Chapter 14

Curves and Surfaces

This chapter describes how to draw curves and surfaces using *Non-uniform Rational B-splines (NURBS)*. NURBS are useful for creating the types of smooth curves and surfaces that you see on airplane wings, cars, and machinery. You can render NURBS with color, lighting, and texture.You can also cut holes in the interior of a NURBS surface to create more complex surfaces.

- Section 14.1, “Introduction to Curves,” develops the background and terminology for curves and surfaces.
- Section 14.2, “B-Splines,” discusses characteristics of B-Splines.
- Section 14.3, “GL NURBS Curves,” tells you how to draw NURBS curves.
- Section 14.4, “NURBS Surfaces,” tells you how to draw NURBS surfaces.
- Section 14.5, “Trimming NURBS Surfaces,” tells you how to cut holes in NURBS surfaces.
- Section 14.6, “NURBS Properties,”describes how NURBS are rendered.
- Section 14.8, “Old-Style Curves and Surfaces,” describes the GL method of defining curves and surfaces that was used in previous releases of the software. This section is included for compatibility only—all new development should use the GL NURBS method.

The GL draws NURBS curves by efficiently approximating them with line segments, and it draws NURBS surfaces by efficiently approximating them with polygon meshes. This means you can use the GL commands for transformations, lighting, and hidden-surface removal to control how NURBS curves and surfaces are drawn, just as you do with other GL primitives.

To see an interactive NURBS curve demonstration, use the Demos Toolchest on your workstation to look at *Curve*s and *NURBS* under *GL Demos*.

You can develop an intuitive understanding about NURBS by studying the text and accompanying illustrations presented in this chapter. For a more rigorous mathematical treatment, consult the “Suggestions for Further Reading” at the end of this chapter.

14.1 Introduction to Curves

This section discusses curves. First, basic curve information is presented, then B-Splines are discussed, and finally, NURBS are introduced. Once understood, principles learned about curves are easily extended to surfaces.

14.1.1 Parametric Curves

The most commonly used representation for curves (and for surfaces) is the *parametric* form, which is the one used in the GL; it is discussed in detail in the next section. You may be aware of other forms such as *implicit* or *algebraic*. An implicit or algebraic representation is of the form:

$$f(x, y) = 0 \qquad \text{(EQ 14-6)}$$

The following *explicit* representation is an alternative to the implicit form:

$$y = f(x) \qquad \text{(EQ 14-7)}$$

In this case, x is the independent variable and y is the dependent variable.

In the parametric representation, the coordinate functions x,y, (and z) are all explicit functions of a *parameter*. In this chapter, this parameter is called u.

In Figure 14-1, the function $\vec{F}(u)$ maps points from a subset of the real line $\Re$, to model coordinates (which can be 2-D, 3-D, or 4-D).

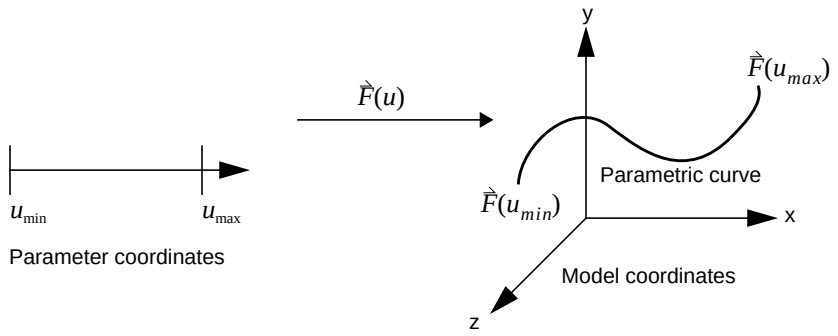

Figure 14-1 Parametric Curve as a Mapping

Call this vector-valued function the curve $\vec{C}$, which is defined for values of u in a given subset of the real line:

$$\vec{C}(u) = (x(u), y(u), z(u)), \quad \text{where} u \in [u_{min}, u_{max}] \in \Re \qquad \text{(EQ 14-8)}$$

As u takes on values from u_{min} to u_{max}, a *parametric curve* is traced out.

