric continuity** by matching the parametric derivatives of adjoining curve sections at their common boundary.

Zero-order parametric continuity, represented as C^0 continuity, means simply that the curves meet. That is, the values of x, y, and z evaluated at u_2 for the first curve section are equal, respectively, to the values of x, y, and z evaluated at u_1 for the next curve section. **First-order parametric continuity,** referred to as C^1 continuity, means that the first parametric derivatives (tangent lines) of the coordinate functions in Equation 1 for two successive curve sections are equal at their joining point. **Second-order parametric continuity,** or C^2 continuity, means that both the first and second parametric derivatives of the two curve sections are the same at the intersection. Higher-order parametric continuity conditions are defined similarly. Figure 6 shows examples of C^0, C^1, and C^2 continuity.

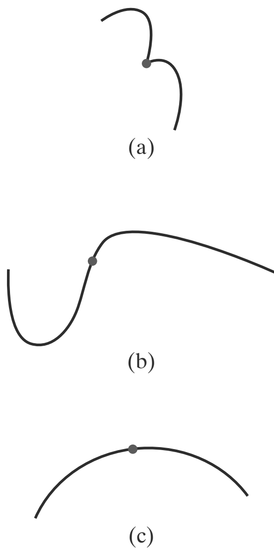


FIGURE 6
Piecewise construction of a curve by joining two curve segments using different orders of continuity: (a) zero-order continuity only, (b) first-order continuity, and (c) second-order continuity.

With second-order parametric continuity, the rates of change of the tangent vectors of connecting sections are equal at their intersection. Thus, the tangent line transitions smoothly from one section of the curve to the next [Figure 6(c)]. With first-order parametric continuity, however, the rate of change of tangent vectors for the two sections can be quite different [Figure 6(b)], so that the general shapes of the two adjacent sections can change abruptly. First-order parametric continuity is often sufficient for digitizing drawings and some design applications, while second-order parametric continuity is useful for setting up animation paths for camera motion and for many precision CAD requirements. A camera traveling along the curve path in Figure 6(b) with equal steps in parameter u would experience an abrupt change in acceleration at the boundary of the two sections, producing a discontinuity in the motion sequence. But if the camera was traveling along the path in Figure 6(c), the frame sequence for the motion would smoothly transition across the boundary.

3 Geometric Continuity Conditions

Another method for joining two successive curve sections is to specify conditions for **geometric continuity.** In this case, we require only that the parametric derivatives of the two sections are proportional to each other at their common boundary, instead of requiring equality.

Zero-order geometric continuity, described as G^0 continuity, is the same as zero-order parametric continuity. That is, two successive curve sections must have the same coordinate position at the boundary point. **First-order geometric continuity,** or G^1 continuity, means that the parametric first derivatives are proportional at the intersection of two successive sections. If we denote the parametric position on the curve as $\mathbf{P}(u)$, the direction of the tangent vector $\mathbf{P}'(u)$, but not necessarily its magnitude, will be the same for two successive curve sections at their common point under G^1 continuity. **Second-order geometric continuity,** or G^2 continuity, means that both the first and second parametric derivatives of the two curve sections are proportional at their boundary. Under G^2 continuity, curvatures of two curve sections will match at the joining position.

A curve generated with geometric continuity conditions is similar to one generated with parametric continuity, but with slight differences in curve shape. Figure 7 provides a comparison of geometric and parametric continuity. With geometric continuity, the curve is pulled toward the section with the greater magnitude for the tangent vector.

https://hemanthrajhemu.github.io

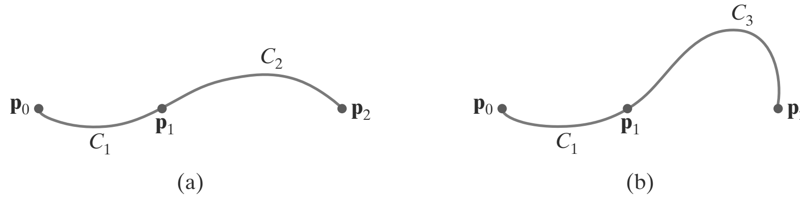


FIGURE 7
Three control points fitted with two curve sections joined with (a) parametric continuity and (b) geometric continuity, where the tangent vector of curve C_3 at point $\mathbf{P}_1$ has a greater magnitude than the tangent vector of curve C_1 at $\mathbf{P}_1$.

4 Spline Specifications

There are three equivalent methods for specifying a particular spline representation, given the degree of the polynomial and the control-point positions: (1) We can state the set of boundary conditions that are imposed on the spline; or (2) we can state the matrix that characterizes the spline; or (3) we can state the set of *blending functions* (or *basis functions*) that determine how specified constraints on the curve are combined to calculate positions along the curve path.

To illustrate these three equivalent specifications, suppose we have the following parametric cubic polynomial representation for the x coordinate along the path of a spline-curve section:

$$x(u) = a_x u^3 + b_x u^2 + c_x u + d_x, \qquad 0 \le u \le 1 \tag{2}$$

Boundary conditions for this curve can be set for the endpoint coordinate positions $x(0)$ and $x(1)$ and for the parametric first derivatives at the endpoints: $x'(0)$ and $x'(1)$. These four boundary conditions are sufficient to determine the values of the four coefficients a_x, b_x, c_x, and d_x.

From the boundary conditions, we can obtain the matrix that characterizes this spline curve by first rewriting Equation 2 as the following matrix product:

$$\begin{aligned} x(u) &= [u^3 \quad u^2 \quad u \quad 1] \begin{bmatrix} a_x \\ b_x \\ c_x \\ d_x \end{bmatrix} \\ &= \mathbf{U} \cdot \mathbf{C} \end{aligned} \tag{3}$$

where $\mathbf{U}$ is the row matrix of powers of parameter u and $\mathbf{C}$ is the coefficient column matrix. Using Equation 3, we can write the boundary conditions in matrix form and solve for the coefficient matrix $\mathbf{C}$ as

$$\mathbf{C} = \mathbf{M}_{\text{spline}} \cdot \mathbf{M}_{\text{geom}} \tag{4}$$

where $\mathbf{M}_{\text{geom}}$ is a four-element column matrix containing the geometric constraint values (boundary conditions) on the spline, and $\mathbf{M}_{\text{spline}}$ is the 4 by 4 matrix that transforms the geometric constraint values to the polynomial coefficients and provides a characterization for the spline curve. Matrix $\mathbf{M}_{\text{geom}}$ contains control-point coordinate values and other geometric constraints that have been specified. Thus, we can substitute the matrix representation for $\mathbf{C}$ into Equation 3 to obtain

$$x(u) = \mathbf{U} \cdot \mathbf{M}_{\text{spline}} \cdot \mathbf{M}_{\text{geom}} \tag{5}$$

The matrix $\mathbf{M}_{\text{spline}}$, characterizing a spline representation, sometimes called the *basis matrix*, is particularly useful for transforming from one spline representation to another.

https://hemanthrajhemu.github.io

Finally, we can expand Equation 5 to obtain a polynomial representation for coordinate x in terms of the geometric constraint parameters g_k, such as the control-point coordinates and slope of the curve at the control points:

$$x(u) = \sum_{k=0}^{3} g_k \cdot \mathrm{BF}_k(u) \tag{6}$$

The polynomials $\mathrm{BF}_k(u)$, for $k = 0, 1, 2, 3$, are called **blending functions** or **basis functions** because they combine (blend) the geometric constraint values to obtain coordinate positions along the curve. In subsequent sections, we explore the features of the various spline curves and surfaces that are useful in computer-graphics applications, including the specification of their matrix and blending-function representations.

5 Spline Surfaces

The usual procedure for defining a spline surface is to specify two sets of spline curves using a mesh of control points over some region of space. If we denote the control-point positions as $\mathbf{p}_{k_u,k_v}$, then any point position on the spline surface can be computed as the product of the spline-curve blending functions as follows:

$$\mathbf{P}(u, v) = \sum_{k_u,k_v} \mathbf{p}_{k_u,k_v} \mathrm{BF}_{k_u}(u)\mathrm{BF}_{k_v}(v) \tag{7}$$

Surface parameters u and v often vary over the range from 0 to 1, but this range depends on the type of spline curves we use. One method for designating the three-dimensional control-point positions is to select height values above a two-dimensional mesh of positions on a ground plane.

6 Trimming Spline Surfaces

In CAD applications, a surface design may require some features that are not implemented just by adjusting control-point positions. For instance, a section of a spline surface may need to be snipped off to fit two design pieces together, or a hole may be needed so that a conduit can pass through the surface. For these applications, graphics packages often provide functions to generate **trimming curves** that can be used to take out sections of a spline surface, as illustrated in Figure 8. Trimming curves are typically defined in parametric uv surface coordinates, and often they must be specified as closed curves.

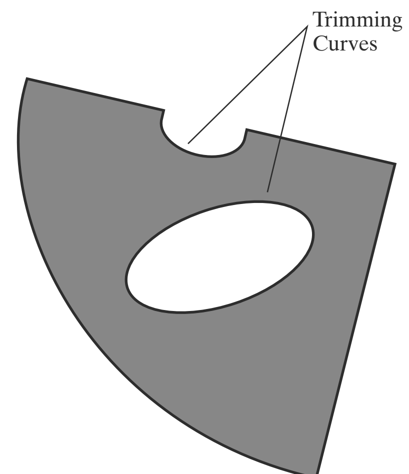


FIGURE 8
Modification of a surface section using trimming curves.

https://hemanthrajhemu.github.io

7 Cubic-Spline Interpolation Methods

This class of splines is most often used to set up paths for object motions or to provide a representation for an existing object or drawing, but interpolation splines are also used sometimes to design object shapes. Cubic polynomials offer a reasonable compromise between flexibility and speed of computation. Compared to higher-order polynomials, cubic splines require less calculations and storage space, and they are more stable. Compared to quadratic polynomials and straight-line segments, cubic splines are more flexible for modeling object shapes.

Given a set of control points, cubic interpolation splines are obtained by fitting the input points with a piecewise cubic polynomial curve that passes through every control point. Suppose that we have $n + 1$ control points specified with coordinates

$$\mathbf{p}_k = (x_k, y_k, z_k), \qquad k = 0, 1, 2, \ldots, n$$

A cubic interpolation fit of these points is illustrated in Figure 9. We can describe the parametric cubic polynomial that is to be fitted between each pair of control points with the following set of equations:

$$\begin{aligned} x(u) &= a_x u^3 + b_x u^2 + c_x u + d_x \\ y(u) &= a_y u^3 + b_y u^2 + c_y u + d_y, \qquad (0 \le u \le 1) \\ z(u) &= a_z u^3 + b_z u^2 + c_z u + d_z \end{aligned} \tag{8}$$

For each of these three equations, we need to determine the values for the four coefficients a, b, c, and d in the polynomial representation for each of the n curve sections between the $n+1$ control points. We do this by setting enough boundary conditions at the control-point positions between curve sections so that we can obtain numerical values for all the coefficients. In the following sections, we discuss common methods for setting the boundary conditions for cubic interpolation splines.

Natural Cubic Splines

One of the first spline curves to be developed for graphics applications is the **natural cubic spline.** This interpolation curve is a mathematical representation of the original drafting spline. We formulate a natural cubic spline by requiring that two adjacent curve sections have the same first and second parametric derivatives at their common boundary. Thus, natural cubic splines have C^2 continuity.

If we have $n + 1$ control points, as in Figure 9, then we have n curve sections with a total of $4n$ polynomial coefficients to be determined. At each of the $n - 1$ interior control points, we have four boundary conditions: The two curve sections on either side of a control point must have the same first and second parametric derivatives at that control point, and each curve must pass through that control point. This gives us $4n - 4$ equations to be satisfied by the $4n$ polynomial coefficients. We obtain an additional equation from the first control point $\mathbf{p}_0$, the position of the beginning of the curve, and another condition from control point $\mathbf{p}_n$, which must be the last point on the curve. However, we still need

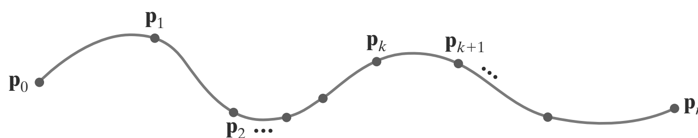


FIGURE 9
A piecewise continuous cubic-spline interpolation of $n + 1$ control points.

https://hemanthrajhemu.github.io

two more conditions to be able to determine values for all the coefficients. One method for obtaining the two additional conditions is to set the second derivatives at $\mathbf{p}_0$ and $\mathbf{p}_n$ equal to 0. Another approach is to add two extra control points (called *dummy points*), one at each end of the original control-point sequence. That is, we add a control point labeled $\mathbf{p}_{-1}$ at the beginning of the curve and a control point labeled $\mathbf{p}_{n+1}$ at the end. Then all the original control points are interior points, and we have the necessary $4n$ boundary conditions.

Although natural cubic splines are a mathematical model for the drafting spline, they have a major disadvantage. If the position of any of the control points is altered, the entire curve is affected. Thus, natural cubic splines allow for no "local control," so that we cannot restructure part of the curve without specifying an entirely new set of control points. For this reason, other representations for a cubic-spline interpolation have been developed.

Hermite Interpolation

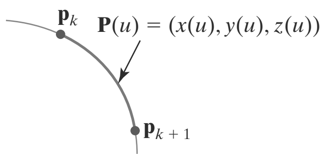


FIGURE 10 Parametric point function $\mathbf{P}(u)$ for a Hermite curve section between control points $\mathbf{p}_k$ and $\mathbf{p}_{k+1}$.

A **Hermite spline** (named after the French mathematician Charles Hermite) is an interpolating piecewise cubic polynomial with a specified tangent at each control point. Unlike the natural cubic splines, Hermite splines can be adjusted locally because each curve section depends only on its endpoint constraints.

If $\mathbf{P}(u)$ represents a parametric cubic point function for the curve section between control points $\mathbf{p}_k$ and $\mathbf{p}_{k+1}$, as shown in Figure 10, then the boundary conditions that define this Hermite curve section are

$$\begin{aligned} \mathbf{P}(0) &= \mathbf{p}_k \\ \mathbf{P}(1) &= \mathbf{p}_{k+1} \\ \mathbf{P}'(0) &= \mathbf{Dp}_k \\ \mathbf{P}'(1) &= \mathbf{Dp}_{k+1} \end{aligned} \tag{9}$$

with $\mathbf{Dp}_k$ and $\mathbf{Dp}_{k+1}$ specifying the values for the parametric derivatives (slope of the curve) at control points $\mathbf{p}_k$ and $\mathbf{p}_{k+1}$, respectively.

We can write the vector equivalent of Equations 8 for this Hermite curve section as

$$\mathbf{P}(u) = \mathbf{a}\,u^3 + \mathbf{b}\,u^2 + \mathbf{c}\,u + \mathbf{d}, \qquad 0 \le u \le 1 \tag{10}$$

where the x component of $\mathbf{P}$(u) is $x(u) = a_x\,u^3 + b_x\,u^2 + c_x\,u + d_x$, and similarly for the y and z components. The matrix equivalent of Equation 10 is

$$\mathbf{P}(u) = [u^3 \quad u^2 \quad u \quad 1] \cdot \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \end{bmatrix} \tag{11}$$

and the derivative of the point function can be expressed as

$$\mathbf{P}'(u) = [3u^2 \quad 2u \quad 1 \quad 0] \cdot \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \end{bmatrix} \tag{12}$$

Substituting endpoint values 0 and 1 for parameter u into the preceding two equations, we can express the Hermite boundary conditions 9 in the matrix form

$$\begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{Dp}_k \\ \mathbf{Dp}_{k+1} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \end{bmatrix} \tag{13}$$

https://hemanthrajhemu.github.io

Solving this equation for the polynomial coefficients, we get

$$\begin{bmatrix} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix}^{-1} \cdot \begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{Dp}_k \\ \mathbf{Dp}_{k+1} \end{bmatrix}$$

$$= \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{Dp}_k \\ \mathbf{Dp}_{k+1} \end{bmatrix}$$

$$= \mathbf{M}_H \cdot \begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{Dp}_k \\ \mathbf{Dp}_{k+1} \end{bmatrix} \tag{14}$$

where $\mathbf{M}_H$, the Hermite matrix, is the inverse of the boundary constraint matrix. Equation 11 can thus be written in terms of the boundary conditions as

$$\mathbf{P}(u) = [u^3 \quad u^2 \quad u \quad 1] \cdot \mathbf{M}_H \cdot \begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{Dp}_k \\ \mathbf{Dp}_{k+1} \end{bmatrix} \tag{15}$$

Finally, we can determine expressions for the polynomial Hermite blending functions, $H_k(u)$ for $k = 0, 1, 2, 3$, by carrying out the matrix multiplications in Equation 15 and collecting coefficients for the boundary constraints to obtain the polynomial form

$$\begin{aligned} \mathbf{P}(u) &= \mathbf{p}_k(2u^3 - 3u^2 + 1) + \mathbf{p}_{k+1}(-2u^3 + 3u^2) + \mathbf{Dp}_k(u^3 - 2u^2 + u) \\ &\qquad + \mathbf{Dp}_{k+1}(u^3 - u^2) \\ &= \mathbf{p}_k H_0(u) + \mathbf{p}_{k+1} H_1 + \mathbf{Dp}_k H_2 + \mathbf{Dp}_{k+1} H_3 \end{aligned} \tag{16}$$

Figure 11 shows the shape of the four Hermite blending functions.

Hermite polynomials can be useful for some digitizing applications, where it may not be too difficult to specify or approximate the curve slopes. But for most problems in computer graphics, it is more useful to generate spline curves without requiring input values for curve slopes or other geometric information, in addition to control-point coordinates. Cardinal splines and Kochanek-Bartels splines, discussed in the following two sections, are variations on the Hermite splines that do not require input values for the curve derivatives at the control points. Procedures for these splines compute parametric derivatives from the coordinate positions of the control points.

Cardinal Splines

As with Hermite splines, the **cardinal splines** are interpolating piecewise cubic polynomials with specified endpoint tangents at the boundary of each curve section. The difference is that we do not input the values for the endpoint tangents. For a cardinal spline, the slope at a control point is calculated from the coordinates of the two adjacent control points.

https://hemanthrajhemu.github.io

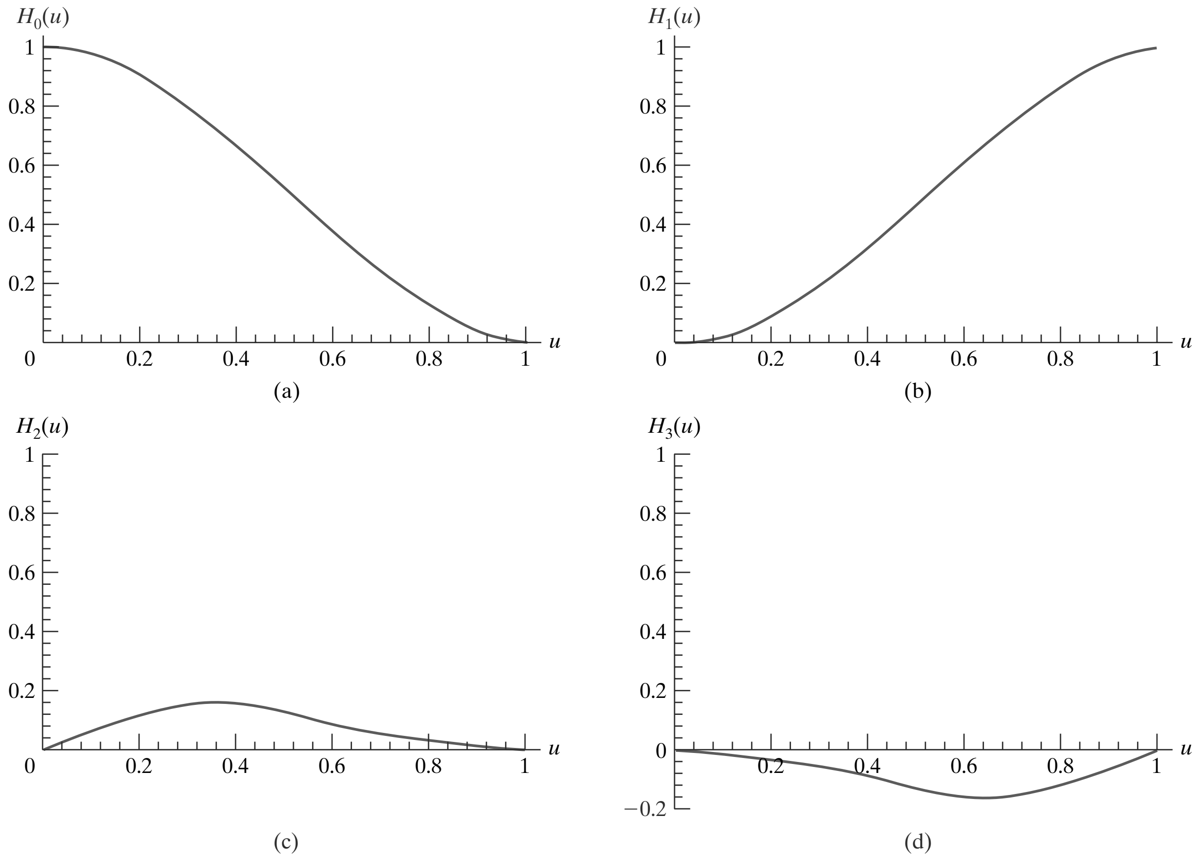


FIGURE 11
The Hermite blending functions.