Each coordinate on the curve is a function of the parameter u.
Let u^* represent a specific value of u. The coordinates of the curve at that value of u are obtained by evaluating the functions x, y, and z at u^*:

$$\vec{C}(u^*) = (x^*, y^*, z^*), \quad \text{where} \quad x^* = x(u^*), y^* = y(u^*), z^* = z(u^*) \qquad 4\text{-}9)$$

Figure 14-2 is called a *cross-plot*. A cross-plot helps you visualize how values from the real line are mapped to a parametric curve.

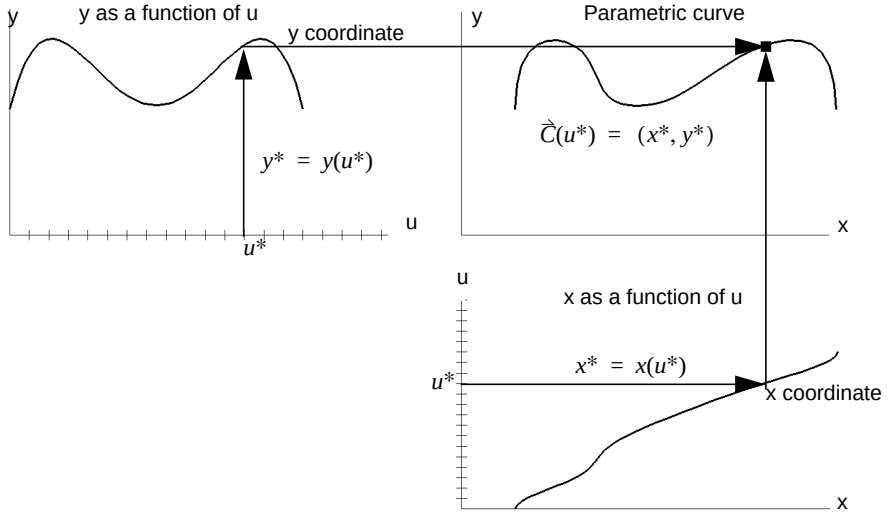

Figure 14-2 Cross-plot

The upper right graph in Figure 14-2 is a plot of a curve in 2-D model coordinates (x,y). The upper left graph is a plot of the y coordinate as a function of the parameter u: $(u, y(u))$. Likewise, the lower right graph is a plot of x as a function of u that is drawn with u on the vertical axis $(x(u), u)$.

For a particular value of u, u^*, the corresponding $y(u^*)$ and $x(u^*)$ are found, as shown by the arrows from the u axis to the functions. By drawing horizontal and vertical lines as shown, the point (x^*,y^*) is located on the parametric curve.

14.1.2 Polynomial Curves

Curves can be represented mathematically as *polynomials*. The equations below, showing x and y as functions of u are examples of *cubic polynomials*. The *degree* of the polynomial is 3, because the highest exponent is 3.

(EQ 14-10)

$$x(u) = a_0 + a_1 u + a_2 u^2 + a_3 u^3$$

$$y(u) = b_0 + b_1 u + b_2 u^2 + b_3 u^3$$

The a_i's and b_i's in the polynomials above are called *coefficients*, also known as *control points*. Notice that the total number of coefficients in each polynomial is one greater than the degree. This number is also called the *order* of the polynomial. Order is equal to the degree plus one, so these are polynomials of order 4.

Putting the two polynomials into the parametric representation of the curve $\vec{C}$, gives you a *parametric cubic polynomial* curve:

$$\vec{C}(u) = (x(u), y(u)) \qquad \text{(EQ 14-11)}$$

14.1.3 Parametric Spline Curves

You can join polynomials together to make a *piecewise polynomial*, which gives you more flexibility in shaping the curve. This piecewise polynomial is called a *spline*. The order of a spline is defined as the maximum order of the component polynomial pieces. Generally, each polynomial has the same order. If necessary, you can elevate the order of the lower degree polynomials to the order of the highest order piece, using a process called *degree elevation* (See "B-Spline Basics" in *Curves and Surfaces for Computer Aided Geometric Design*). The breakpoints where the polynomial pieces are joined are called *knots*.