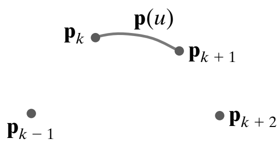


FIGURE 12
Parametric point function $\mathbf{P}(u)$ for a cardinal-spline section between control points $\mathbf{p}_k$ and $\mathbf{p}_{k+1}$.

A cardinal spline section is completely specified with four consecutive control-point positions. The middle two control points are the section endpoints, and the other two points are used in the calculation of the endpoint slopes. If we take $\mathbf{P}(u)$ as the representation for the parametric cubic point function for the curve section between control points $\mathbf{p}_k$ and $\mathbf{p}_{k+1}$, as in Figure 12, then the four control points from $\mathbf{p}_{k-1}$ to $\mathbf{p}_{k+1}$ are used to set the boundary conditions for the cardinal-spline section as

$$\begin{aligned}\mathbf{P}(0) &= \mathbf{p}_k \\ \mathbf{P}(1) &= \mathbf{p}_{k+1} \\ \mathbf{P}'(0) &= \frac{1}{2}(1-t)(\mathbf{p}_{k+1} - \mathbf{p}_{k-1}) \\ \mathbf{P}'(1) &= \frac{1}{2}(1-t)(\mathbf{p}_{k+2} - \mathbf{p}_k)\end{aligned} \tag{17}$$

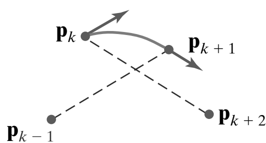


FIGURE 13
Tangent vectors at the endpoints of a cardinal-spline section are parallel to the chords formed with neighboring control points (dashed lines).

Thus, the slopes at control points $\mathbf{p}_k$ and $\mathbf{p}_{k+1}$ are taken to be proportional, respectively, to the chords $\overline{\mathbf{p}_{k-1}\mathbf{p}_{k+1}}$ and $\overline{\mathbf{p}_k\mathbf{p}_{k+2}}$ (Figure 13). Parameter t is called the **tension** parameter because it controls how loosely or tightly the cardinal spline fits the input control points. Figure 14 illustrates the shape of a cardinal curve for very small and very large values of tension t. When $t = 0$, this class of curves is referred to as **Catmull-Rom splines,** or **Overhauser splines.**

https://hemanthrajhemu.github.io

Using methods similar to those for Hermite splines, we can convert the boundary conditions 17 into the matrix form

$$\mathbf{P}(u) = [u^3 \quad u^2 \quad u \quad 1] \cdot \mathbf{M}_C \cdot \begin{bmatrix} \mathbf{p}_{k-1} \\ \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{p}_{k+2} \end{bmatrix} \tag{18}$$

where the cardinal matrix is

$$\mathbf{M}_C = \begin{bmatrix} -s & 2-s & s-2 & s \\ 2s & s-3 & 3-2s & -s \\ -s & 0 & s & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{19}$$

with $s = (1-t)/2$.

Expanding Equation 18 into polynomial form, we have

$$\begin{aligned} \mathbf{P}(u) &= \mathbf{p}_{k-1}(-s\,u^3 + 2s\,u^2 - s\,u) + \mathbf{p}_k[(2-s)u^3 + (s-3)u^2 + 1] \\ &\quad + \mathbf{p}_{k+1}[(s-2)u^3 + (3-2s)u^2 + s\,u] + \mathbf{p}_{k+2}(s\,u^3 - s\,u^2) \\ &= \mathbf{p}_{k-1}\,\text{CAR}_0(u) + \mathbf{p}_k\,\text{CAR}_1(u) + \mathbf{p}_{k+1}\,\text{CAR}_2(u) + \mathbf{p}_{k+2}\,\text{CAR}_3(u) \end{aligned} \tag{20}$$

where the polynomials $\text{CAR}_k(u)$ for $k = 0, 1, 2, 3$ are the cardinal-spline blending (basis) functions. Figure 15 gives a plot of the basis functions for cardinal splines with $t = 0$.

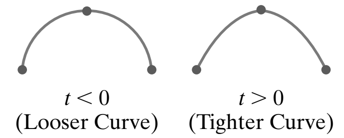


FIGURE 14
Effect of the tension parameter on the shape of a cardinal-spline section.

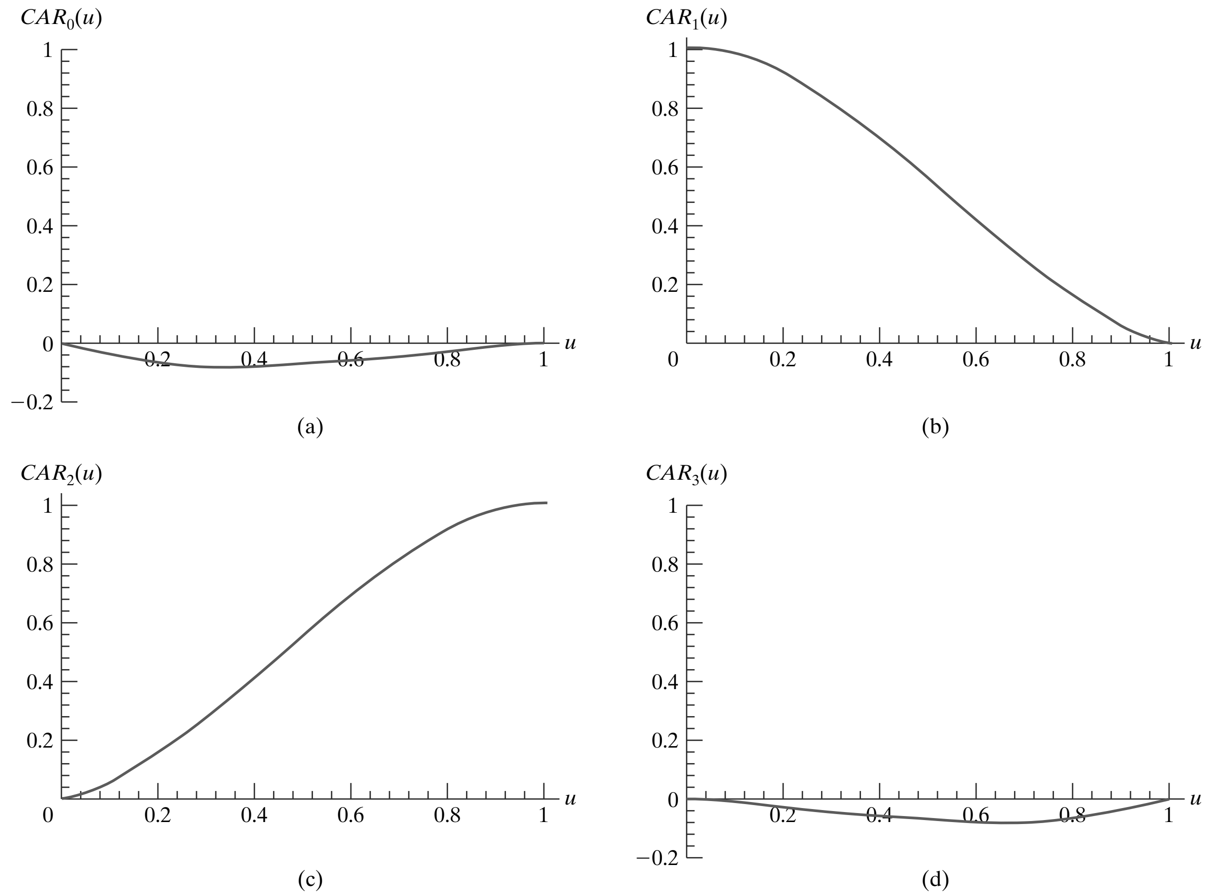


FIGURE 15
The cardinal-spline blending functions for $t = 0$ ($s = 0.5$).

https://hemanthrajhemu.github.io

Examples of curves produced with the cardinal-spline blending functions are given in Figures 16, 17, and 18. In Figure 16, four cardinal-spline sections are plotted to form a closed curve. The first curve section is generated using the control-point set $\{\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3\}$, the second curve is produced with the control-point set $\{\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_0\}$, the third curve section has control points $\{\mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_0, \mathbf{p}_1\}$, and the final curve section has control points $\{\mathbf{p}_3, \mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2\}$. In Figure 17, a closed curve is obtained with a single cardinal-spline section by setting the position of the third control point to the coordinate position of the

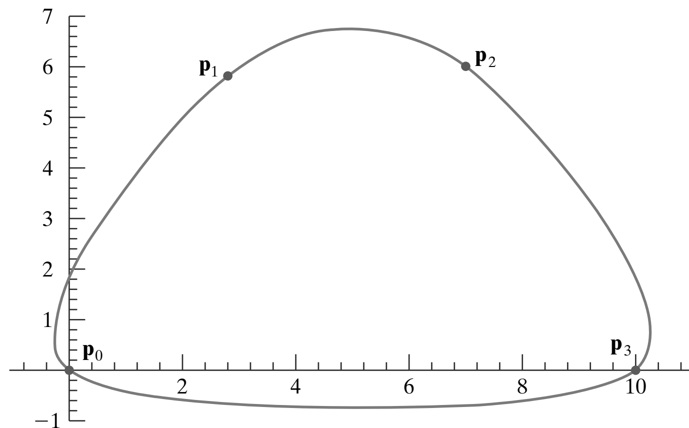


FIGURE 16
A closed curve with four cardinal-spline sections, obtained with a cyclic permutation of the control points and with tension parameter $t = 0$.

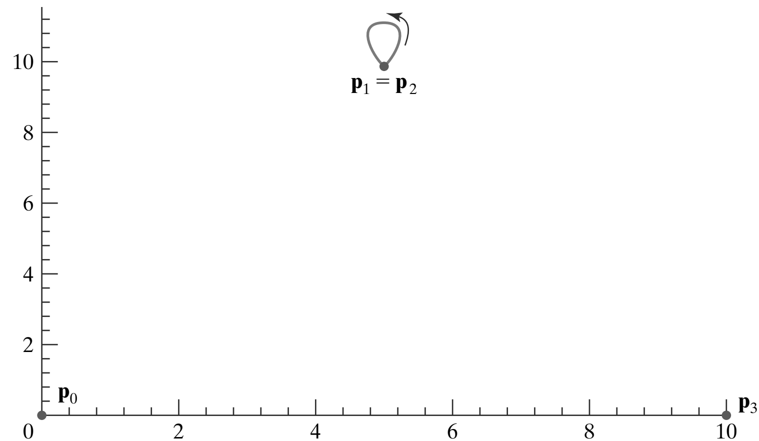


FIGURE 17
A cardinal- spline loop produced with curve endpoints at the same coordinate position. The tension parameter is set to the value 0.

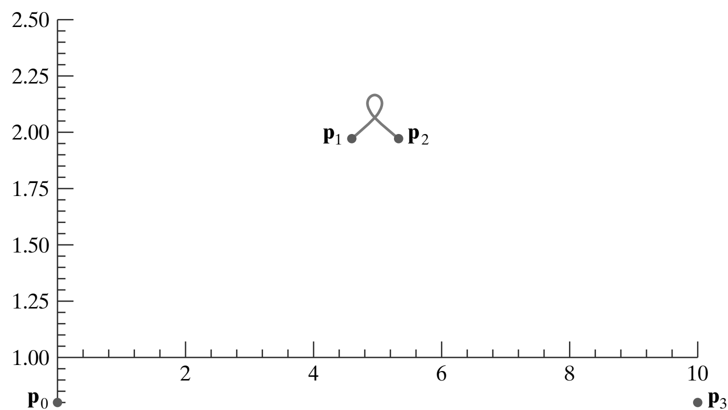


FIGURE 18
A self-intersecting cardinal-spline curve section produced with closely spaced curve endpoint positions. The tension parameter is set to the value 0.

https://hemanthrajhemu.github.io

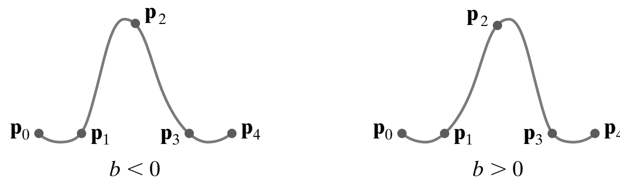


FIGURE 19 Effect of the bias parameter on the shape of a Kochanek-Bartels spline section.

second control point. In Figure 18, a self-intersecting cardinal-spline section is produced by setting the position of the third control point very near the coordinate position of the second control point. The resulting self-intersection is due to the constraints on the curve slope at the endpoints $\mathbf{p}_1$ and $\mathbf{p}_2$.

Kochanek-Bartels Splines

These interpolating cubic polynomials are extensions of the cardinal splines. Two additional parameters are introduced into the constraint equations defining **Kochanek-Bartels splines** to provide further flexibility in adjusting the shapes of curve sections.

Given four consecutive control points, labeled $\mathbf{p}_{k-1}$, $\mathbf{p}_k$, $\mathbf{p}_{k+1}$, and $\mathbf{p}_{k+2}$, we define the boundary conditions for a Kochanek-Bartels curve section between $\mathbf{p}_k$ and $\mathbf{p}_{k+1}$ as

$$\begin{aligned}
\mathbf{P}(0) &= \mathbf{p}_k \\
\mathbf{P}(1) &= \mathbf{p}_{k+1} \\
\mathbf{P}'(0)_{\text{in}} &= \frac{1}{2}(1-t)[(1+b)(1-c)(\mathbf{p}_k - \mathbf{p}_{k-1}) \\
&\qquad + (1-b)(1+c)(\mathbf{p}_{k+1} - \mathbf{p}_k)] \\
\mathbf{P}'(1)_{\text{out}} &= \frac{1}{2}(1-t)[(1+b)(1+c)(\mathbf{p}_{k+1} - \mathbf{p}_k) \\
&\qquad + (1-b)(1-c)(\mathbf{p}_{k+2} - \mathbf{p}_{k+1})]
\end{aligned} \tag{21}$$

where t is the **tension** parameter, b is the **bias** parameter, and c is the **continuity** parameter. In the Kochanek-Bartels formulation, parametric derivatives might not be continuous across section boundaries.

Tension parameter t has the same interpretation as in the cardinal spline formulation; that is, it controls the looseness or tightness of the curve sections. Bias, b, is used to adjust the curvature at each end of a section so that curve sections can be skewed toward one end or the other (Figure 19). Parameter c controls the continuity of the tangent vector across the boundaries of sections. If c is assigned a nonzero value, there is a discontinuity in the slope of the curve across section boundaries.

Kochanek-Bartels splines were designed to model animation paths. In particular, abrupt changes in the motion of an object can be simulated with nonzero values for parameter c. These motion changes are used in cartoon animations, for example, when a cartoon character stops quickly, changes direction, or collides with some other object.

8 Bézier Spline Curves

This spline approximation method was developed by the French engineer Pierre Bézier for use in the design of Renault automobile bodies. **Bézier splines** have a number of properties that make them highly useful and convenient for curve and

https://hemanthrajhemu.github.io

surface design. They are also easy to implement. For these reasons, Bézier splines are widely available in various CAD systems, in general graphics packages, and in assorted drawing and painting packages.

In general, a Bézier curve section can be fitted to any number of control points, although some graphic packages limit the number of control points to four. The degree of the Bézier polynomial is determined by the number of control points to be approximated and their relative position. As with the interpolation splines, we can specify the Bézier curve path in the vicinity of the control points using blending functions, a characterizing matrix, or boundary conditions. For general Bézier curves, with no restrictions on the number of control points, the blending-function specification is the most convenient representation.

Bézier Curve Equations

We first consider the general case of $n + 1$ control-point positions, denoted as $\mathbf{p}_k = (x_k, y_k, z_k)$, with k varying from 0 to n. These coordinate points are blended to produce the following position vector $\mathbf{P}(u)$, which describes the path of an approximating Bézier polynomial function between $\mathbf{p}_0$ and $\mathbf{p}_n$:

$$\mathbf{P}(u) = \sum_{k=0}^{n} \mathbf{p}_k \, \mathrm{BEZ}_{k,n}(u), \qquad 0 \le u \le 1 \tag{22}$$

The Bézier blending functions $\mathrm{BEZ}_{k,n}(u)$ are the *Bernstein polynomials*

$$\mathrm{BEZ}_{k,n}(u) = C(n, k)u^k(1 - u)^{n-k} \tag{23}$$

where parameters $C(n, k)$ are the binomial coefficients

$$C(n, k) = \frac{n!}{k!(n - k)!} \tag{24}$$

Equation 22 represents a set of three parametric equations for the individual curve coordinates:

$$\begin{aligned} x(u) &= \sum_{k=0}^{n} x_k \, \mathrm{BEZ}_{k,n}(u) \\ y(u) &= \sum_{k=0}^{n} y_k \, \mathrm{BEZ}_{k,n}(u) \\ z(u) &= \sum_{k=0}^{n} z_k \, \mathrm{BEZ}_{k,n}(u) \end{aligned} \tag{25}$$

In most cases, a Bézier curve is a polynomial of a degree that is one less than the designated number of control points: Three points generate a parabola, four points a cubic curve, and so forth. Figure 20 demonstrates the appearance of some Bézier curves for various selections of control points in the xy plane ($z = 0$). With certain control-point placements, however, we obtain degenerate Bézier polynomials. For example, a Bézier curve generated with three collinear control points is a straight-line segment; and a set of control points that are all at the same coordinate position produce a Bézier "curve" that is a single point.

Recursive calculations can be used to obtain successive binomial-coefficient values as

$$C(n, k) = \frac{n - k + 1}{k} C(n, k - 1) \tag{26}$$

for $n \ge k$. Also, the Bézier blending functions satisfy the recursive relationship

$$\mathrm{BEZ}_{k,n}(u) = (1 - u)\mathrm{BEZ}_{k,n-1}(u) + u \, \mathrm{BEZ}_{k-1,n-1}(u), \qquad n > k \ge 1 \tag{27}$$

with $\mathrm{BEZ}_{k,k} = u^k$ and $\mathrm{BEZ}_{0,k} = (1 - u)^k$.

https://hemanthrajhemu.github.io

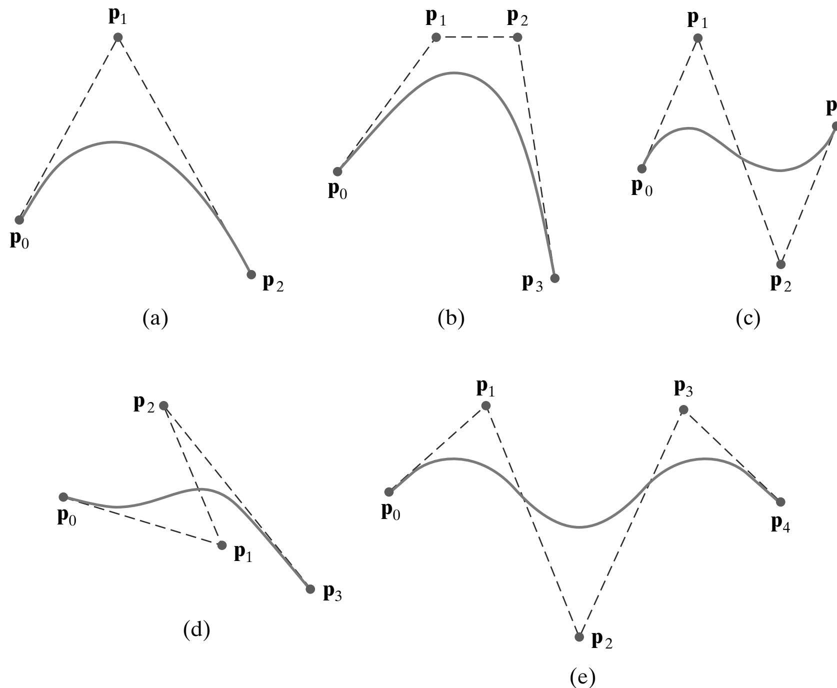


FIGURE 20
Examples of two-dimensional Bézier curves generated with three, four, and five control points. Dashed lines connect the control-point positions.