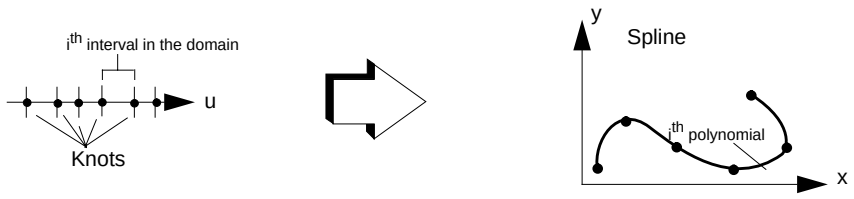

Figure 14-3 Spline

14.2 B-Splines

A *B-Spline* is a compact representation for piecewise polynomials. In the previous section, you learned that the knots of a spline tell you where one polynomial segment ends and a new one begins. This section describes how to shape the curve segments.

14.2.1 Control Points

Associated with a B-Spline is a set of points called *control points*. You use control points to shape the curve. Control points are like magnets—they pull the curve into a certain shape (see Figure 14-4).

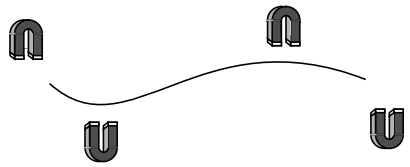

Figure 14-4 Influence of Control Points on a Curve

Connecting the control points in order forms a *control polygon*. The B-Spline lies close to its control polygon, as shown in Figure 14-5.

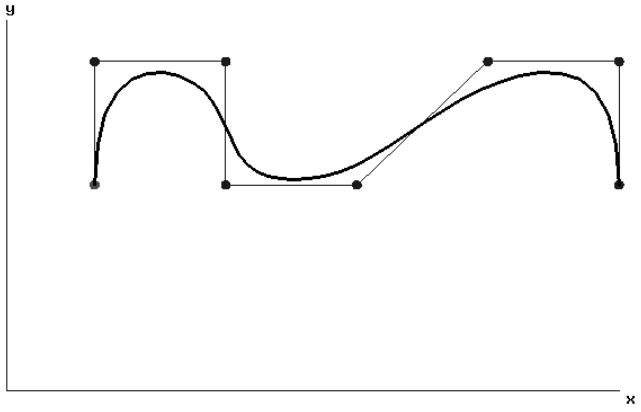

Figure 14-5 A B-Spline with Its Control Polygon

Moving a control point influences the shape of the curve segment near it. Figure 14-6 shows how the shape of the B-Spline changes when you move one of its control points.

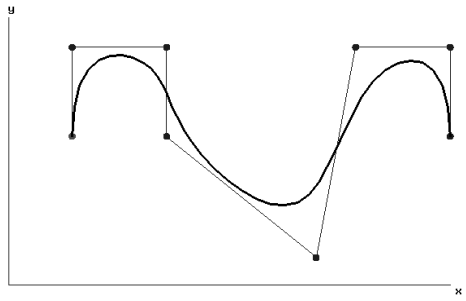

Figure 14-6 Moving a Control Point to Change the Shape of a B-Spline

The curve is really a weighted average of the influence of the control points. The notation below represents this concept mathematically:

$$\vec{C}(u) = \sum \vec{d}_i B_i(u) \qquad \text{(EQ 14-12)}$$

In this representation, the d_i's are the control points and the $B_i(u)$'s are known as *basis functions*.

14.2.2 Basis Functions

Basis functions determine *how much* the control points influence the curve, *which* control points influence the curve and *where* the curve is influenced by each control point. These rules govern basis functions used in the GL:

1. There is a basis function corresponding to each control point.
2. Basis functions are positive (they attract rather than repel the curve).
3. The curve is defined only where *order* number of basis functions are defined.
4. At most, only *order* number of basis functions are non-zero at a time.
5. At any u in the domain, the basis functions add up to exactly 1.
6. The number of knots equals the number of control points plus the order.
7. The knots are given in a non-decreasing sequence.

Figure 14-7 shows an example of the basis functions for a cubic B-Spline. The basis functions in this example are uniformly spaced and are identical, except for a translation. Since the order is 4, the curve is defined only for values of u where all four basis functions are defined (nonzero), within the shaded area.

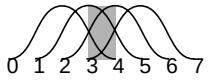

Figure 14-7 Basis Functions for an Example Cubic B-Spline

14.2.3 Knots

Knots determine *how* and *where* the basis functions are defined. The knots in the example shown in Figure 14-7 are placed at 0,1,2,3,4,5,6, and 7. This series of numbers is called the *knot sequence*. A knot sequence is specified as a series of non-decreasing real numbers. There can be multiple knots, where two or more knots have the same value. The term *knot multiplicity* is used to refer to the number of knots that have the same value for any one location.

Increasing the knot multiplicity causes the curve to move closer to the corresponding control point. At a knot multiplicity equal to *order-1*, the curve passes through the control point. In general, there may be a discontinuity in the curve at the point where the knot multiplicity is equal to the order.

Figure 14-8 shows an example of the basis functions for a cubic B-Spline with a multiple interior knot. The knot multiplicities are listed below the *u*-axis and each basis function is numbered. A knot of multiplicity of 3 exists at u=5. The basis functions always sum to 1, so as u approaches 5, basis function 4, $B_4(u)$, is increasing in value, while the other basis functions are decreasing in value. At u=5, $B_4=1$ and all the other basis functions are zero.

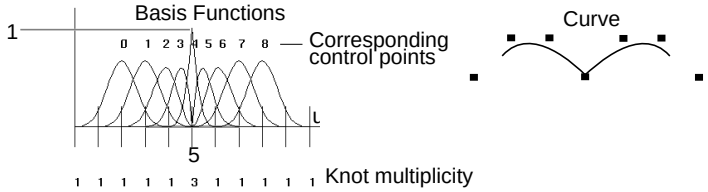

Figure 14-8 Basis Functions for a Cubic B-Spline with a Multiple Interior Knot

Figure 14-9 shows the relationships between the B-Spline and its basis functions, control points, and knots.

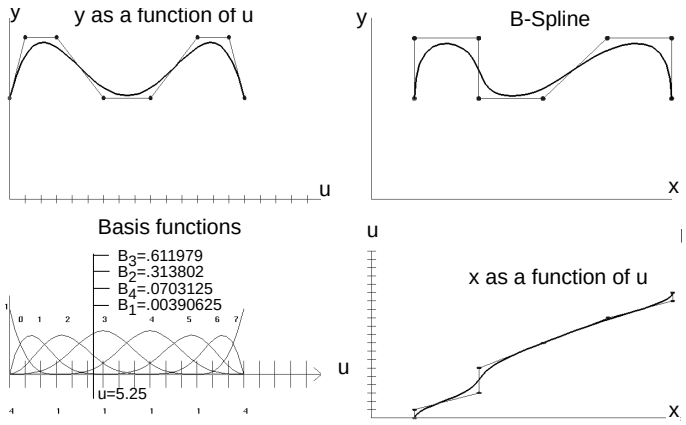

Figure 14-9 B-Spline Relationships

Review the rules governing B-Spline basis functions as you study the figure.

In this example of a cubic B-Spline, there are 8 control points and consequently there are 8 basis functions, which are numbered 0 through 7. (Rule 1)

The basis functions are positive (they attract the curve). (Rule 2)

The curve is defined where *order* number (4 in this case) of basis functions are defined. (Rule 3)

There are at most *order* number (4) of non-zero basis functions; for example, at u=5.25, the only non-zero basis functions are B_1, B_2, B_3, and B_4. (Rule 4)

Summing the basis functions (B_1+B_2+B_3+B_4) at u=5.25 gives:

.00390625 + .313802 + .611979 + .0703125 = 1 (Rule 5)

The numbers below the u axis in the basis function plot represent the knot multiplicity. Adding up these numbers gives you the total number of knots. In this example, the number of knots (12) equals the number of control points (8) plus the order (4). (Rule 6)

The knot sequence (0,0,0,0,3,6,9,12,15,15,15,15) is non-decreasing. (Rule 7)

The B-Spline in Figure 14-9 has knot multiplicities equal to its order (4) at the first and last knots. This causes the B-Spline to interpolate (pass through) the first and last control points. By using this technique, you can make a B-Spline go through its endpoints, which is useful for joining B-Spline segments.