Example Bézier Curve-Generating Program

An implementation for calculating the Bézier blending functions and generating a two-dimensional, cubic Bézier-spline curve is given in the following program. Four control points are defined in the xy plane, and 1000 pixel positions are plotted along the curve path using a pixel width of 4. Values for the binomial coefficients are calculated in procedure `binomialCoeffs`, and coordinate positions along the curve path are calculated in procedure `computeBezPt`. These values are passed to procedure `bezier`, and pixel positions are plotted using the OpenGL point-plotting routines. Alternatively, we could have approximated the curve path with straight-line sections, using fewer points. More efficient methods for generating coordinate positions along the path of a spline curve are explored in Section 15. For this example, the world-coordinate limits are set so that only the curve points are displayed within the viewport (Figure 21). If we also wanted to plot the control-point positions, the control graph, or the convex hull, we would need to extend the limits of the world-coordinate clipping window.

```
#include <GL/glut.h>
#include <stdlib.h>
#include <math.h>

/*  Set initial size of the display window.  */
GLsizei winWidth = 600, winHeight = 600;

/*  Set size of world-coordinate clipping window.  */
GLfloat xwcMin = -50.0, xwcMax = 50.0;
GLfloat ywcMin = -50.0, ywcMax = 50.0;
```


https://hemanthrajhemu.github.io

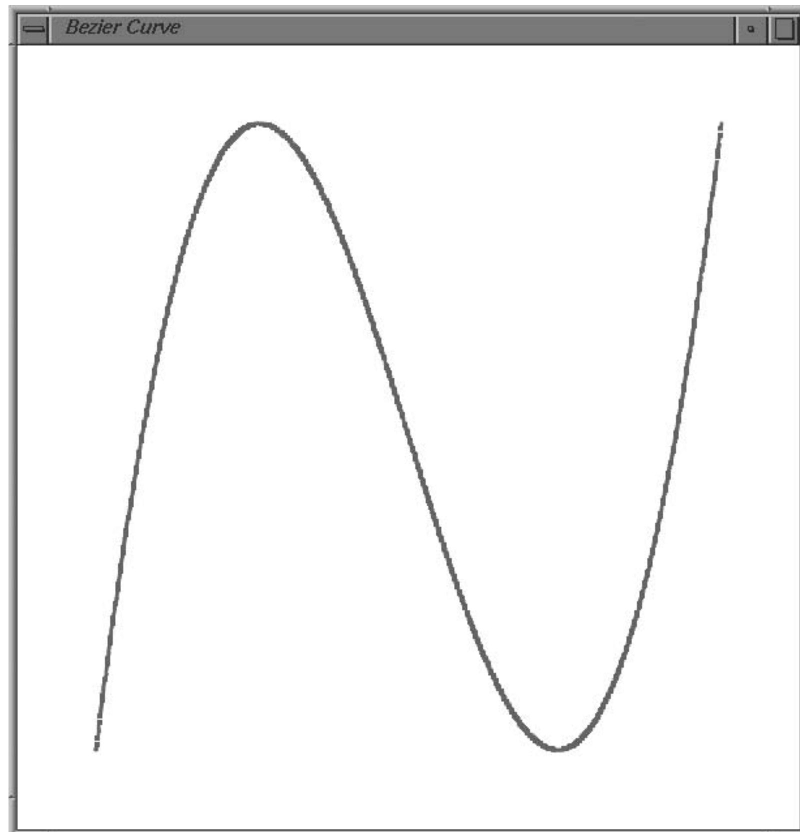


FIGURE 21
A Bézier curve displayed by the example program.

```
class wcPt3D {
   public:
      GLfloat x, y, z;
};

void init (void)
{
   /*  Set color of display window to white.  */
   glClearColor (1.0, 1.0, 1.0, 0.0);
}

void plotPoint (wcPt3D bezCurvePt)
{
    glBegin (GL_POINTS);
        glVertex2f (bezCurvePt.x, bezCurvePt.y);
    glEnd ( );
}

/*  Compute binomial coefficients C for given value of n.  */
void binomialCoeffs (GLint n, GLint * C)
{
   GLint k, j;

   for (k = 0;  k <= n;  k++) {
      /*  Compute n!/(k!(n - k)!).  */
```


https://hemanthrajhemu.github.io

```
      C [k] = 1;
      for (j = n;  j >= k + 1;  j--)
        C [k] *= j;
      for (j = n - k;  j >= 2;  j--)
        C [k] /= j;
   }
}

void computeBezPt (GLfloat u, wcPt3D * bezPt, GLint nCtrlPts,
                    wcPt3D * ctrlPts, GLint * C)
{
   GLint k, n = nCtrlPts - 1;
   GLfloat bezBlendFcn;

   bezPt->x = bezPt->y = bezPt->z = 0.0;

   /*  Compute blending functions and blend control points. */
   for (k = 0; k < nCtrlPts; k++) {
      bezBlendFcn = C [k] * pow (u, k) * pow (1 - u, n - k);
      bezPt->x += ctrlPts [k].x * bezBlendFcn;
      bezPt->y += ctrlPts [k].y * bezBlendFcn;
      bezPt->z += ctrlPts [k].z * bezBlendFcn;
   }
}

void bezier (wcPt3D * ctrlPts, GLint nCtrlPts, GLint nBezCurvePts)
{
   wcPt3D bezCurvePt;
   GLfloat u;
   GLint *C, k;

   /*  Allocate space for binomial coefficients  */
   C = new GLint [nCtrlPts];

   binomialCoeffs (nCtrlPts - 1, C);
   for (k = 0;  k <= nBezCurvePts;  k++) {
      u = GLfloat (k) / GLfloat (nBezCurvePts);
      computeBezPt (u, &bezCurvePt, nCtrlPts, ctrlPts, C);
      plotPoint (bezCurvePt);
   }
   delete [ ] C;
}

void displayFcn (void)
{
   /*  Set example number of control points and number of
    *  curve positions to be plotted along the Bezier curve.
    */
   GLint nCtrlPts = 4, nBezCurvePts = 1000;

   wcPt3D ctrlPts [4] = { {-40.0, -40.0, 0.0}, {-10.0, 200.0, 0.0},
                          {10.0, -200.0, 0.0}, {40.0, 40.0, 0.0} };

   glClear (GL_COLOR_BUFFER_BIT);   //  Clear display window.
```


https://hemanthrajhemu.github.io

```
   glPointSize (4);
   glColor3f (1.0, 0.0, 0.0);      //  Set point color to red.

   bezier (ctrlPts, nCtrlPts, nBezCurvePts);
   glFlush ( );
}

void winReshapeFcn (GLint newWidth, GLint newHeight)
{
   /*  Maintain an aspect ratio of 1.0.  */
   glViewport (0, 0, newHeight, newHeight);

   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ( );

   gluOrtho2D (xwcMin, xwcMax, ywcMin, ywcMax);

   glClear (GL_COLOR_BUFFER_BIT);
}

void main (int argc, char** argv)
{
   glutInit (&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutInitWindowPosition (50, 50);
   glutInitWindowSize (winWidth, winHeight);
   glutCreateWindow ("Bezier Curve");

   init ( );
   glutDisplayFunc (displayFcn);
   glutReshapeFunc (winReshapeFcn);

   glutMainLoop ( );
}
```

Properties of Bézier Curves

A very useful property of a Bézier curve is that the curve connects the first and last control points. Thus, a basic characteristic of any Bézier curve is that

$$\begin{aligned} \mathbf{P}(0) &= \mathbf{p}_0 \\ \mathbf{P}(1) &= \mathbf{p}_n \end{aligned} \tag{28}$$

Values for the parametric first derivatives of a Bézier curve at the endpoints can be calculated from control-point coordinates as

$$\begin{aligned} \mathbf{P}'(0) &= -n\mathbf{p}_0 + n\mathbf{p}_1 \\ \mathbf{P}'(1) &= -n\mathbf{p}_{n-1} + n\mathbf{p}_n \end{aligned} \tag{29}$$

From these expressions, we see that the slope at the beginning of the curve is along the line joining the first two control points, and the slope at the end of the curve is along the line joining the last two endpoints. Similarly, the parametric second derivatives of a Bézier curve at the endpoints are calculated as

$$\begin{aligned} \mathbf{P}''(0) &= n(n-1)[(\mathbf{p}_2 - \mathbf{p}_1) - (\mathbf{p}_1 - \mathbf{p}_0)] \\ \mathbf{P}''(1) &= n(n-1)[(\mathbf{p}_{n-2} - \mathbf{p}_{n-1}) - (\mathbf{p}_{n-1} - \mathbf{p}_n)] \end{aligned} \tag{30}$$

https://hemanthrajhemu.github.io

Another important property of any Bézier curve is that it lies within the convex hull (convex polygon boundary) of the control points. This follows from the fact that the Bézier blending functions are all positive and their sum is always 1:

$$\sum_{k=0}^{n} \text{BEZ}_{k,n}(u) = 1 \tag{31}$$

so that any curve position is simply the weighted sum of the control-point positions. The convex-hull property for a Bézier curve ensures that the polynomial smoothly follows the control points without erratic oscillations.

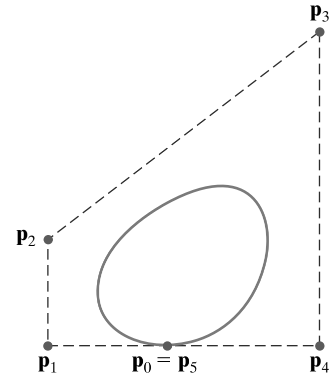


FIGURE 22
A closed Bézier curve generated by specifying the first and last control points at the same location.

Design Techniques Using Bézier Curves

A closed Bézier curve is generated when we set the last control-point position to the coordinate position of the first control point, as in the example shown in Figure 22. Also, specifying multiple control points at a single coordinate position gives more weight to that position. In Figure 23, a single coordinate position is input as two control points, and the resulting curve is pulled nearer to this position.

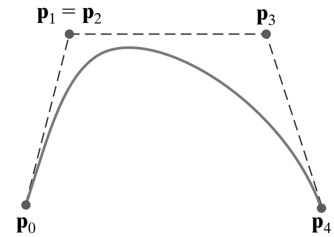


FIGURE 23
A Bézier curve can be made to pass closer to a given coordinate position by assigning multiple control points to that position.

We can fit a Bézier curve to any number of control points, but this requires the calculation of polynomial functions of higher degree. When complicated curves are to be generated, they can be formed by piecing together several Bézier sections of lower degree. Generating smaller Bézier-curve sections also gives us better local control over the shape of the curve. Because Bézier curves connect the first and last control points, it is easy to match curve sections (zero-order continuity). Also, Bézier curves have the important property that the tangent to the curve at an endpoint is along the line joining that endpoint to the adjacent control point. Therefore, to obtain first-order continuity between curve sections, we can pick control points $\mathbf{p}_{0'}$ and $\mathbf{p}_{1'}$ for the next curve section to be along the same straight line as control points $\mathbf{p}_{n-1}$ and $\mathbf{p}_n$ of the preceding section (Figure 24). If the first curve section has n control points and the next curve section has n' control points, then we match curve tangents by placing control point $\mathbf{p}_{1'}$ at the position

$$\mathbf{p}_{1'} = \mathbf{p}_n + \frac{n}{n'}(\mathbf{p}_n - \mathbf{p}_{n-1}) \tag{32}$$

To simplify the placement of $\mathbf{p}_{1'}$, we can require only geometric continuity and place $\mathbf{p}_{1'}$ anywhere along the line of $\mathbf{p}_{n-1}$ and $\mathbf{p}_n$.

We obtain C^2 continuity by using the expressions in Equations 30 to match parametric second derivatives for two adjacent Bézier sections. This establishes a coordinate position for control point $\mathbf{p}_{2'}$, in addition to the fixed positions for

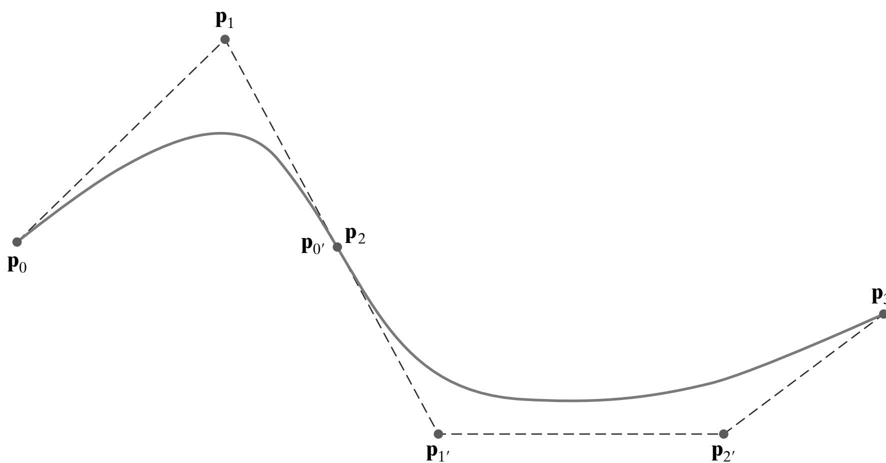


FIGURE 24
Piecewise approximation curve formed with two Bézier sections. Zero-order and first-order continuity is attained between the two curve sections by setting $\mathbf{p}_{0'} = \mathbf{p}_2$ and by setting $\mathbf{p}_{1'}$ along the line formed with points $\mathbf{p}_1$ and $\mathbf{p}_2$.

https://hemanthrajhemu.github.io

$\mathbf{p}_{0'}$ and $\mathbf{p}_{1'}$ that we need for C^0 and C^1 continuity. However, requiring second-order continuity for Bézier curve sections can be unnecessarily restrictive. This is particularly true with cubic curves, which have only four control points per section. In this case, second-order continuity fixes the position of the first three control points and leaves us only one point that we can use to adjust the shape of the curve segment.

Cubic Bézier Curves

Many graphics packages provide functions for displaying only cubic splines. This allows reasonable design flexibility while avoiding the increased calculations needed with higher-order polynomials. Cubic Bézier curves are generated with four control points. The four blending functions for cubic Bézier curves, obtained by substituting $n = 3$ into Equation 23, are

$$\begin{aligned} \mathrm{BEZ}_{0,3} &= (1-u)^3 \\ \mathrm{BEZ}_{1,3} &= 3u(1-u)^2 \\ \mathrm{BEZ}_{2,3} &= 3u^2(1-u) \\ \mathrm{BEZ}_{3,3} &= u^3 \end{aligned} \tag{33}$$

Plots of the four cubic Bézier blending functions are given in Figure 25. The form of the blending functions determine how the control points influence the shape of the curve for values of parameter u over the range from 0 to 1. At $u = 0$,

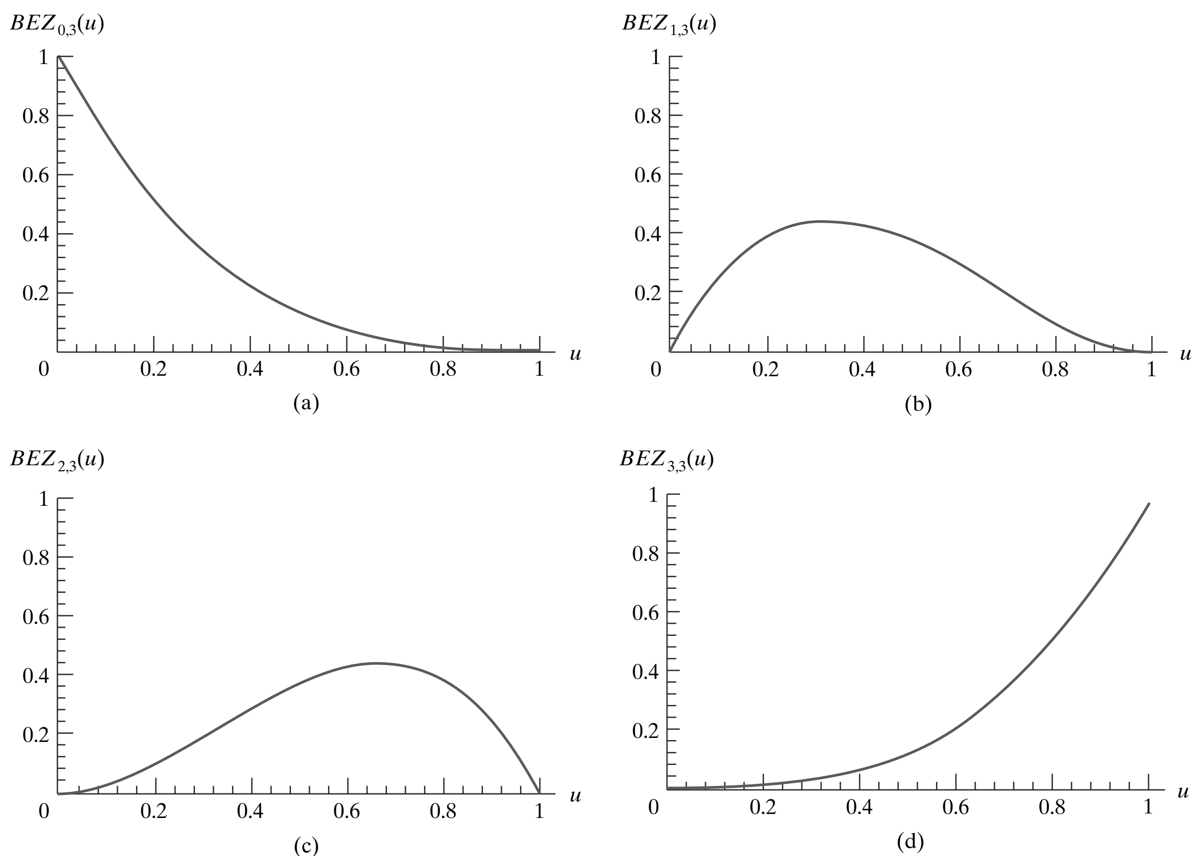


FIGURE 25
The four Bézier blending functions for cubic curves ($n = 3$).

https://hemanthrajhemu.github.io

the only nonzero blending function is $\text{BEZ}_{0,3}$, which has the value 1. At $u = 1$, the only nonzero function is $\text{BEZ}_{3,3}(1) = 1$. Thus, a cubic Bézier curve always begins at control point $\mathbf{p}_0$ and ends at the position of control point $\mathbf{p}_3$. The other functions, $\text{BEZ}_{1,3}$ and $\text{BEZ}_{2,3}$, influence the shape of the curve at intermediate values of the parameter u so that the resulting curve tends toward the points $\mathbf{p}_1$ and $\mathbf{p}_2$. Blending function $\text{BEZ}_{1,3}$ is maximized at $u = 1/3$, and $\text{BEZ}_{2,3}$ is maximized at $u = 2/3$.

We note in Figure 25 that each of the four blending functions is nonzero over the entire range of parameter u between the endpoint positions. Thus, Bézier curves do not allow for *local control* of the curve shape. If we reposition any one of the control points, the entire curve is affected.