14.2.4 Weights

When you want to create a curve that is influenced more by a particular control point than by the others, you can assign a *weight* to alter that point's influence.

The polynomial curves that you have learned about so far have had equally weighted control points. When you increase the weight of a control point, you increase its influence on the curve. When the weight of one control point increases, the influence of the other nearby control points decreases. (Rule 5)

Figure 14-10 shows the same B-Spline as Figure 14-9 with increased weight at the third control point (d_2). This control point now exerts more influence on the curve. The greater influence can also be seen in the corresponding basis function (B_2). B_2 is now much larger than the other 3 basis functions. This means that d_2 now has more influence than its nearby control points. In the magnet analogy of control points, this is equivalent to having a strong magnet at d_2 and much weaker magnets of equal strengths at the other control points.

Note: Use positive values for weights.

.

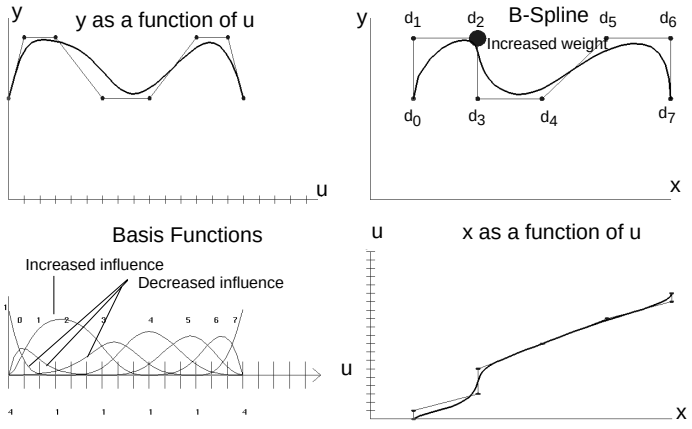

Figure 14-10 Increasing the Weight of a Control Point Increases Its Influence

14.3 GL NURBS Curves

So far, you have learned that curves can be represented with parametric polynomials. You know that polynomials can be joined to create a B-spline. You have seen how control points shape a B-spline and how knots and basis functions contribute to the definition of a B-Spline. You now have the information you need to understand what a NURBS representation is:

Non-**U**niform Knots do not have to be spaced at equal intervals.

Rational Ratio of polynomials (see Note below) allows the weights of the control points to be specified.

B-**S**pline B-Spline, basis functions are used.

Note: Throughout this discussion, polynomial is used, rather than nonrational, to signify curves whose control points are equally weighted (have x,y,z coordinates, but no (or equal) w coordinates).

Use the `nurbscurve()` subroutine to draw GL NURBS curves, by specifying the knot sequence, the control points, and the order. As in other GL constructions, call the `nurbscurve()` subroutine between a `bgncurve()/endcurve()` pair

The ANSI C specification for `nurbscurve()` is:

```
void nurbscurve(long knot_count, double knot_list, long
offset, double *ctlarray, long order, long type);
```

where:

knot_count is the number of knots in *knot_list*

knot_list is an array of *knot_count* non-decreasing *knot_values*

offset is the offset in bytes between successive control points

ctlarray is an array of control points

order is the order of the curve (degree +1)

type is the type of curve, either `N_V3D` for a polynomial (x,y,z) curve, or `N_V3DR` for a rational (*x*,*y*,*z*,*w*) curve

Note: There are two other curve types, `N_P2D` and `N_P2DR`, used as trimming curves. See Section 14.5, “Trimming NURBS Surfaces.”

Follow these rules for NURBS:

- *knot_count* = number of control points + order
- offset allows control points to be part of a structure
- control point coordinates are of form (*x*,*y*,*z*) for polynomial curves, (*wx*,*wy*,*wz*,*w*) for rational curves
- use positive weights
- maximum order is `getgdesc(GD_NURBS_ORDER)`

The following code fragment defines and draws a NURBS curve:

```
double ctlpts[4][3]={{-.75,-.75,0.0},{-.5,.75,0.0},{.5,.75,0.0},{.75,-.75,0.0}};
double knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
bgncurve();
    nurbscurve(8, knots, 3*sizeof(double), &ctlpts[0][0], 4, N_V3D);
endcurve();
```

NURBS have many interesting properties, for example, you can represent conic sections (circles, parabolas, ellipses, hyperbolas) exactly with NURBS. Figure 14-11 shows how to represent a circle with a quadratic NURBS curve:

```
double knots[14] = {0.0,0.0,0.0,2.5,2.5,5.,5.,7.5,7.5,10.,10.,10.}
double control_points[][4]={{9.,5.,0.1,},{6.369,6.369,0.,.708},{,5.,9.,0.,1.},
{.708,6.369,0.,.708}, {1.,5.,0.,1.}, {.708,.708,0.,.708}, {5.,1.,0.,1.},
{6.369,.708,0.,.708}, {9.,5.,0.,1.}};
```

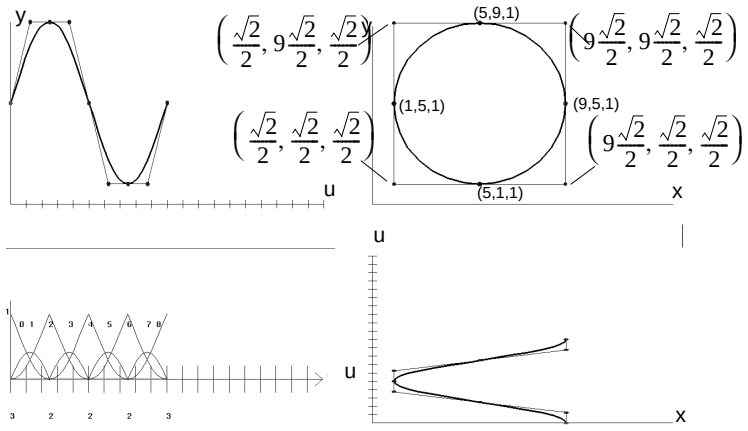

Figure 14-11 Representing a Circle with NURBS

14.4 NURBS Surfaces

A *parametric surface* is traced out as the two parameters *u* and *v* take on values in the domain (shaded region of the real plane), as shown in Figure 14-12. Parametric surfaces are analogous to the parametric curves you learned about in the beginning of the chapter, except that the domain now has two parameters, the basis functions are now three-dimensional, and the control points now connect to form a *control net*.

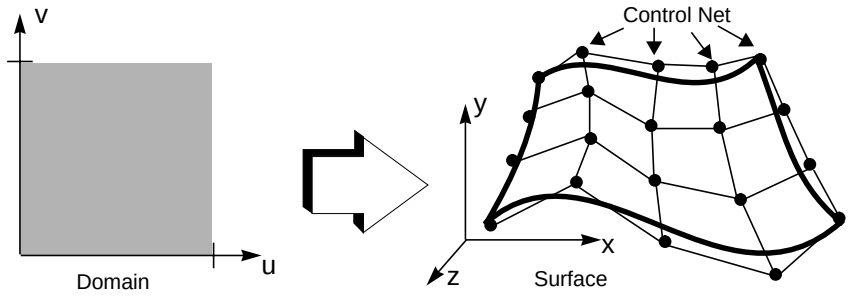

Figure 14-12 Parametric NURBS Surface

To create a NURBS surface, call `nurbssurface()` within a `bgnsurface()/endsurface()` pair.

There are three types of NURBS surfaces in the GL:

- *Geometric surface* - defines the geometry of the surface
- *Color surface* - defines the color of the surface as a NURBS surface
- *Texture surface* - defines the texture of the surface as a NURBS surface

Define one (and only one) geometric surface within each `bgn/endsurface` pair. You can optionally specify one color surface and one texture surface per geometric surface.

If you do specify a color and / or texture surface, the defined color / texture is applied to the NURBS surface in the same way that color-per-vertex or texture mapping is applied to polygons. Defining a color or texture NURBS surface is one way to apply color and texture to NURBS surfaces, but it is not the only way. See Chapter 4, “Display and Color Modes” and Chapter 18, “Texture” for more information about defining colors and textures.