At the end positions of the cubic Bézier curve, the parametric first derivatives (slopes) are

$$\mathbf{P}'(0) = 3(\mathbf{p}_1 - \mathbf{p}_0), \qquad \mathbf{P}'(1) = 3(\mathbf{p}_3 - \mathbf{p}_2)$$

and the parametric second derivatives are

$$\mathbf{P}''(0) = 6(\mathbf{p}_0 - 2\mathbf{p}_1 + \mathbf{p}_2), \qquad \mathbf{P}''(1) = 6(\mathbf{p}_1 - 2\mathbf{p}_2 + \mathbf{p}_3)$$

We can construct complex spline curves using a series of cubic-Bézier sections. Using expressions for the parametric derivatives, we can equate curve tangents to attain C^1 continuity between the curve sections. In addition, we could use the expressions for the second derivatives to obtain C^2 continuity, although this leaves us with no options for the placement of the first three control points.

A matrix formulation for the cubic-Bézier curve function is obtained by expanding the polynomial expressions for the blending functions and restructuring the equations as

$$\mathbf{P}(u) = [u^3 \quad u^2 \quad u \quad 1] \cdot \mathbf{M}_{\text{Bez}} \cdot \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{bmatrix} \tag{34}$$

where the **Bézier matrix** is

$$\mathbf{M}_{\text{Bez}} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{35}$$

We could also introduce additional parameters to allow adjustment of curve "tension" and "bias," as we did with the interpolating splines. But more versatile types of splines (such as B-splines and beta-splines, discussed later in this chapter) are often provided with this capability.

9 Bézier Surfaces

Two sets of orthogonal Bézier curves can be used to design an object surface. The parametric vector function for the Bézier surface is formed as the tensor product of Bézier blending functions:

$$\mathbf{P}(u, v) = \sum_{j=0}^{m} \sum_{k=0}^{n} \mathbf{p}_{j,k} \, \text{BEZ}_{j,m}(v) \, \text{BEZ}_{k,n}(u) \tag{36}$$

with $\mathbf{p}_{j,k}$ specifying the location of the $(m + 1)$ by $(n + 1)$ control points.

Figure 26 illustrates two Bézier surface plots. The control points are connected by dashed lines, and the solid lines show curves of constant u and

https://hemanthrajhemu.github.io

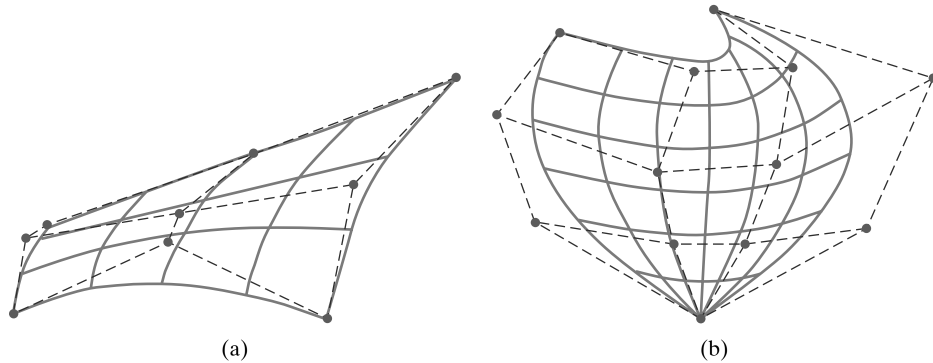


FIGURE 26
Wire-frame Bézier surfaces constructed with (a) 9 control points arranged in a 3×3 mesh and (b) 16 control points arranged in a 4×4 mesh. Dashed lines connect the control points.

constant v. Each curve of constant u is plotted by varying v over the interval from 0 to 1, with u fixed at one of the values in this unit interval. Curves of constant v are plotted similarly.

Bézier surfaces have the same properties as Bézier curves, and they provide a convenient method for interactive design applications. To specify the three-dimensional coordinate positions for the control points, we could first construct a rectangular grid in the xy "ground" plane. We then choose elevations above the ground plane at the grid intersections as the z-coordinate values for the control points.

Figure 27 illustrates a surface formed with two Bézier sections. As with curves, a smooth transition from one section to the other is assured by establishing both zero-order and first-order continuity at the boundary line. Zero-order continuity is obtained by matching control points at the boundary. First-order continuity is obtained by choosing control points along a straight line across the boundary and by maintaining a constant ratio of collinear line segments for each set of specified control points across section boundaries.

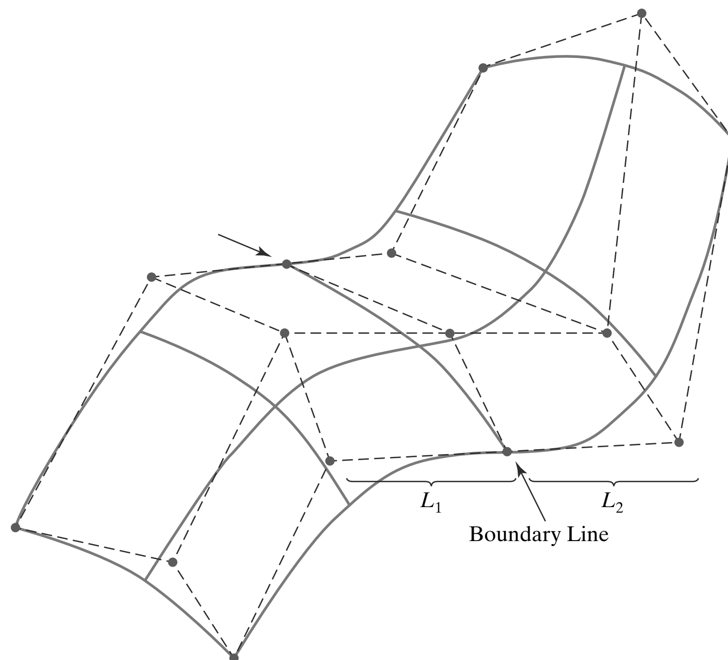


FIGURE 27
A composite Bézier surface constructed with two Bézier sections, joined at the indicated boundary line. The dashed lines connect the control points. First-order continuity is established by making the ratio of length L_1 to length L_2 constant for each collinear line of control points across the boundary between the surface sections.

https://hemanthrajhemu.github.io

10 B-Spline Curves

This spline category is the most widely used, and **B-spline** functions are commonly available in CAD systems and many graphics-programming packages. Like Bézier splines, B-splines are generated by approximating a set of control points. But **B-splines** have two advantages over Bézier splines: (1) the degree of a B-spline polynomial can be set independently of the number of control points (with certain limitations), and (2) B-splines allow local control over the shape of a spline. The tradeoff is that B-splines are more complex than Bézier splines.

B-Spline Curve Equations

We can write a general expression for the calculation of coordinate positions along a B-spline curve using a blending-function formulation as

$$\mathbf{P}(u) = \sum_{k=0}^{n} \mathbf{p}_k B_{k,d}(u), \qquad u_{\min} \le u \le u_{\max}, \quad 2 \le d \le n+1 \tag{37}$$

where $\mathbf{p}_k$ is an input set of $n + 1$ control points. There are several differences between this B-spline formulation and the expression for a Bézier spline curve. The range of parameter u now depends on how we choose the other B-spline parameters. And the B-spline blending functions $B_{k,d}$ are polynomials of degree $d - 1$, where d is the **degree parameter.** (Sometimes parameter d is alluded to as the "order" of the polynomial, but this can be misleading because the term order is also often used to mean simply the degree of the polynomial.) The degree parameter d can be assigned any integer value in the range from 2 up to the number of control points ($n + 1$). Actually, we could also set the value of the degree parameter at 1, but then our "curve" is just a point plot of the control points. Local control for B-splines is achieved by defining the blending functions over subintervals of the total range of u.

Blending functions for B-spline curves are defined by the Cox-deBoor recursion formulas:

$$\begin{aligned} B_{k,1}(u) &= \begin{cases} 1 & \text{if } u_k \le u \le u_{k+1} \\ 0 & \text{otherwise} \end{cases} \\ B_{k,d}(u) &= \frac{u - u_k}{u_{k+d-1} - u_k} B_{k,d-1}(u) + \frac{u_{k+d} - u}{u_{k+d} - u_{k+1}} B_{k+1,d-1}(u) \end{aligned} \tag{38}$$

where each blending function is defined over d subintervals of the total range of u. Each subinterval endpoint u_j is referred to as a **knot,** and the entire set of selected subinterval endpoints is called a **knot vector.** We can choose any values for the subinterval endpoints, subject to the condition $u_j \le u_{j+1}$. Values for $u_{\min}$ and $u_{\max}$ then depend on the number of control points we select, the value we choose for the degree parameter d, and how we set up the subintervals (knot vector). Because it is possible to choose the elements of the knot vector so that some denominators in the Cox-deBoor calculations evaluate to 0, this formulation assumes that any terms evaluated as 0/0 are to be assigned the value 0.

Figure 28 demonstrates the local-control characteristics of B-splines. In addition to local control, B-splines allow us to vary the number of control points used to design a curve without changing the degree of the polynomial. Also, we can increase the number of values in the knot vector to aid in curve design. When we do this, however, we must add control points because the size of the knot vector depends on parameter n.

https://hemanthrajhemu.github.io

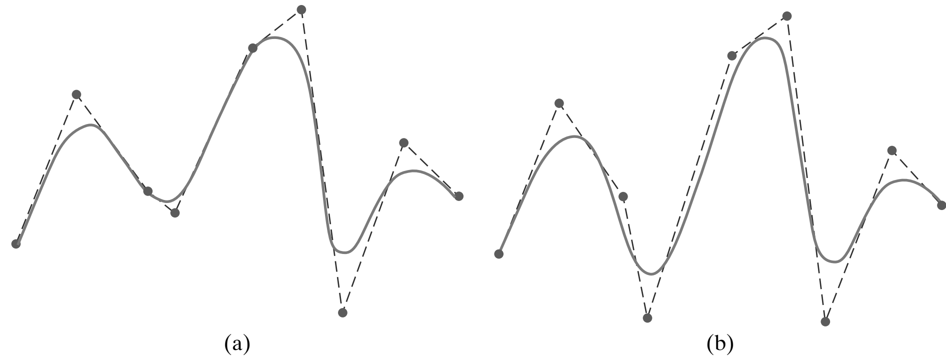


FIGURE 28
Local modification of a B-spline curve. Changing one of the control points in (a) produces curve (b), which is modified only in the neighborhood of the altered control point.

B-spline curves have the following properties:

- The polynomial curve has degree $d-1$ and C^{d-2} continuity over the range of u.
- For $n+1$ control points, the curve is described with $n+1$ blending functions.
- Each blending function $B_{k,d}$ is defined over d subintervals of the total range of u, starting at knot value u_k.
- The range of parameter u is divided into $n+d$ subintervals by the $n+d+1$ values specified in the knot vector.
- With knot values labeled as $\{u_0, u_1, \ldots, u_{n+d}\}$, the resulting B-spline curve is defined only in the interval from knot value u_{d-1} up to knot value u_{n+1}. (Some blending functions are undefined outside this interval.)
- Each section of the spline curve (between two successive knot values) is influenced by d control points.
- Any one control point can affect the shape of at most d curve sections.

In addition, a B-spline curve lies within the convex hull of at most $d+1$ control points, so that B-splines are tightly bound to the input positions. For any value of u in the interval from knot value u_{d-1} to u_{n+1}, the sum over all basis functions is 1, as follows:

$$\sum_{k=0}^{n} B_{k,d}(u) = 1 \tag{39}$$

Given the control-point positions and the value of the degree parameter d, we then need to specify the knot values to obtain the blending functions using the recurrence relations 38. There are three general classifications for knot vectors: uniform, open uniform, and nonuniform. B-splines are commonly described according to the selected knot vector class.

Uniform Periodic B-Spline Curves

When the spacing between knot values is constant, the resulting curve is called a **uniform** B-spline. For example, we can set up a uniform knot vector as

$$\{-1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0\}$$

Often knot values are normalized to the range between 0 and 1, as in

$$\{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$$

https://hemanthrajhemu.github.io

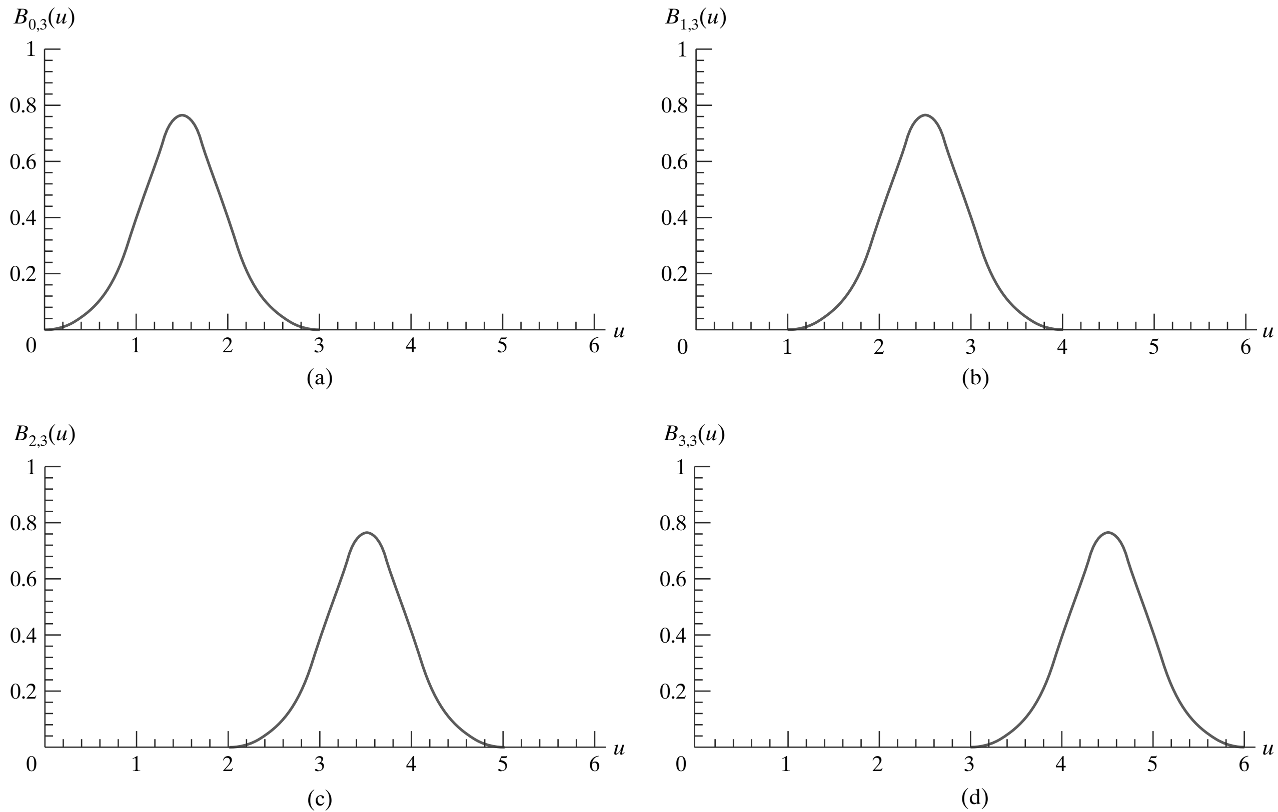


FIGURE 29
Periodic B-spline blending functions for $n = d = 3$ and a uniform, integer knot vector.

It is convenient in many applications to set up uniform knot values with a separation of 1 and a starting value of 0. The following knot vector is an example of this specification scheme:

$$\{0, 1, 2, 3, 4, 5, 6, 7\}$$

Uniform B-splines have **periodic** blending functions. That is, for given values of n and d, all blending functions have the same shape. Each successive blending function is simply a shifted version of the previous function:

$$B_{k,d}(u) = B_{k+1,d}(u + \Delta u) = B_{k+2,d}(u + 2\Delta u) \tag{40}$$

where Δu is the interval between adjacent knot values. Figure 29 shows the quadratic, uniform B-spline blending functions generated in the following example for a curve with four control points.

EXAMPLE 1 Uniform, Quadratic B-Splines

To illustrate the formulation of B-spline blending functions for a uniform, integer knot vector, we select parameter values $d = n = 3$. The knot vector must then contain $n + d + 1 = 7$ knot values:

$$\{0, 1, 2, 3, 4, 5, 6\}$$

and the range for parameter u is from 0 to 6, with $n + d = 6$ subintervals.

https://hemanthrajhemu.github.io

Each of the four blending functions spans $d = 3$ subintervals of the total range for u. Using the recurrence relations 38, we obtain the first blending function as

$$B_{0,3}(u) = \begin{cases} \frac{1}{2}u^2, & \text{for } 0 \le u < 1 \\ \frac{1}{2}u(2-u) + \frac{1}{2}(u-1)(3-u), & \text{for } 1 \le u < 2 \\ \frac{1}{2}(3-u)^2, & \text{for } 2 \le u < 3 \end{cases}$$

We obtain the next periodic blending function using Equation 40, substituting $u - 1$ for u in $B_{0,3}$, and shifting the starting positions up by 1:

$$B_{1,3}(u) = \begin{cases} \frac{1}{2}(u-1)^2, & \text{for } 1 \le u < 2 \\ \frac{1}{2}(u-1)(3-u) + \frac{1}{2}(u-2)(4-u), & \text{for } 2 \le u < 3 \\ \frac{1}{2}(4-u)^2, & \text{for } 3 \le u < 4 \end{cases}$$

Similarly, the remaining two periodic functions are obtained by successively shifting $B_{1,3}$ to the right:

$$B_{2,3}(u) = \begin{cases} \frac{1}{2}(u-2)^2, & \text{for } 2 \le u < 3 \\ \frac{1}{2}(u-2)(4-u) + \frac{1}{2}(u-3)(5-u), & \text{for } 3 \le u < 4 \\ \frac{1}{2}(5-u)^2, & \text{for } 4 \le u < 5 \end{cases}$$

$$B_{3,3}(u) = \begin{cases} \frac{1}{2}(u-3)^2, & \text{for } 3 \le u < 4 \\ \frac{1}{2}(u-3)(5-u) + \frac{1}{2}(u-4)(6-u), & \text{for } 4 \le u < 5 \\ \frac{1}{2}(6-u)^2, & \text{for } 5 \le u < 6 \end{cases}$$

A plot of the four periodic, quadratic blending functions is given in Figure 29, which demonstrates the local feature of B-splines. The first control point is multiplied by blending function $B_{0,3}(u)$. Therefore, changing the position of the first control point affects the shape of the curve only up to $u = 3$. Similarly, the last control point influences the shape of the spline curve in the interval where $B_{3,3}$ is defined.

Figure 29 also illustrates the limits of the B-spline curve for this example. All blending functions are present in the interval from $u_{d-1} = 2$ to $u_{n+1} = 4$. Below 2 and above 4, not all blending functions are present. This interval, from 2 to 4, is the range of the polynomial curve, and the interval in which Equation 39 is valid. Thus, the sum of all blending functions is 1 within this interval. Outside this interval, we cannot sum all blending functions, since they are not all defined below 2 and above 4.

https://hemanthrajhemu.github.io

Because the range of the resulting polynomial curve is from 2 to 4, we can determine the starting and ending position of the curve by evaluating theblending functions at these points to obtain

$$\mathbf{P}_{\text{start}} = \frac{1}{2}(\mathbf{p}_0 + \mathbf{p}_1), \qquad \mathbf{P}_{\text{end}} = \frac{1}{2}(\mathbf{p}_2 + \mathbf{p}_3)$$

Thus, the curve starts at the midposition between the first two control points and ends at the midposition between the last two control points.

We can also determine the parametric derivatives at the starting and ending positions of the curve. Taking the derivatives of the blending functions and substituting the endpoint values for parameter u, we find that

$$\mathbf{P}'_{\text{start}} = \mathbf{p}_1 - \mathbf{p}_0, \qquad \mathbf{P}'_{\text{end}} = \mathbf{p}_3 - \mathbf{p}_2$$

The parametric slope of the curve at the start position is parallel to the line joining the first two control points, and the parametric slope at the end of the curve is parallel to the line joining the last two control points.

An example plot of the quadratic periodic B-spline curve is given in Figure 30 for four control points selected in the xy plane.

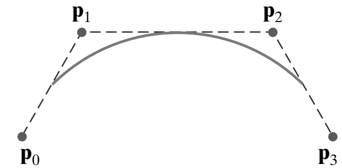


FIGURE 30
A quadratic, periodic B-spline fitted to four control points in the xy plane.

In the preceeding example, we noted that the quadratic curve starts between the first two control points and ends at a position between the last two control points. This result is valid for a quadratic periodic B-spline fitted to any number of distinct control points. In general, for higher-order polynomials, the start and end positions are each weighted averages of $d - 1$ control points. We can pull a spline curve closer to any control-point position by specifying that position multiple times.

General expressions for the boundary conditions for periodic B-splines can be obtained by reparameterizing the blending functions so that parameter u is mapped onto the unit interval from 0 to 1. Beginning and ending conditions are then obtained at $u = 0$ and $u = 1$.

Cubic Periodic B-Spline Curves

Because cubic periodic B-splines are commonly used in graphics packages, we consider the formulation for this class of splines. Periodic splines are particularly useful for generating certain closed curves. For example, the closed curve in Figure 31 can be generated in sections by cyclically specifying four of the six control points for each section. Also, if the coordinate positions for three consecutive control points are identical, the curve passes through that point.

For cubic B-spline curves, $d = 4$ and each blending function spans four subintervals of the total range of u. If we are to fit the cubic to four control points, then we could use the integer knot vector

$$\{0, 1, 2, 3, 4, 5, 6, 7\}$$

and recurrence relations 38 to obtain the periodic blending functions, as we did in the last section for quadratic periodic B-splines.

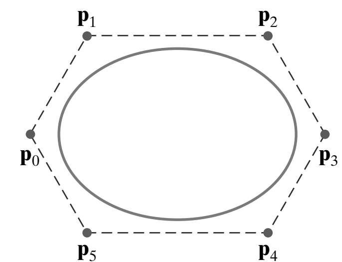


FIGURE 31
A closed, periodic, piecewise, cubic B-spline constructed using a cyclic specification of four control points for each curve section.

To derive the curve equations for a periodic, cubic B-spline, we consider an alternate formulation by starting with the boundary conditions and obtaining the blending functions normalized to the interval $0 \le u \le 1$. Using this formulation, we can also obtain the characteristic matrix easily. The boundary conditions for

https://hemanthrajhemu.github.io

periodic cubic B-splines with four control points, labeled $\mathbf{p}_0$, $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$, are

$$\begin{aligned} \mathbf{P}(0) &= \frac{1}{6}(\mathbf{p}_0 + 4\mathbf{p}_1 + \mathbf{p}_2) \\ \mathbf{P}(1) &= \frac{1}{6}(\mathbf{p}_1 + 4\mathbf{p}_2 + \mathbf{p}_3) \\ \mathbf{P}'(0) &= \frac{1}{2}(\mathbf{p}_2 - \mathbf{p}_0) \\ \mathbf{P}'(1) &= \frac{1}{2}(\mathbf{p}_3 - \mathbf{p}_1) \end{aligned} \tag{41}$$

These boundary conditions are similar to those for cardinal splines: Curve sections are defined with four control points, and parametric derivatives (slopes) at the beginning and end of each curve section are parallel to the chords joining adjacent control points. The B-spline curve section starts at a position near $\mathbf{p}_1$ and ends at a position near $\mathbf{p}_2$.

A matrix formulation for a cubic periodic B-spline with four control points can then be written as

$$\mathbf{P}(u) = [u^3 \quad u^2 \quad u \quad 1] \cdot \mathbf{M}_B \cdot \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{bmatrix} \tag{42}$$

where the B-spline matrix for periodic cubic polynomials is

$$\mathbf{M}_B = \frac{1}{6}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \tag{43}$$

This matrix can be obtained by solving for the coefficients in a general cubic polynomial expression, using the specified four boundary conditions.

We can also modify the B-spline equations to include a tension parameter t (as in cardinal splines). The matrix for the periodic cubic B-spline, with tension parameter t, is

$$\mathbf{M}_{B_t} = \frac{1}{6}\begin{bmatrix} -t & 12-9t & 9t-12 & t \\ 3t & 12t-18 & 18-15t & 0 \\ -3t & 0 & 3t & 0 \\ t & 6-2t & t & 0 \end{bmatrix} \tag{44}$$

which reduces to M_B when $t = 1$.

We obtain the periodic cubic B-spline blending functions over the parameter range from 0 to 1 by expanding the matrix representation into polynomial form. For example, using the tension value $t = 1$, we have

$$\begin{aligned} B_{0,3}(u) &= \frac{1}{6}(1-u)^3, \qquad 0 \le u \le 1 \\ B_{1,3}(u) &= \frac{1}{6}(3u^3 - 6u^2 + 4) \\ B_{2,3}(u) &= \frac{1}{6}(-3u^3 + 3u^2 + 3u + 1) \\ B_{3,3}(u) &= \frac{1}{6}u^3 \end{aligned} \tag{45}$$

https://hemanthrajhemu.github.io

Open Uniform B-Spline Curves

This class of B-splines is a cross between uniform B-splines and nonuniform B-splines. Sometimes it is treated as a special type of uniform B-spline, and sometimes it is considered to be in the nonuniform B-spline classification. For the **open uniform** B-splines, or simply **open** B-splines, the knot spacing is uniform except at the ends, where knot values are repeated d times.

Here are two examples of open, uniform, integer knot vectors, each with a starting value of 0:

$$\begin{aligned} &\{0, 0, 1, 2, 3, 3\} && \text{for } d = 2 \text{ and } n = 3 \\ &\{0, 0, 0, 0, 1, 2, 2, 2, 2\} && \text{for } d = 4 \text{ and } n = 4 \end{aligned} \tag{46}$$

We can normalize these knot vectors to the unit interval from 0 to 1 as

$$\begin{aligned} &\{0, 0, 0.33, 0.67, 1, 1\} && \text{for } d = 2 \text{ and } n = 3 \\ &\{0, 0, 0, 0, 0.5, 1, 1, 1, 1\} && \text{for } d = 4 \text{ and } n = 4 \end{aligned} \tag{47}$$

For any values of parameters d and n, we can generate an open uniform knot vector with integer values using the calculations

$$u_j = \begin{cases} 0 & \text{for } 0 \le j < d \\ j - d + 1 & \text{for } d \le j \le n \\ n - d + 2 & \text{for } j > n \end{cases} \tag{48}$$

for values of j ranging from 0 to $n + d$. With this assignment, the first d knots are assigned the value 0, and the last d knots have the value $n - d + 2$.

Open uniform B-splines have characteristics that are very similar to Bézier splines. In fact, when $d = n + 1$ (degree of the polynomial is n), open B-splines reduce to Bézier splines, and all knot values are either 0 or 1. For example, with a cubic open B-spline ($d = 4$) and four control points, the knot vector is

$$\{0, 0, 0, 0, 1, 1, 1, 1\}$$

The polynomial curve for an open B-spline connects the first and last control points. Also, the parametric slope of the curve at the first control point is parallel to the straight line formed by the first two control points, and the parametric slope at the last control point is parallel to the line defined by the last two control points. Thus, the geometric constraints for matching curve sections are the same as for Bézier curves.

As with Bézier curves, specifying multiple control points at the same coordinate position pulls any B-spline curve closer to that position. Since open B-splines start at the first control point and end at the last control point, a closed curve can be generated by setting the first and last control points at the same coordinate position.

EXAMPLE 2 Open Uniform, Quadratic B-Splines

From conditions 48 with $d = 3$ and $n = 4$ (five control points), we obtain the following eight values for the knot vector:

$$\{u_0, u_1, u_2, u_3, u_4, u_5, u_6, u_7\} = \{0, 0, 0, 1, 2, 3, 3, 3\}$$

The total range of u is divided into seven subintervals, and each of the five blending functions $B_{k,3}$ is defined over three subintervals, starting at knot position u_k. Thus $B_{0,3}$ is defined from $u_0 = 0$ to $u_3 = 1$, $B_{1,3}$ is defined

https://hemanthrajhemu.github.io

from $u_1 = 0$ to $u_4 = 2$, and $B_{4,3}$ is defined from $u_4 = 2$ to $u_7 = 3$. Explicit polynomial expressions are obtained for the blending functions from recurrence relations 38 as

$$B_{0,3}(u) = (1-u)^2 \qquad 0 \le u < 1$$

$$B_{1,3}(u) = \begin{cases} \frac{1}{2}u(4-3u) & 0 \le u < 1 \\ \frac{1}{2}(2-u)^2 & 1 \le u < 2 \end{cases}$$

$$B_{2,3}(u) = \begin{cases} \frac{1}{2}u^2 & 0 \le u < 1 \\ \frac{1}{2}u(2-u) + \frac{1}{2}(u-1)(3-u) & 1 \le u < 2 \\ \frac{1}{2}(3-u)^2 & 2 \le u < 3 \end{cases}$$

$$B_{3,3}(u) = \begin{cases} \frac{1}{2}(u-1)^2 & 1 \le u < 2 \\ \frac{1}{2}(3-u)(3u-5) & 2 \le u < 3 \end{cases}$$

$$B_{4,3}(u) = (u-2)^2 \qquad 2 \le u < 3$$

Figure 32 shows the shape of the these five blending functions. The local features of B-splines are again demonstrated. Blending function $B_{0,3}$ is nonzero only in the subinterval from 0 to 1, so the first control point influences the curve only in this interval. Similarly, function $B_{4,3}$ is 0 outside the interval from 2 to 3, and the position of the last control point does not affect the shape of the beginning and middle parts of the curve.

Matrix formulations for open B-splines are not generated as conveniently as they are for periodic uniform B-splines. This is due to the multiplicity of knot values at the beginning and end of the knot vector.

Nonuniform B-Spline Curves

For this class of splines, we can specify any values and intervals for the knot vector. With **nonuniform** B-splines, we can choose multiple internal knot values and unequal spacing between the knot values. Some examples are

$$\{0, 1, 2, 3, 3, 4\}$$
$$\{0, 2, 2, 3, 3, 6\}$$
$$\{0, 0, 0, 1, 1, 3, 3, 3\}$$
$$\{0, 0.2, 0.6, 0.9, 1.0\}$$

Nonuniform B-splines provide increased flexibility in controlling a curve shape. With unequally spaced intervals in the knot vector, we obtain different shapes for the blending functions in different intervals, which can be used in designing the spline features. By increasing knot multiplicity, we can produce subtle variations in the curve path and introduce discontinuities. Multiple knot values also reduce the continuity by 1 for each repeat of a particular value.

https://hemanthrajhemu.github.io

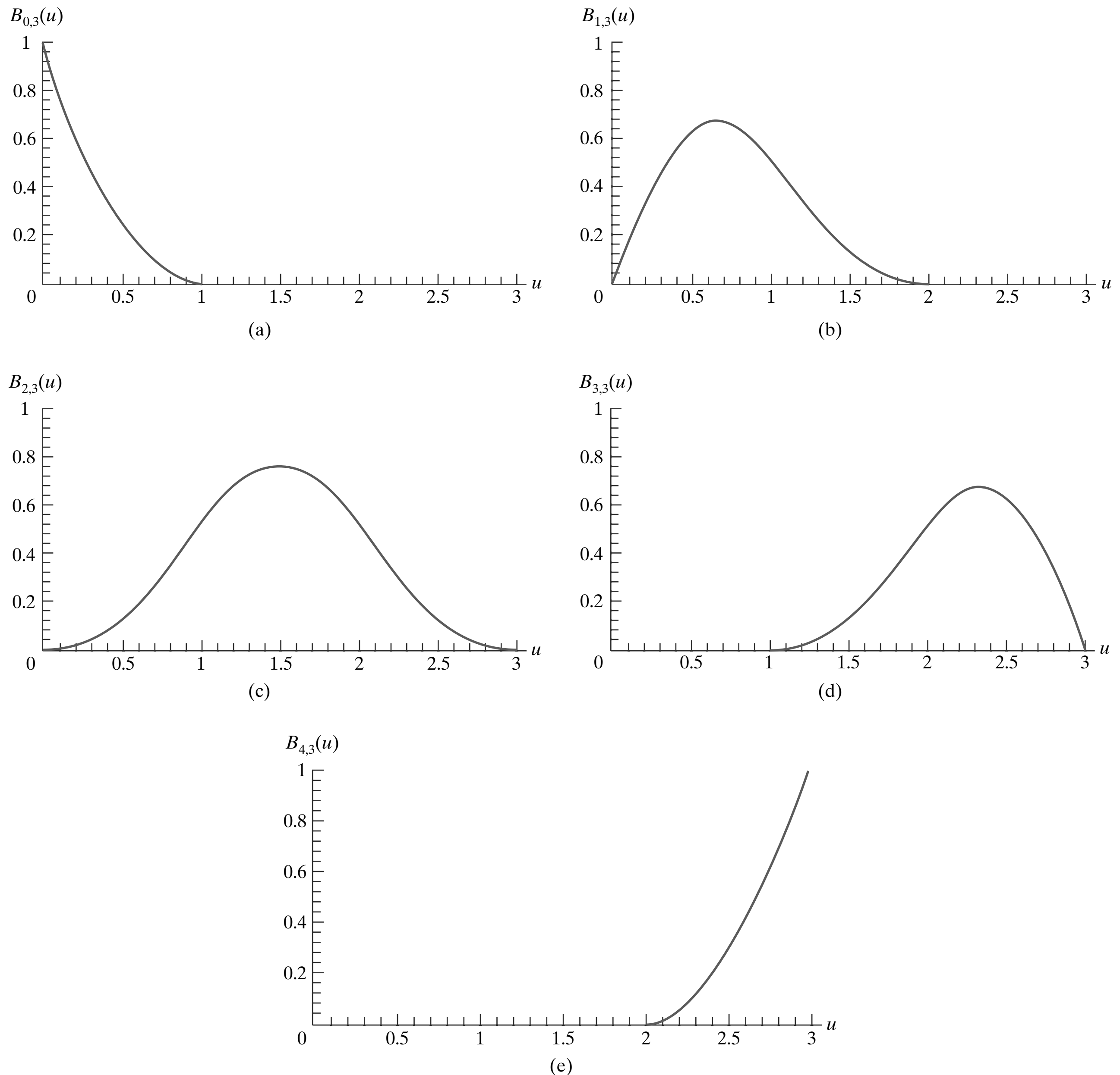


FIGURE 32
Open, uniform B-spline blending functions for $n = 4$ and $d = 3$.

We obtain the blending functions for a nonuniform B-spline using methods similar to those discussed for uniform and open B-splines. Given a set of $n + 1$ control points, we set the degree of the polynomial and select the knot values. Then, using the recurrence relations, we could either obtain the set of blending functions or evaluate curve positions directly for the display of the curve. Graphics packages often restrict the knot intervals to be either 0 or 1 to reduce computations. A set of characteristic matrices can then be stored and used to compute values along the spline curve without evaluating the recurrence relations for each curve point to be plotted.

https://hemanthrajhemu.github.io

11 B-Spline Surfaces

Formulation of a B-spline surface is similar to that for Bézier splines. We can obtain a vector point function over a B-spline surface using the tensor product of B-spline blending functions in the form

$$\mathbf{P}(u, v) = \sum_{k_u=0}^{n_u} \sum_{k_v=0}^{n_v} \mathbf{p}_{k_u,k_v} B_{k_u,d_u}(u) B_{k_v,d_v}(v) \tag{49}$$

where the vector values for $\mathbf{p}_{k_u,k_v}$ specify the positions of the $(n_u + 1)$ by $(n_v + 1)$ control points.

B-spline surfaces exhibit the same properties as those of their component B-spline curves. A surface can be constructed from selected values for degree parameters d_u and d_v, which set the degrees for the orthogonal surface polynomials at $d_u - 1$ and $d_v - 1$. For each surface parameter u and v, we also select values for the knot vectors, which determines the parameter range for the blending functions.

12 Beta-Splines

A generalization of B-splines are the **beta-splines,** also referred to as **β-splines,** that are formulated by imposing geometric continuity conditions on the first and second parametric derivatives. The continuity parameters for beta-splines are called *β parameters*.

Beta-Spline Continuity Conditions

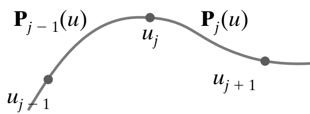


FIGURE 33
Position vectors along curve sections to the left and right of knot u_j.

For a specified knot vector, we designate the spline sections to the left and right of a particular knot u_j with the position vectors $\mathbf{P}_{j-1}(u)$ and $\mathbf{P}_j(u)$ (Figure 33). Zero-order continuity (*positional continuity*), G^0, at u_j is obtained by requiring that

$$\mathbf{P}_{j-1}(u_j) = \mathbf{P}_j(u_j) \tag{50}$$

First-order continuity (*unit tangent continuity*), G^1, is obtained by requiring tangent vectors to be proportional:

$$\beta_1 \mathbf{P}'_{j-1}(u_j) = \mathbf{P}'_j(u_j), \qquad \beta_1 > 0 \tag{51}$$

Here, parametric first derivatives are proportional, and the unit tangent vectors are continuous across the knot.

Second-order continuity (*curvature vector continuity*), G^2, is imposed with the condition

$$\beta_1^2 \mathbf{P}''_{j-1}(u_j) + \beta_2 \mathbf{P}'_{j-1}(u_j) = \mathbf{P}''_j(u_j) \tag{52}$$

where β_2 can be assigned any real number and $\beta_1 > 0$. The curvature vector provides a measure of the amount of bending for the curve at position u_j. When $\beta_1 = 1$ and $\beta_2 = 0$, beta-splines reduce to B-splines.

Parameter β_1 is called the *bias parameter* since it controls the skewness of the curve. For $\beta_1 > 1$, the curve tends to flatten to the right in the direction of the unit tangent vector at the knots. For $0 < \beta_1 < 1$, the curve tends to flatten to the left. The effect of β_1 on the shape of the spline curve is shown in Figure 34.

Parameter β_2 is called the *tension parameter* since it controls how tightly or loosely the spline fits the control graph. As β_2 increases, the curve approaches the shape of the control graph, as shown in Figure 35.

https://hemanthrajhemu.github.io

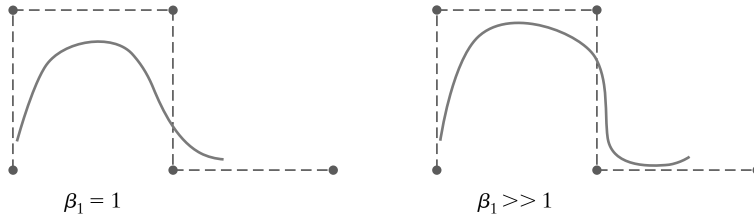


FIGURE 34 Effect of parameter β_1 on the shape of a beta-spline curve.

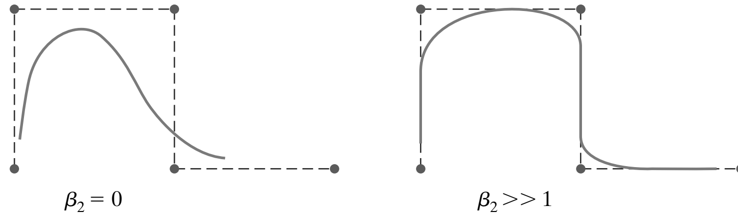


FIGURE 35 Effect of parameter β_2 on the shape of a beta-spline curve.

Cubic Periodic Beta-Spline Matrix Representation

Applying the beta-spline boundary conditions to a cubic polynomial with a uniform knot vector, we obtain the matrix representation for a periodic beta-spline as

$$\mathbf{M}_\beta = \frac{1}{\delta}\begin{bmatrix} -2\beta_1^3 & 2(\beta_2+\beta_1^3+\beta_1^2+\beta_1) & -2(\beta_2+\beta_1^2+\beta_1+1) & 2 \\ 6\beta_1^3 & -3(\beta_2+2\beta_1^3+2\beta_1^2) & 3(\beta_2+2\beta_1^2) & 0 \\ -6\beta_1^3 & 6(\beta_1^3-\beta_1) & 6\beta_1 & 0 \\ 2\beta_1^3 & \beta_2+4(\beta_1^2+\beta_1) & 2 & 0 \end{bmatrix} \tag{53}$$

where $\delta = \beta_2 + 2\beta_1^3 + 4\beta_1^2 + 4\beta_1 + 2$.

We obtain the B-spline matrix M_B when $\beta_1 = 1$ and $\beta_2 = 0$. And we have the B-spline tension matrix M_{B_t} (Eq. 44) when

$$\beta_1 = 1, \qquad \beta_2 = \frac{12}{t}(1-t)$$

13 Rational Splines

A rational function is simply the ratio of two polynomials. Thus, a **rational spline** is the ratio of two spline functions. For example, a rational B-spline curve can be described with the position vector

$$\mathbf{P}(u) = \frac{\sum_{k=0}^{n} \omega_k \mathbf{p}_k B_{k,d}(u)}{\sum_{k=0}^{n} \omega_k B_{k,d}(u)} \tag{54}$$

where the $\mathbf{p}_k$ are the $n+1$ control-point positions. Parameters ω_k are weight factors for the control points. The greater the value of a particular ω_k, the closer the curve is pulled toward the control point $\mathbf{p}_k$ weighted by that parameter. When all weight factors are set to the value 1, we have the standard B-spline curve, because the denominator in Equation 54 is then just the sum of the blending functions, which has the value 1 (Equation 39).

Rational splines have two important advantages, compared to nonrational splines. First, they provide an exact representation for quadric curves (conics), such as circles and ellipses. Nonrational splines, which are polynomials, can only approximate conics. This allows graphics packages to model all curve shapes with one representation—rational splines—without needing a library of curve

https://hemanthrajhemu.github.io

functions to handle different design shapes. The second advantage of rational splines is that they are invariant with respect to a perspective viewing transformation. This means that we can apply a perspective viewing transformation to the control points of the rational curve, and we will obtain the correct view of the curve. Nonrational splines, on the other hand, are not invariant with respect to a perspective viewing transformation. Typically, graphics design packages use nonuniform knot vector representations for constructing rational B-splines. These splines are referred to as *nonuniform rational B-splines* (NURBs).

Homogeneous coordinate representations are used for rational splines because the denominator can be treated as the homogeneous factor h in a four-dimensional representation of the control points. Thus, a rational spline can be thought of as the projection of a four-dimensional nonrational spline into three-dimensional space.

In general, constructing a rational B-spline representation is carried out using the same procedures that we employed to obtain a nonrational representation. Given the set of control points, the degree of the polynomial, the weighting factors, and the knot vector, we apply the recurrence relations to obtain the blending functions. With some CAD systems, we construct a conic section by specifying three points on an arc. A rational homogeneous-coordinate spline representation is then determined by computing control-point positions that would generate the selected conic type.

As an example of describing conic sections with rational splines, we can use a quadratic B-spline function ($d = 3$), three control points, and the open knot vector

$$\{0, 0, 0, 1, 1, 1\}$$

which is the same as a quadratic Bézier spline. We then set the weighting functions to the values

$$\begin{aligned} \omega_0 &= \omega_2 = 1 \\ \omega_1 &= \frac{r}{1-r}, \qquad 0 \le r < 1 \end{aligned} \tag{55}$$

and the rational B-spline representation is

$$\mathbf{P}(u) = \frac{\mathbf{p}_0 B_{0,3}(u) + [r/(1-r)]\mathbf{p}_1 B_{1,3}(u) + \mathbf{p}_2 B_{2,3}(u)}{B_{0,3}(u) + [r/(1-r)]B_{1,3}(u) + B_{2,3}(u)} \tag{56}$$

We then obtain the various conics (Figure 36) with the following values for parameter r:

$r > 1/2$,	$\omega_1 > 1$	Hyperbola section
$r = 1/2$,	$\omega_1 = 1$	Parabola section
$r < 1/2$,	$\omega_1 < 1$	Ellipse section
$r = 0$,	$\omega_1 = 0$	Straight-line segment

We can generate a one-quarter arc of a unit circle in the first quadrant of the xy plane (Figure 37) by setting $\omega_1 = \cos\phi$ and by choosing the control points as

$$\mathbf{p}_0 = (0, 1), \qquad \mathbf{p}_1 = (1, 1), \qquad \mathbf{p}_2 = (1, 0)$$

A complete circle can be obtained by generating sections in the other three quadrants using similar control-point placements. Or we could produce a complete circle from the first-quadrant section using geometric transformations in the xy plane. For example, we can reflect the one-quarter circular arc about the x and y axes to produce the circular arcs in the other three quadrants.

https://hemanthrajhemu.github.io

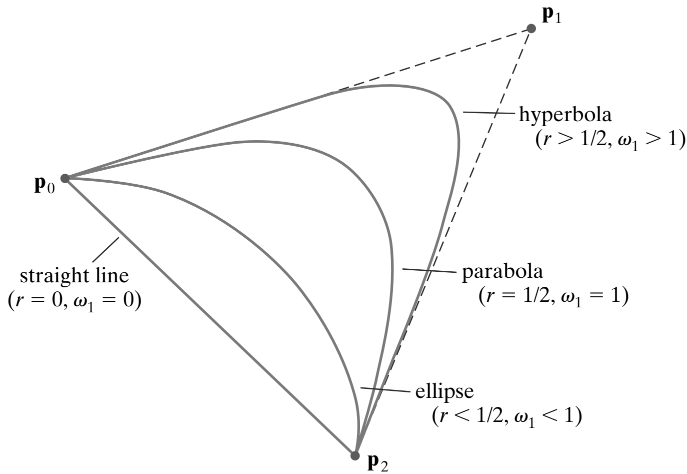


FIGURE 36
Conic sections generated using various values for the rational-spline weighting factor ω_1.

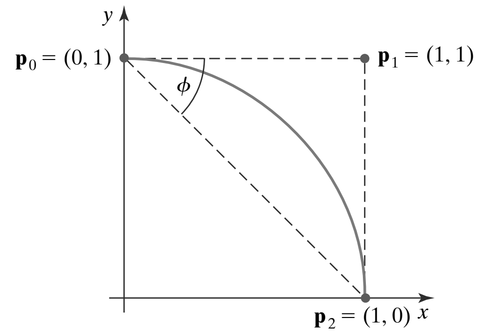


FIGURE 37
A circular arc in the first quadrant of the xy plane.

A homogeneous representation for a unit circular arc in the first quadrant of the xy plane is

$$\begin{bmatrix} x_h(u) \\ y_h(u) \\ z_h(u) \\ h(u) \end{bmatrix} = \begin{bmatrix} 1-u^2 \\ 2u \\ 0 \\ 1+u^2 \end{bmatrix} \tag{57}$$

This homogeneous representation yields the parametric circle equations for the first quadrant as

$$\begin{aligned} x &= \frac{x_h(u)}{h(u)} = \frac{1-u^2}{1+u^2} \\ y &= \frac{y_h(u)}{h(u)} = \frac{2u}{1+u^2} \end{aligned} \tag{58}$$

14 Conversion Between Spline Representations

Sometimes it is desirable to be able to switch from one spline representation to another. For instance, a Bézier representation is most convenient for subdividing a spline curve, while a B-spline representation offers greater design flexibility. Therefore, we might design a curve using B-spline sections, then convert to an equivalent Bézier representation to display the object using a recursive subdivision procedure to locate coordinate positions along the curve.

Suppose that we have a spline description of an object that can be expressed with the following matrix product:

$$\mathbf{P}(u) = \mathbf{U} \cdot \mathbf{M}_{\text{spline1}} \cdot \mathbf{M}_{\text{geom1}} \tag{59}$$

where $\mathbf{M}_{\text{spline1}}$ is the matrix characterizing the spline representation and $\mathbf{M}_{\text{geom1}}$ is the column matrix of geometric constraints (for example, control-point coordinates). To transform to a second representation with spline matrix $\mathbf{M}_{\text{spline2}}$, we must determine the geometric constraint matrix $\mathbf{M}_{\text{geom2}}$ that produces the same

https://hemanthrajhemu.github.io

vector point function for the object. That is,

$$\mathbf{P}(u) = \mathbf{U} \cdot \mathbf{M}_{\text{spline2}} \cdot \mathbf{M}_{\text{geom2}} \tag{60}$$

or

$$\mathbf{U} \cdot \mathbf{M}_{\text{spline2}} \cdot \mathbf{M}_{\text{geom2}} = \mathbf{U} \cdot \mathbf{M}_{\text{spline1}} \cdot \mathbf{M}_{\text{geom1}} \tag{61}$$

Solving for $\mathbf{M}_{\text{geom2}}$, we have

$$\begin{aligned} \mathbf{M}_{\text{geom2}} &= \mathbf{M}^{-1}_{\text{spline2}} \cdot \mathbf{M}_{\text{spline1}} \cdot \mathbf{M}_{\text{geom1}} \\ &= \mathbf{M}_{s1,s2} \cdot \mathbf{M}_{\text{geom1}} \end{aligned} \tag{62}$$

Thus, the required transformation matrix that converts from the first spline representation to the second is

$$\mathbf{M}_{s1,s2} = \mathbf{M}^{-1}_{\text{spline2}} \cdot \mathbf{M}_{\text{spline1}} \tag{63}$$

A nonuniform B-spline cannot be characterized with a general spline matrix. But we can rearrange the knot sequence to change the nonuniform B-spline to a Bézier representation. Then the Bézier matrix could be converted to any other form.

The following example calculates the transformation matrix for conversion from a periodic, cubic B-spline representation to a cubic Bézier spline representation:

$$\begin{aligned} \mathbf{M}_{B,\text{Bez}} &= \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}^{-1} \cdot \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 4 & 1 & 0 \\ 0 & 4 & 2 & 0 \\ 0 & 2 & 4 & 0 \\ 0 & 1 & 4 & 1 \end{bmatrix} \end{aligned} \tag{64}$$

The transformation matrix for converting from a cubic Bézier representation to a periodic, cubic B-spline representation is

$$\begin{aligned} \mathbf{M}_{\text{Bez},B} &= \begin{bmatrix} -\frac{1}{6} & \frac{1}{2} & -\frac{1}{2} & \frac{1}{6} \\ \frac{1}{2} & -1 & \frac{1}{2} & 0 \\ -\frac{1}{2} & 0 & \frac{1}{2} & 0 \\ \frac{1}{6} & \frac{2}{3} & \frac{1}{6} & 0 \end{bmatrix}^{-1} \cdot \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 6 & -7 & 2 & 0 \\ 0 & 2 & -1 & 0 \\ 0 & -1 & 2 & 0 \\ 0 & 2 & -7 & 6 \end{bmatrix} \end{aligned} \tag{65}$$

15 Displaying Spline Curves and Surfaces

To display a spline curve or surface, we must determine coordinate positions on the curve or surface that project to pixel positions on the display device. This means that we must evaluate the parametric polynomial spline functions in certain increments over the range of the functions, and several methods have been developed for accomplishing this evaluation efficiently.

https://hemanthrajhemu.github.io

Horner's Rule

The simplest method for evaluating a polynomial, other than direct calculation of each term in succession, is *Horner's rule,* which performs the calculations by successive factoring. This requires one multiplication and one addition at each step. For a polynomial of degree n, there are n steps.

For example, suppose that we have a cubic-spline representation where the x coordinate is expressed as

$$x(u) = a_x u^3 + b_x u^2 + c_x u + d_x \tag{66}$$

with similar expressions for the y and z coordinates. For a particular value of parameter u, we evaluate this polynomial in the following factored order:

$$x(u) = [(a_x\, u + b_x)u + c_x]u + d_x \tag{67}$$

The calculation of each x value requires three multiplications and three additions, so that the determination of each coordinate position (x, y, z) along a cubic-spline curve requires nine multiplications and nine additions.

Additional factoring manipulations could be applied to reduce the number of computations required by Horner's method, especially for higher-order polynomials (degree greater than 3). But repeated determination of coordinate positions over the range of a spline function can be computed much faster using forward-difference calculations or spline-subdivision methods.

Forward-Difference Calculations

A fast method for evaluating polynomial functions is to generate successive values recursively by incrementing previously calculated values as, for example,

$$x_{k+1} = x_k + \Delta x_k \tag{68}$$

Thus, once we know the increment and the value of x_k at any step, we get the next value simply by adding the increment to x_k. The increment Δx_k at each step is called the *forward difference*. For the parametric curve representation, we obtain the forward differences from the intervals we select for parameter u. If we divide the total range of u into subintervals of fixed size δ, then two successive x positions occur at $x_k = x(u_k)$ and $x_{k+1} = x(u_{k+1})$, where

$$u_{k+1} = u_k + \delta, \qquad k = 0, 1, 2, \ldots \tag{69}$$

and $u_0 = 0$.

As an illustration of this method, we first consider the polynomial representation $x(u) = a_x\, u + b_x$ for the x-coordinate position along a linear-spline curve. Two successive x-coordinate positions are represented as

$$\begin{aligned} x_k &= a_x\, u_k + b_x \\ x_{k+1} &= a_x(u_k + \delta) + b_x \end{aligned} \tag{70}$$

Subtracting the two equations, we obtain the forward difference:

$$\Delta x_k = x_{k+1} - x_k = a_x \delta \tag{71}$$

In this case, the forward difference is a constant. With higher-order polynomials, the forward difference is itself a polynomial function of parameter u. This forward-difference polynomial has degree one less than the original polynomial.

For the cubic-spline representation in Equation 66, two successive x-coordinate positions have the polynomial representations

$$\begin{aligned} x_k &= a_x\, u_k^3 + b_x\, u_k^2 + c_x\, u_k + d_x \\ x_{k+1} &= a_x(u_k + \delta)^3 + b_x(u_k + \delta)^2 + c_x(u_k + \delta) + d_x \end{aligned} \tag{72}$$

The forward difference now evaluates to

$$\Delta x_k = 3a_x \delta\, u_k^2 + (3a_x\, \delta^2 + 2b_x\, \delta)u_k + (a_x\, \delta^3 + b_x\, \delta^2 + c_x\, \delta) \tag{73}$$

https://hemanthrajhemu.github.io

which is a quadratic function of parameter u_k. Because Δx_k is a polynomial function of u, we can use the same incremental procedure to obtain successive values of Δx_k. That is,

$$\Delta x_{k+1} = \Delta x_k + \Delta_2 x_k \tag{74}$$

where the second forward difference is the linear function

$$\Delta_2 x_k = 6a_x \delta^2 u_k + 6a_x \delta^3 + 2b_x \delta^2 \tag{75}$$

Repeating this process once more, we can write

$$\Delta_2 x_{k+1} = \Delta_2 x_k + \Delta_3 x_k \tag{76}$$

with the third forward difference as the constant expression

$$\Delta_3 x_k = 6a_x \delta^3 \tag{77}$$

Equations 68, 74, 76, and 77 provide an incremental forward-difference calculation of points along the cubic curve. Starting at $u_0 = 0$ with a constant step size δ, the initial values for the x coordinate and its first two forward differences are

$$\begin{aligned} x_0 &= d_x \\ \Delta x_0 &= a_x \delta^3 + b_x \delta^2 + c_x \delta \\ \Delta_2 x_0 &= 6a_x \delta^3 + 2b_x \delta^2 \end{aligned} \tag{78}$$

Once these initial values have been computed, the calculation for each successive x-coordinate position requires only three additions.

We can apply forward-difference methods to determine positions along spline curves of any degree n. Each successive coordinate position (x, y, z) is evaluated with a series of $3n$ additions. For surfaces, the incremental calculations are applied to both parameter u and parameter v.

Subdivision Methods

Recursive spline-subdivision procedures are used to repeatedly divide a given curve section in half, increasing the number of control points at each step. Subdivision methods are useful for displaying approximation spline curves since we can continue the subdivision process until the control graph approximates the curve path. Control-point coordinates can then be plotted as curve positions. Another application of subdivision is to generate more control points for shaping a curve. Thus, we could design a general curve shape with a few control points, then apply a subdivision procedure to obtain additional control points. With the added control points, we can then make fine adjustments to small sections of the curve.

Spline subdivision is applied to a Bézier curve section most easily because the curve begins at the first control point and ends at the last control point, the range of parameter u is always between 0 and 1, and it is easy to determine when the control points are "near enough" to the curve path. Bézier subdivision can be applied to other spline representations with the following sequence of actions:

1. Convert the current spline representation to a Bézier representation.
2. Apply the Bézier subdivision algorithm.
3. Convert the Bézier representation to the original spline representation.

Figure 38 shows the first step in a recursive subdivision of a cubic Bézier curve section. Positions along the Bézier curve are described with the parametric point function $\mathbf{P}(u)$ for $0 \le u \le 1$. At the first subdivision step, we use the halfway point $\mathbf{P}(0.5)$ to divide the original curve into two segments. The first segment is then described with the point function $\mathbf{P}_1(s)$, and the second segment is described

https://hemanthrajhemu.github.io

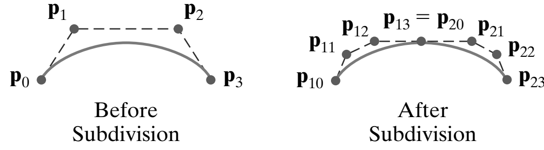


FIGURE 38
Subdividing a cubic Bézier curve section into two segments, each with four control points.

with $\mathbf{P}_2(t)$, where

$$s = 2u, \qquad \text{for } 0.0 \leq u \leq 0.5$$
$$t = 2u - 1, \qquad \text{for } 0.5 \leq u \leq 1.0 \tag{79}$$

Each of the two curve segments has the same number of control points as the original curve. Also, the boundary conditions (position and parametric slope) at the ends of each of the two curve segments must match the position and slope values for the original curve function $\mathbf{P}(u)$. This gives us four conditions for each curve segment that we can use to determine the control-point positions. For the first segment, the four control points are

$$\begin{aligned}
\mathbf{p}_{1,0} &= \mathbf{p}_0 \\
\mathbf{p}_{1,1} &= \frac{1}{2}(\mathbf{p}_0 + \mathbf{p}_1) \\
\mathbf{p}_{1,2} &= \frac{1}{4}(\mathbf{p}_0 + 2\mathbf{p}_1 + \mathbf{p}_2) \\
\mathbf{p}_{1,3} &= \frac{1}{8}(\mathbf{p}_0 + 3\mathbf{p}_1 + 3\mathbf{p}_2 + \mathbf{p}_3)
\end{aligned} \tag{80}$$

For the second segment, we obtain the four control points

$$\begin{aligned}
\mathbf{p}_{2,0} &= \frac{1}{8}(\mathbf{p}_0 + 3\mathbf{p}_1 + 3\mathbf{p}_2 + \mathbf{p}_3) \\
\mathbf{p}_{2,1} &= \frac{1}{4}(\mathbf{p}_1 + 2\mathbf{p}_2 + \mathbf{p}_3) \\
\mathbf{p}_{2,2} &= \frac{1}{2}(\mathbf{p}_2 + \mathbf{p}_3) \\
\mathbf{p}_{2,3} &= \mathbf{p}_3
\end{aligned} \tag{81}$$

An efficient order for computing the new set of control points can be set up using only add and shift (division by 2) operations as

$$\begin{aligned}
\mathbf{p}_{1,0} &= \mathbf{p}_0 \\
\mathbf{p}_{1,1} &= \frac{1}{2}(\mathbf{p}_0 + \mathbf{p}_1) \\
\mathbf{T} &= \frac{1}{2}(\mathbf{p}_1 + \mathbf{p}_2) \\
\mathbf{p}_{1,2} &= \frac{1}{2}(\mathbf{p}_{1,1} + \mathbf{T}) \\
\mathbf{p}_{2,3} &= \mathbf{p}_3 \\
\mathbf{p}_{2,2} &= \frac{1}{2}(\mathbf{p}_2 + \mathbf{p}_3) \\
\mathbf{p}_{2,1} &= \frac{1}{2}(\mathbf{T} + \mathbf{p}_{2,2}) \\
\mathbf{p}_{2,0} &= \frac{1}{2}(\mathbf{p}_{1,2} + \mathbf{p}_{2,1}) \\
\mathbf{p}_{1,3} &= \mathbf{p}_{2,0}
\end{aligned} \tag{82}$$

https://hemanthrajhemu.github.io

The preceding steps can be repeated any number of times, depending on whether we are subdividing the curve to gain more control points or trying to locate approximate curve positions. When we are subdividing to obtain a set of display points, we can terminate the subdivision procedure when the curve segments are small enough. One way to determine this is to check the distance from the first control point to the last control point for each segment. If this distance is "sufficiently" small, we can stop subdividing. Another test is to check the distances between adjacent pairs of control points. Alternatively, we could stop subdividing when the set of control points for each segment is nearly along a straight-line path.

Subdivision methods can be applied to Bézier curves of any degree. For a Bézier polynomial of degree $n-1$, the $2n$ control points for each of the initial two curve segments are

$$\begin{aligned} \mathbf{p}_{1,k} &= \frac{1}{2^k}\sum_{j=0}^{k} C(k, j)\mathbf{p}_j, \qquad k = 0, 1, 2, \ldots, n \\ \mathbf{p}_{2,k} &= \frac{1}{2^{n-k}}\sum_{j=k}^{n} C(n-k, n-j)\mathbf{p}_j \end{aligned} \tag{83}$$

where $C(k, j)$ and $C(n-k, n-j)$ are the binomial coefficients.

Subdivision methods can be applied directly to nonuniform B-splines by adding values to the knot vector. In general, however, these methods are not as efficient as Bézier subdivision.

16 OpenGL Approximation-Spline Functions

Both Bézier splines and B-splines can be displayed using OpenGL functions, as well as trimming curves for spline surfaces. The core library contains the Bézier functions, and the OpenGL Utility (GLU) has the B-spline and trimming-curve functions. Bézier functions are often hardware implemented, and the GLU functions provide a B-spline interface that accesses OpenGL point-plotting and line-drawing routines.

OpenGL Bézier-Spline Curve Functions

We specify parameters and activate the routines for Bézier-curve display with the OpenGL functions

```
glMap1* (GL_MAP1_VERTEX_3, uMin, uMax, stride, nPts, *ctrlPts);
glEnable (GL_MAP1_VERTEX_3);
```

We deactivate the routines with

```
glDisable (GL_MAP1_VERTEX_3);
```

A suffix code of `f` or `d` is used with `glMap1` to indicate either floating-point or double precision for the data values. Minimum and maximum values for the curve parameter u are specified in `uMin` and `uMax`, although these values for a Bézier curve are typically set to 0 and 1.0, respectively. The three-dimensional, floating-point, Cartesian-coordinate values for the Bézier control points are listed in array `ctrlPts`, and the number of elements in this array is given as a positive integer using parameter `nPts`. Parameter `stride` is assigned an integer offset that indicates the number of data values between the beginning of one coordinate position in array `ctrlPts` and the beginning of the next coordinate position. For a list of three-dimensional control-point positions, we set `stride` = 3. A

https://hemanthrajhemu.github.io

higher value for `stride` would be used if we specified the control points using four-dimensional homogeneous coordinates or intertwined the coordinate values with other data, such as color values. To express control-point positions in four-dimensional homogeneous coordinates (x, y, z, h), we need only change the value of `stride` and change the symbolic constant in `glMap1` and in `glEnable` to `GL_MAP1_VERTEX_4`.

After we have set up the Bézier parameters and activated the curve-generation routines, we need to evaluate positions along the spline path and display the resulting curve. A coordinate position along the curve path is calculated with

```
glEvalCoord1* (uValue);
```

where parameter `uValue` is assigned some value in the interval from `uMin` to `uMax`. The suffix code for this function can be either `f` or `d`, and we can also use the suffix code `v` to indicate that the value for the argument is given in an array. Function `glEvalCoord1` calculates a coordinate position using Equation 22 with the parameter value

$$u = \frac{u_{\text{value}} - u_{\text{min}}}{u_{\text{max}} - u_{\text{min}}} \tag{84}$$

which maps the `uValue` to the interval from 0 to 1.0.

When `glEvalCoord1` processes a value for the curve parameter u, it generates a `glVertex3` function. To obtain a Bézier curve, we thus repeatedly invoke the `glEvalCoord1` function to produce a set of points along the curve path, using selected values in the range from `uMin` to `uMax`. Joining these points with straight-line segments, we can approximate the spline curve as a polyline.

As an example of the OpenGL Bézier-curve routines, the following code uses the four control-point positions from the program in Section 8 to generate a two-dimensional cubic Bézier curve. In this instance, 50 points are plotted along the curve path, and the curve points are connected with straight-line segments. The curve path is then displayed as a blue polyline, and the control points are plotted as red points of size 5 (Figure 39).

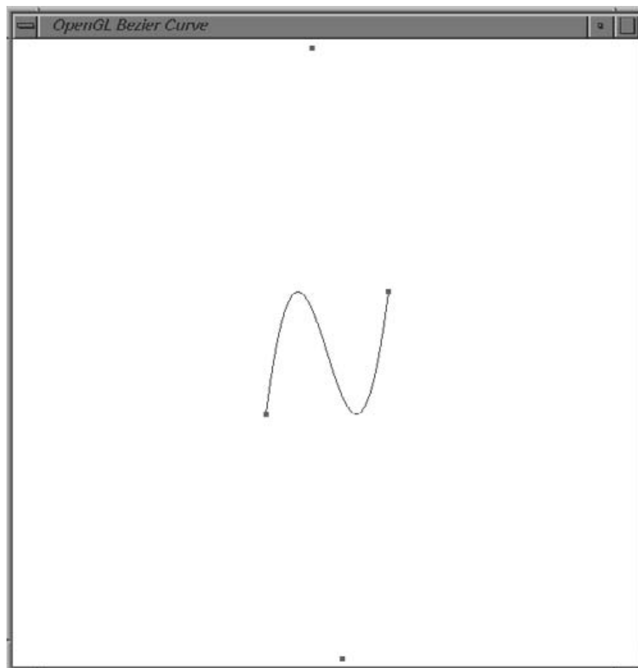


FIGURE 39
A set of four control points and the associated Bézier curve, displayed with OpenGL routines as an approximating polyline.

https://hemanthrajhemu.github.io

```
GLfloat ctrlPts [4][3] = { {-40.0, 40.0, 0.0}, {-10.0, 200.0, 0.0},
                           {10.0, -200.0, 0.0}, {40.0, 40.0, 0.0} };

glMap1f (GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, *ctrlPts);
glEnable (GL_MAP1_VERTEX_3);

GLint k;

glColor3f (0.0, 0.0, 1.0);            //  Set line color to blue.
glBegin (GL_LINE_STRIP);              //  Generate Bezier "curve".
    for (k = 0; k <= 50; k++)
        glEvalCoord1f (GLfloat (k) / 50.0);
glEnd ( );


glColor (1.0, 0.0, 0.0);              //  Set point color to red.
glPointSize (5.0);                    //  Set point size to 5.0.
glBegin (GL_POINTS);                  //  Plot control points.
    for (k = 0; k < 4; k++);
            glVertex3fv (&ctrlPts [k][0]);
glEnd ( );
```

Although the previous example generated a spline curve with evenly spaced parameter values, we can use the `glEvalCoord1f` function to obtain any spacing for parameter u. Usually, however, a spline curve is generated with evenly spaced parameter values, and OpenGL provides the following functions, which we can use to produce a set of uniformly spaced parameter values:

```
glMapGrid1* (n, u1, u2);
glEvalMesh1 (mode, n1, n2);
```

The suffix code for `glMapGrid1` can be either `f` or `d`. Parameter `n` specifies the integer number of equal subdivisions over the range from `u1` to `u2`, and parameters `n1` and `n2` specify an integer range corresponding to `u1` and `u2`. Parameter `mode` is assigned either `GL_POINT` or `GL_LINE`, depending on whether we want to display the curve using discrete points (a dotted curve) or using straight-line segments. For a curve that is to be displayed as a polyline, the output of these two functions is the same as the output from the following code, except that the argument of `glEvalCoord1` is set either to `u1` or to `u2` if $k = 0$ or $k = n$, respectively, to avoid round-off error. In other words, with `mode = GL_LINE`, the preceding OpenGL commands are equivalent to

```
glBegin (GL_LINE_STRIP);
    for (k = n1; k <= n2; k++)
        glEvalCoord1f (u1 + k * (u2 - u1) / n);
glEnd ( );
```

Thus, in the previous programming example, we could replace the block of code containing the loop for generating the Bézier curve with the following statements.

```
glColor3f (0.0, 0.0, 1.0);
glMapGrid1f (50, 0.0, 1.0);
glEvalMesh1 (GL_LINE, 0, 50);
```


https://hemanthrajhemu.github.io

Using the `glMapGrid1` and `glEvalMesh1` functions, we can divide a curve into a number of segments and select the parameter spacing for each segment according to its curvature. Therefore, a segment with more oscillations could be assigned more intervals, and a flatter section of the curve could be assigned fewer intervals.

Instead of displaying Bézier curves, we can use the `glMap1` function to designate values for other kinds of data, and seven other OpenGL symbolic constants are available for this purpose. With the symbolic constant `GL_MAP1_COLOR_4`, we use the array `ctrlPts` to specify a list of four-element (red, green, blue, alpha) colors. Then a linearly interpolated set of colors can be generated for use in an application, and these generated color values do not change the current setting for the color state. Similarly, we can designate a list of values from a color-index table with `GL_MAP1_INDEX`, and a list of three-dimensional, surface-normal vectors is specified in array `ctrlPts` when we use the symbolic constant `GL_MAP1_NORMAL`. The remaining four symbolic constants are used with lists of surface-texture information.

Multiple `glMap1` functions can be activated simultaneously, and calls to `glEvalCoord1` or to `glMapGrid1` and `glEvalMesh1` then produce data points for each data type that is enabled. This allows us to generate combinations of coordinate positions, color values, surface-normal vectors, and surface-texture data. Note, however, we cannot activate `GL_MAP1_VERTEX_3` and `GL_MAP1_VERTEX_4` simultaneously, and we can activate only one of the surface-texture generators at any one time.

OpenGL Bézier-Spline Surface Functions

Activation and parameter specification for the OpenGL Bézier-surface routines are accomplished with

```
glMap2* (GL_MAP2_VERTEX_3, uMin, uMax, uStride, nuPts,
                    vMin, vMax, vStride, nvPts, *ctrlPts);
glEnable (GL_MAP2_VERTEX_3);
```

A suffix code of `f` or `d` is used with `glMap2` to indicate either floating-point or double precision for the data values. For a surface, we specify minimum and maximum values for both parameter u and parameter v. The three-dimensional Cartesian coordinates for the Bézier control points are listed in the double-subscripted array `ctrlPts`, and the integer size of the array is given with parameters `nuPts` and `nvPts`. If control points are to be specified using four-dimensional homogeneous coordinates, we use the symbolic constant `GL_MAP2_VERTEX_4` instead of `GL_MAP2_VERTEX_3`. The integer offset between the beginning of coordinate values for control point $\mathbf{p}_{j,k}$ and the beginning of coordinate values for $\mathbf{p}_{j+1,k}$ is given in `uStride`; and the integer offset between the beginning of coordinate values for control point $\mathbf{p}_{j,k}$ and the beginning of coordinate values for $\mathbf{p}_{j,k+1}$ is given in `vStride`. This allows the coordinate data to be intertwined with other data, so that we need to specify only the offsets to locate coordinate values. We deactivate the Bézier-surface routines with

```
glDisable {GL_MAP2_VERTEX_3}
```

Coordinate positions on the Bézier surface can be calculated with

```
glEvalCoord2* (uValue, vValue);
```

or

```
glEvalCoord2*v (uvArray);
```


https://hemanthrajhemu.github.io

Parameter `uValue` is assigned some value in the interval from `uMin` to `uMax`, and parameter `vValue` is assigned some value in the interval from `vMin` to `vMax`. With the vector version, `uvArray` = (`uValue`, `vValue`). The suffix code for either function can be `f` or `d`. Function `glEvalCoord2` calculates a coordinate position using Equation 36 with the parameter values

$$u = \frac{u\text{Value} - u\text{Min}}{u\text{Max} - u\text{Min}}, \qquad v = \frac{v\text{Value} - v\text{Min}}{v\text{Max} - v\text{Min}} \tag{85}$$

which maps each of `uValue` and `vValue` to the interval from 0 to 1.0.

To display a Bézier surface, we repeatedly invoke `glEvalCoord2`, which generates a series of `glVertex3` functions. This is similar to generating a spline curve, except that we now have two parameters, u and v. For example, a surface defined with 16 control points, arranged in a 4×4 grid, can be displayed as a set of surface lines with the following code. The offset for the coordinate values in the u direction is 3, and the offset in the v direction is 12. Each coordinate position is specified with three values, and the y coordinate for each group of four positions is constant.

```
GLfloat ctrlPts [4][4][3] = {
   { {-1.5, -1.5,  4.0}, {-0.5, -1.5,  2.0},
     {-0.5, -1.5, -1.0}, { 1.5, -1.5,  2.0} },
   { {-1.5, -0.5,  1.0}, {-0.5, -0.5,  3.0},
     { 0.5, -0.5,  0.0}, { 1.5, -0.5, -1.0} },
   { {-1.5,  0.5,  4.0}, {-0.5,  0.5,  0.0},
     { 0.5,  0.5,  3.0}, { 1.5,  0.5,  4.0} },
   { {-1.5,  1.5, -2.0}, {-0.5,  1.5, -2.0},
     { 0.5,  1.5,  0.0}, { 1.5,  1.5, -1.0} }
};

glMap2f (GL_MAP2_VERTEX_3, 0.0, 1.0, 3, 4,
               0.0, 1.0, 12, 4, &ctrlPts[0][0][0]);
glEnable (GL_MAP2_VERTEX_3);

GLint k, j;

glColor3f (0.0, 0.0, 1.0);
for (k = 0; k <= 8; k++)
{
   glBegin (GL_LINE_STRIP);  // Generate Bezier surface lines.
    for (j = 0; j <= 40; j++)
        glEvalCoord2f (GLfloat (j) / 40.0, GLfloat (k) / 8.0);
    glEnd ( );
    glBegin (GL_LINE_STRIP);
    for (j = 0; j <= 40; j++)
        glEvalCoord2f (GLfloat (k) / 8.0, GLfloat (j) / 40.0);
    glEnd ( );
}
```

Instead of using the `glEvalCoord2` function, we can generate evenly spaced parameter values over the surface with

```
glMapGrid2* (nu, u1, u2, nv, v1, v2);
glEvalMesh2 (mode, nu1, nu2, nv1, nv2);
```


https://hemanthrajhemu.github.io

The suffix code for glMapGrid2 is again either f or d, and parameter mode can be assigned the value GL_POINT, GL_LINE, or GL_FILL. A two-dimensional grid of points is produced, with nu equally spaced intervals between u1 and u2, and with nv equally spaced intervals between v1 and v2. The corresponding integer range for parameter u is nu1 to nu2, and the corresponding integer range for parameter v is nv1 to nv2.

For a surface that is to be displayed as a grid of polylines, the output of glMapGrid2 and glEvalMesh2 is the same as the following program sequence except for the conditions that avoid round-off error at the beginning and ending values of the loop variables. At the beginning of the loops, the argument of glEvalCoord1 is set to (u1, v1), and at the end of the loops, the argument of glEvalCoord1 is set to (u2, v2).

```
for (k = nu1; k <= nu2; k++) {
    glBegin (GL_LINES);
        for (j = nv1; j <= nv2; j++)
           glEvalCoord2f (u1 + k * (u2 - u1) / nu,
                            v1 + j * (v2 - v1) / nv);
    glEnd ( );
}
for (j = nv1; j <= nv2; j++) {
    glBegin (GL_LINES);
        for (k = nu1; k <= nu2; k++)
           glEvalCoord2f (u1 + k * (u2 - u1) / nu,
                            v1 + j * (v2 - v1) / nv);
    glEnd ( );
}
```

Similarly, for a surface displayed as a set of filled-polygon facets (mode = GL_FILL), the output of glMapGrid2 and glEvalMesh2 is the same as the following program sequence, except for the round-off avoiding conditions for the beginning and ending values of the loop variables:

```
for (k = nu1; k < nu2; k++) {
    glBegin (GL_QUAD_STRIP);
        for (j = nv1; j <= nv2; j ++) {
            glEvalCoord2f (u1 + k * (u2 - u1) / nu,
                            v1 + j * (v2 - v1) / nv);
            glEvalCoord2f (u1 + (k + 1) * (u2 - u1) / nu,
                            v1 + j * (v2 - v1) / nv);
```

We can use the glMap2 function to designate values for other kinds of data, just as we did with glMap1. Similar symbolic constants, such as GL_MAP2_COLOR_4 and GL_MAP2_NORMAL, are available for this purpose. And we can activate multiple glMap2 functions to generate various data combinations.

GLU B-Spline Curve Functions

Although the GLU B-spline routines are referred to as NURBs functions, they can be used to generate B-splines that are neither nonuniform nor rational. Thus, we

https://hemanthrajhemu.github.io

can use these GLU routines to display a polynomial B-spline that has uniform knot spacing. And the GLU routines can also be used to produce Bézier splines, rational or nonrational. To generate a B-spline (or Bézier spline), we need to define a name for the spline, activate the GLU B-spline renderer, and then define the spline parameters.

The following statements illustrate the basic sequence of calls for displaying a B-spline curve:

```
GLUnurbsObj *curveName;

curveName = gluNewNurbsRenderer ( );
gluBeginCurve (curveName);
    gluNurbsCurve (curveName, nknots, *knotVector, stride, *ctrlPts,
                    degParam, GL_MAP1_VERTEX_3);
gluEndCurve (curveName);
```

In the first statement, we assign a name to the curve, then we invoke the GLU B-spline rendering routines for that curve using the `gluNewNurbsRenderer` command. A value of 0 is assigned to `curveName` when there is not enough memory available to create a B-spline curve. Inside a `gluBeginCurve/gluEndCurve` pair, we next state the attributes for the curve using a `gluNurbsCurve` function. This allows us to set up multiple curve sections, and each section is referenced with a distinct curve name. Parameter `knotVector` designates the set of floating-point knot values, and integer parameter `nknots` specifies the number of elements in the knot vector. The degree of the polynomial is `degParam` − 1. We list the values for the three-dimensional, control-point coordinates in array parameter `ctrlPts`, which contains `nknots` − `degParam` elements. And the integer offset between the start of successive coordinate positions in array `ctrlPts` is specified by integer parameter `stride`. If the control-point positions are contiguous (not interspersed between other data types), the value of `stride` is set to 3. We eliminate a defined B-spline with

```
gluDeleteNurbsRenderer (curveName);
```

As an example of the use of GLU routines to display a spline curve, the following code generates a cubic, Bézier polynomial. To obtain this cubic curve, we set the degree parameter to the value 4. We use four control points, and we select an eight-element, open-uniform knot sequence with four repeated values at each end.

```
GLfloat knotVector [8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
GLfloat ctrlPts [4][3] = { {-4.0, 0.0, 0.0}, {-2.0, 8.0, 0.0},
                           {2.0, -8.0, 0.0}, {4.0, 0.0, 0.0} };
GLUnurbsObj *cubicBezCurve;

cubicBezCurve = gluNewNurbsRenderer ( );
gluBeginCurve (cubicBezCurve);
    gluNurbsCurve (cubicBezCurve, 8, knotVector, 3, &ctrlPts [0][0],
                    4, GL_MAP1_VERTEX_3);
gluEndCurve(cubicBezCurve);
```


https://hemanthrajhemu.github.io

To create a rational B-spline curve, we replace the symbolic constant GL_MAP1_VERTEX_3 with GL_MAP1_VERTEX_4. Four-dimensional, homogeneous coordinates (x_h, y_h, z_h, h) are then used to specify the control points, and the resulting homogeneous division produces the desired rational polynomial form.

We can also use the gluNurbsCurve function to specify lists of color values, normal vectors, or surface-texture properties, just as we did with the glMap1 and glMap2 functions. Any of the symbolic constants, such as GL_MAP1_COLOR_4 or GL_MAP1_NORMAL, can be used as the last argument in the gluNurbsCurve function. Each call is then listed inside the gluBeginCurve/gluEndCurve pair, with two restrictions: We cannot list more than one function for each data type, and we must include exactly one function to generate the B-spline curve.

A B-spline curve is divided automatically into a number of sections and displayed as a polyline by the GLU routines. However, a variety of B-spline rendering options can also be selected with repeated calls to the following GLU function:

```
gluNurbsProperty (splineName, property, value);
```

Parameter splineName is assigned the name of a B-spline, parameter hboxproperty is assigned a GLU symbolic constant that identifies the rendering property that we want to set, and parameter value is assigned either a floating-point numerical value or a GLU symbolic constant that sets the value for the selected property. Several gluNurbsProperty functions can be specified following the gluNewNurbsRenderer statement. Many of the properties that can be set using the gluNurbsProperty function are surface parameters, as described in the next section.

GLU B-Spline Surface Functions

The following statements illustrate a basic sequence of calls for generating a B-spline surface:

```
GLUnurbsObj *surfName

surfName = gluNewNurbsRenderer ( );
gluNurbsProperty (surfName, property1, value1);
gluNurbsProperty (surfName, property2, value2);
gluNurbsProperty (surfName, property3, value3);
    .
    .
    .
gluBeginSurface (surfName);
    gluNurbsSurface (surfName, nuKnots, uKnotVector, nvKnots,
                      vKnotVector, uStride, vStride, &ctrlPts [0][0][0],
                       uDegParam, vDegParam, GL_MAP2_VERTEX_3);
gluEndSurface (surfName);
```

In general, the GLU statements and parameters for defining a B-spline surface are similar to those for a B-spline curve. After invoking the B-spline rendering routines with gluNewNurbsRenderer, we could specify a number of optional surface-property values. Attributes for the surface are then set with a gluNurbsSurface

https://hemanthrajhemu.github.io

call. Multiple surfaces, each with a distinct identifying name, can be defined in this way. A value of 0 is returned to variable `surfName` by the system when there is not enough memory available to store a B-spline object. Parameters `uKnotVector` and `vKnotVector` designate the arrays of floating-point knot values in the parametric u and v directions. We specify the number of elements in each knot vector with parameters `nuKnots` and `nvKnots`. The degree of the polynomial in parameter u is given by the value of `uDegParam` − 1, and the degree of the polynomial in parameter v is the value of `vDegParam` − 1. We list the floating-point values for the three-dimensional, control-point coordinates in array parameter `ctrlPts`, which contains (`nuKnots` − `uDegParam`) × (`nvKnots` − `vDegParam`) elements. The integer offset between the start of successive control points in the parametric u direction is specified with integer parameter `uStride`, and the offset in the parametric v direction is specified with integer parameter `vStride`. We erase a spline surface to free its allocated memory with the same function (`gluDeleteNurbsRenderer`) we used for a B-spline curve.

A B-spline surface, by default, is displayed automatically as a set of polygon fill areas by the GLU routines, but we can choose other display options and parameters. Nine properties, with two or more possible values for each property, can be set for a B-spline surface. As an example of property setting, the following statements specify a wire-frame, triangularly tessellated display for a surface:

```
gluNurbsProperty (surfName, GLU_NURBS_MODE,
                        GLU_NURBS_TESSELLATOR);
gluNurbsProperty (surfName, GLU_DISPLAY_MODE,
                        GLU_OUTLINE_POLYGON};
```

The GLU tessellating routines divide the surface into a set of triangles and display each triangle as a polygon outline. In addition, these triangle primitives can be retrieved using the `gluNurbsCallback` function. Other values for property `GLU_DISPLAY_MODE` are `GLU_OUTLINE_PATCH` and `GLU_FILL` (the default value). With the value `GLU_OUTLINE_PATCH`, we also obtain a wire-frame display, but the surface is not divided into triangular sections. Instead, the original surface is outlined, along with any trimming curves that have been specified. The only other value that can be set for the property `GLU_NURBS_MODE` is `GLU_NURBS_RENDERER` (the default value), which renders objects without making tessellated data available for callback.

We set the number of sampling points per unit length with the properties `GLU_U_STEP` and `GLU_V_STEP`. The default value for each is 100. To set the u or v sampling values, we also must set the property `GLU_SAMPLING_METHOD` to the value `GLU_DOMAIN_DISTANCE`. Several other values can be used with the property `GLU_SAMPLING_METHOD` to specify how surface tessellation is to be carried out. Properties `GLU_SAMPLING_TOLERANCE` and `GLU_PARAMETRIC_TOLERANCE` are used to set maximum sampling lengths. By setting property `GLU_CULLING` to the value `GL_TRUE`, we can improve rendering performance by not tessellating objects that are outside the viewing volume. The default value for GLU culling is `GL_FALSE`, and the property `GLU_AUTO_LOAD_MATRIX` allows the matrices for the viewing, projection, and viewport transformations to be downloaded from the OpenGL server when its value is `GL_TRUE` (the default value). Otherwise, if we set the value to `GL_FALSE`, an application must supply these matrices using the `gluLoadSamplingMatrices` function.

https://hemanthrajhemu.github.io

To determine the current value of a B-spline property, we use the following query function:

```
gluGetNurbsProperty (splineName, property, value);
```

For a specified splineName and property, the corresponding value is returned to parameter value.

When the property GLU_AUTO_LOAD_MATRIX is set to the value GL_FALSE, we invoke

```
gluLoadSamplingMatrices (splineName, modelviewMat, projMat,
                         viewport);
```

This function specifies the modelview matrix, projection matrix, and viewport that are to be used in the sampling and culling routines for a spline object. The current modelview and projection matrices can be obtained with calls to the glGetFloatv function, and the current viewport can be obtained with a call to glGetIntegerv.

Various events associated with spline objects are processed using

```
gluNurbsCallback (splineName, event, fcn);
```

Parameter event is assigned a GLU symbolic constant, and parameter fcn specifies a function that is to be invoked when the event corresponding to the GLU constant is encountered. For example, if we set parameter event to GLU_NURBS_ERROR, then fcn is called when an error occurs. Other events are used by the GLU spline routines to return the OpenGL polygons generated by the tessellation process. The symbolic constant GL_NURBS_BEGIN indicates the start of a primitive such as line segments, triangles, or quadrilaterals, and GL_NURBS_END indicates the end of the primitive. The function argument for the beginning of a primitive is then a symbolic constant such as GL_LINE_STRIP, GL_TRIANGLES, or GL_QUAD_STRIP. Symbolic constant GL_NURBS_VERTEX indicates that three-dimensional coordinate data are to be supplied, and a vertex function is called. Additional constants are available for indicating other data, such as color values.

Data values for the gluNurbsCallback function are supplied by

```
gluNurbsCallbackData (splineName, dataValues);
```

Parameter splineName is assigned the name of the spline object that is to be tessellated, and parameter dataValues is assigned a list of data values.

GLU Surface-Trimming Functions

A set of one or more two-dimensional trimming curves is specified for a B-spline surface with the following statements:

```
gluBeginTrim (surfName);
   gluPwlCurve (surfName, nPts, *curvePts, stride, GLU_MAP1_TRIM_2);
   .
   .
   .
gluEndTrim (surfName);
```


https://hemanthrajhemu.github.io

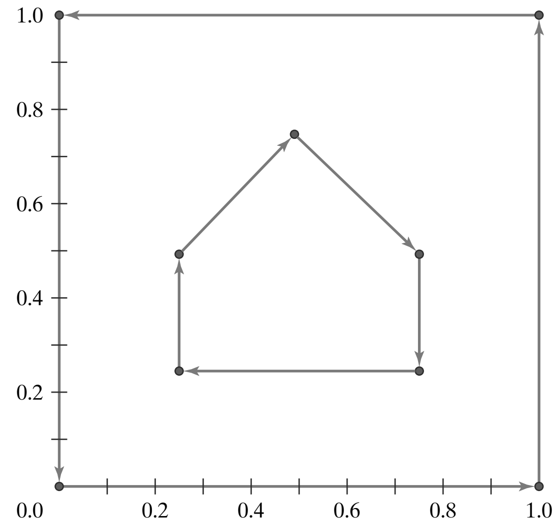


FIGURE 40
An outer trimming curve around the perimeter of the unit square is specified in a counterclockwise direction, and the inner trimming curve sections are defined in a clockwise direction.

Parameter `surfName` is the name of the B-spline surface to be trimmed. A set of floating-point coordinates for the trimming curve is specified in array parameter `curvePts`, which contains `nPts` coordinate positions. An integer offset between successive coordinate positions is given in parameter `stride`. The specified curve coordinates are used to generate a piecewise linear trimming function for the B-spline surface. In other words, the generated trimming "curve" is a polyline. If the curve points are to be given in three-dimensional, homogeneous (u, v, h) parameter space, then the final argument in `gluPwlCurve` is set to the GLU symbolic constant `GLU_MAP1_TRIM_3`.

We can also use one or more `gluNurbsCurve` functions as a trimming curve. In addition, we can construct trimming curves that are combinations of `gluPwlCurve` functions and `gluNurbsCurve` functions. Any specified GLU trimming "curve" must be nonintersecting, and it must be a closed curve.

The following code illustrates the GLU trimming functions for a cubic Bézier surface. We first set the coordinate points for an outermost trimming curve. These positions are specified in a counterclockwise direction completely around the unit square. Next, we set the coordinate points for an innermost trimming curve in two sections, and these positions are specified in a clockwise direction. And the knot vectors for both the surface and the first inner trim-curve section are set up to produce cubic Bézier curves. A plot of the inner and outer trimming curves on the unit square is shown in Figure 40.

```
GLUnurbsObj *bezSurface;

GLfloat outerTrimPts [5][2] = { {0.0, 0.0}, {1.0, 0.0}, {1.0, 1.0),
                                {0.0, 1.0}, {0.0, 0.0} };
GLfloat innerTrimPts1 [3][2] = { {0.25, 0.5}, {0.5, 0.75},
                                {0.75, 0.5) };
GLfloat innerTrimPts2 [4][2] = { {0.75, 0.5}, {0.75, 0.25},
                                {0.25, 0.25), {0.25, 0.5} };

GLfloat surfKnots [8] = (0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0);
GLfloat trimCurveKnots [8] = (0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0);
```


https://hemanthrajhemu.github.io

```
bezSurface = gluNewNurbsRenderer ( );

gluBeginSurface (bezSurface);
    gluNurbsSurface (bezSurface, 8, surfKnots, 8, surfKnots, 4 * 3, 3,
                     &ctrlPts [0][0][0], 4, 4, GL_MAP2_VERTEX_3);
    gluBeginTrim (bezSurface);
       /*  Counterclockwise outer trim curve.  */
       gluPwlCurve (bezSurface, 5, &outerTrimPts [0][0], 2,
                         GLU_MAP1_TRIM_2);
    gluEndTrim (bezSurface);
    gluBeginTrim (bezSurface);
       /*  Clockwise inner trim-curve sections.  */
       gluPwlCurve (bezSurface, 3, &innerTrimPts1 [0][0], 2,
                         GLU_MAP1_TRIM_2);
       gluNurbsCurve (bezSurface, 8, trimCurveKnots, 2,
                      &innerTrimPts2 [0][0], 4, GLU_MAP1_TRIM_2);
    gluEndTrim (bezSurface);
gluEndSurface (bezSurface);
```

17 Summary

The most widely used methods for representing objects in CAD applications are the spline representations, which are piecewise continuous polynomial functions. A spline curve or surface is defined with a set of control points and the boundary conditions on the spline sections. Lines connecting the sequence of control points form the control graph, and all control points are within the convex hull of a spline object. The boundary conditions can be specified using parametric or geometric derivatives, and most spline representations use parametric boundary conditions. Interpolation splines connect all control points; approximation splines do not connect all control points. A spline surface can be described with the tensor product of two polynomials. Cubic polynomials are commonly used for the interpolation representations, which include the Hermite, cardinal, and Kochanek-Bartels splines. Bézier splines provide a simple and powerful approximation method for describing curved lines and surfaces, however the polynomial degree is determined by the number of control points and local control over curve shapes is difficult to attain. B-splines, which include Bézier splines as a special case, are a more versatile approximation representation, but they require the specification of a knot vector. Beta splines are generalizations of B-splines that are specified with geometric boundary conditions. Rational splines are formulated as the ratio of two spline representations. Rational splines can be used to describe quadrics, and they are invariant with respect to a perspective viewing transformation. A rational B-spline with a nonuniform knot vector is commonly referred to as a NURB. To determine the coordinate positions along a spline curve or surface, we can use forward-difference calculations or subdivision methods.

The core library of OpenGL contains functions for producing Bézier splines, and GLU functions are furnished for specifying B-splines and spline-surface trimming curves. Tables 1 and 2 summarize the OpenGL spline functions discussed in this chapter.

https://hemanthrajhemu.github.io

TABLE 1

Summary of OpenGL Bezier Functions

Function	Description
`glMap1`	Specifies parameters for Bézier-curve display, color values, etc., and activate these routines using `glEnable`.
`glEvalCoord1`	Calculates a coordinate position for a Bézier curve.
`glMapGrid1`	Specifies the number of equally spaced subdivisions between two Bézier-curve parameters.
`glEvalMesh1`	Specifies the display mode and integer range for a Bézier-curve display.
`glMap2`	Specifies parameters for a Bézier-surface display, color values, etc., and activate these routines using `glEnable`.
`glEvalCoord2`	Calculates a coordinate position for a Bézier surface.
`glMapGrid2`	Specifies a two-dimensional grid of equally spaced subdivisions over a Bézier surface.
`glEvalMesh2`	Specifies the display mode and integer range for the two-dimensional Bézier-surface grid.

TABLE 2

Summary of OpenGL B-Spline Functions

Function	Description
`gluNewNurbsRenderer`	Activates the GLU B-spline renderer for an object name that has been defined with the declaration `GLUnurbsObj *bsplineName`.
`gluBeginCurve`	Begins the assignment of parameter values for a specified B-spline curve with one or more sections.
`gluEndCurve`	Signals the end of the B-spline curve parameter specifications.
`gluNurbsCurve`	Specifies the parameter values for a named B-spline curve section.
`gluDeleteNurbsRenderer`	Eliminates a specified B-spline.
`gluNurbsProperty`	Specifies rendering options for a designated B-spline.
`gluGetNurbsProperty`	Determines the current value of a designated property for a particular B-spline.

https://hemanthrajhemu.github.io

TABLE 2

Summary of OpenGL B-Spline Functions (*Continued*)

Function	Description
`gluBeginSurface`	Begins the assignment of parameter values for a specified B-spline surface with one or more sections.
`gluEndSurface`	Signals the end of the B-spline surface parameter specifications.
`gluNurbsSurface`	Specifies the parameter values for a named B-spline surface section.
`gluLoadSamplingMatrices`	Specifies viewing and geometric transformation matrices to be used in sampling and culling routines for a B-spline.
`gluNurbsCallback`	Specifies a callback function for a designated B-spline and associated event.
`gluNurbsCallbackData`	Specifies data values that are to be passed to the event callback function.
`gluBeginTrim`	Begins the assignment of trimming-curve parameter values for a B-spline surface.
`gluEndTrim`	Signals the end of the trimming curve parameter specifications.
`gluPwlCurve`	Specifies trimming-curve parameter values for a B-spline surface.

REFERENCES

Sources of information on parametric curve and surface representations include Bézier (1972), Barsky and Beatty (1983), Barsky (1984), Kochanek and Bartels (1984), Huitric and Nahas (1985), Mortenson (1985), Farin (1988), Rogers and Adams (1990), and Piegl and Tiller (1997).

Programming techniques for various representations can be found in Glassner (1990), Arvo (1991), Kirk (1992), Heckbert (1994), and Paeth (1995). Additional programming examples for the OpenGL Bézier-spline, B-spline, and trimming-curve functions can be found in Woo, et al. (1999). And a complete listing of the OpenGL functions in the core library and in GLU is presented in Shreiner (2000).

EXERCISES

1 Write a routine to display a two-dimensional cardinal-spline curve, given an input set of control points in the xy plane.

2 Write a program using the routine developed in the previous exercise to display a two-dimensional cardinal spline curve in the xy plane along with the control points used to generate the curve. The curve should be drawn in black (on a white background) and the control points should be drawn in blue. Additionally, allow the user to modify the control points via keyboard input. The user should be able to cycle through the control points and move each one around in the xy plane. The currently selected control point should be drawn in red. The curve should be redrawn each time a control point is moved.

3 Write a routine to display a two-dimensional Kochanek-Bartels curve, given an input set of control points in the xy plane.

4 Write a program using the routine developed in the previous exercise similar to the program in Exercise 2. Control points should be drawn in addition to the curve on a white background and the user should be able to edit the control points in the same manner. The curve should be redrawn each time a control point is moved.

https://hemanthrajhemu.github.io

5 What are the Bézier-curve blending functions for three control points specified in the xy plane? Plot each function and identify the minimum and maximum blending-function values.

6 What are the Bézier-curve blending functions for five control points specified in the xy plane? Plot each function and identify the minimum and maximum blending-function values.

7 Modify the program example in Section 8 to display any cubic Bézier curve, given a set of four input control points in the xy plane.

8 Modify the program example in Section 8 to display a Bézier curve of degree $n-1$, given a set of n input control points in the xy plane.

9 Complete the OpenGL programming example in Section 8 to display any cubic Bézier curve, given a set of four input control points in the xy plane.

10 Modify the program in the previous exercise to allow the user to edit the control points using keyboard input as in Exercise 2. The currently selected control point should be drawn in red, and the others in blue. The curve should be drawn in black and redrawn each time a control point is moved.

11 Modify the OpenGL program example in Section 8 to display any spatial cubic Bézier curve, given a set of four input control points in xyz space. Use an orthogonal projection to display the curve, with the viewing parameters specified as input.

12 Write a routine that can be used to design two-dimensional Bézier curve shapes that have first-order piecewise continuity. The number and position of the control points for each section of the curve are to be specified as input.

13 Use the routine developed in the previous exercise to allow the user to edit the control points using keyboard input as in Exercise 2. Controls points should be displayed in the same manner.

14 Write a routine that can be used to design two-dimensional Bézier curve shapes that have second-order piecewise continuity. The number and position of the control points for each section of the curve are to be specified as input.

15 Use the routine developed in the previous exercise to allow the user to edit the control points using keyboard input as in Exercise 2. Controls points should be displayed in the same manner.

16 Modify the program example in Section 8 to display any cubic Bézier curve, given a set of four input control points in the xy plane, using the subdivision method to calculate curve points.

17 Modify the program example in Section 8 to display any cubic Bézier curve, given a set of four input control points in the xy plane, using forward differences to calculate curve points.

18 What are the blending functions for a two-dimensional, uniform, periodic B-spline curve with $d = 5$?

19 What are the blending functions for a two-dimensional, uniform, periodic B-spline curve with $d = 6$?

20 Modify the programming example in Section 10 to display a two-dimensional, uniform, periodic B-spline curve, given an input set of control points, using forward differences to calculate positions along the curve path.

21 Modify the program in the previous example to display the B-spline curve using OpenGL functions.

22 Modify the program in the previous exercise to allow the user to edit the control points using keyboard input as in Exercise 2. Controls points should be displayed in the same manner.

23 Write a routine to display any specified conic in the xy plane using a rational Bézier-spline representation.

24 Write a routine to display any specified conic in the xy plane using a rational B-spline representation.

25 Develop an algorithm for calculating the normal vector to a Bézier surface at a given point $\mathbf{P}(u, v)$.

26 Derive expressions for calculating the forward differences for a given quadratic curve.

27 Derive expressions for calculating the forward differences for a given cubic curve.

IN MORE DEPTH

1 In this chapter's exercises, you will experiment with creating and displaying three-dimensional spline surfaces to represent some of the more complex curved objects in your application. Choose some objects that fit this category in your scene and sketch out either a Bézier spline or B-spline representation of their surfaces using the methods discussed in the chapter. Once you have chosen a representation, use the OpenGL functions for displaying spline surfaces to render the objects in the scene using the default resolution of evaluation points (or a reasonable one, in the case of Bézier surfaces). Then, use the visual rendering of the objects to adjust the spline model to improve the visual accuracy of the objects. Use trimming curves where appropriate to produce the right object shapes.

2 Experiment with varying the resolution of the polygon meshes that serve as the approximations to the spline surfaces that

https://hemanthrajhemu.github.io

you defined in the previous exercise. For Bézier surfaces, choose a minimum number of evaluation points in each dimension at which the representation of the objects is minimally acceptable as far as visual appearance goes. Do the same for any B-spline representations, varying the number of sampling points instead. Using this as a baseline, render the scene from the previous exercise several times, each time increasing the number of evaluation or sampling points that define the mesh approximations of the objects by some fixed amount. For each setting of resolution, record the amount of time that it takes to render the scene using shaded fill areas to render the objects. Continue doing this until the resolution produces little or no noticeable difference in approximation quality. Then, make a plot of rendering time as a function of resolution and discuss the properties of the plot. Is there an ideal setting for each object that balances visual quality with performance?

https://hemanthrajhemu.github.io

This page intentionally left blank

https://hemanthrajhemu.github.io