You can also use the standard lighting models when rendering NURBS surfaces. With no other modifications, the NURBS surface appears in the currently bound material and with the currently bound lights and lighting model.

To draw a NURBS surface, specify:

- Control points - an array of data of the appropriate type for the surface
- Knots - a set of non-decreasing numbers for each parameter
- Order - (degree + 1) for each of the two surface parameters

There may be a different order and different knot sequence for each parameter.

As in the curve case, certain dependencies exist between the surface orders, the knot counts, and the number of control points. You specify the surface orders and the knot counts to determine the number and arrangement of the control points.

If O_u and O_v are the surface orders in the u and v directions, and if K_u and K_v are the knot counts in those directions, then the control points form an array of size: $(K_u - O_u) \times (K_v - O_v)$

The ANSI C specification for nurbssurface is:

```
void nurbssurface(long u_knot_count, double u_knots[],
                  long v_knot_count, double v_knots[],
                  long u_offset, long v_offset,
                  double *ctlarray,
                  long u_order, long v_order, long type);
```

The arguments to `nurbssurface()` are:

u_knot_count	number of knots in the *u* parameter of the domain
u_knot	array of *u_knot_count* nondecreasing knot values
v_knot_count	number of knots in the *v* parameter of the domain
v_knot	array of *v_knot_count* non-decreasing knot values
u_offset	offset in bytes between successive control points in *ctlarray* in the *u* parameter of the domain
v_offset	offset in bytes between successive control points in *ctlarray* in the *v* parameter of the domain

ctlarray	array of control points
u_order	order of the surface in *u*
v_order	order of the surface in *v*
type	type of surface, indicated by one of these types:
`N_V3D`	control points define a geometric surface in double-precision coordinates of the form (*x*, *y*, *z*).
`N_V3DR`	control points define a geometric surface in double-precision homogeneous coordinates of the form (*wx*, *wy*, *wz*, *w*).
`N_C4D`	control points define a color surface in double-precision color components of the form (R, G, B, and A).
`N_C4DR`	control points define a color surface in double-precision color components that are in homogeneous form.
`N_T2D`	control points define a texture surface in double-precision texture coordinates that exist in a two-dimensional (*s* and *t)* texture space.
`N_T2DR`	control points define a texture surface in double-precision texture coordinates that are in homogeneous form.

Note: When specifying control points, the coordinates must be appropriate for *type*. That is, if you specify *type* `N_V3D` for the point geometry, *ctlarray* must contain elements in the form of (*x*, *y*, *z*) triples. If you specify `N_C4D`, *ctlarray* must contain an array of (*R*, *G*, *B*, *A*) components in double-precision form.

This interface is powerful in that the only requirement is that the *x*, *y*, *z*, and *w* coordinates of a control point are placed in successive memory locations. However, the control points themselves need not be contiguous. The data can be part of a larger data structure, or the points can form part of a larger array.

For example, suppose the data appears as follows:

```
struct ptdata {
               long tag1, tag2;  double x, y, z, w;
              }
points[4][4][3];
```

To use offsets, set *u_offset* to `4*3*sizeof(double)`, *v_offset* to `3*sizeof(double)`, and *ctlarray* to `&points[0][0][0]`.

14.4.1 Geometric Surfaces

To draw a geometric surface, call `nurbssurface()` between a `bgnsurface()/endsurface()` pair and specify the surface type as `N_V3D` or `N_V3DR`.

For example, to define a geometric surface for which you have specified the knot sequence as *knots* and the control points as *ctlpoints*, use:

```
double ctlpoints[4][4][3];
double knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};

/*
Initializes the control points of the surface to a small hill.
The control points range from -3 to +3 in x, y, and z
*/
void init_surface(void)
{
   int u, v;

   for (u = 0; u < 4; u++)
   {
      for (v = 0; v < 4; v++)
      {
         ctlpoints[u][v][0] = 2.0*((double)u - 1.5);
         ctlpoints[u][v][1] = 2.0*((double)v - 1.5);

         if ( (u == 1 || u == 2) && (v == 1 || v == 2))
            ctlpoints[u][v][2] = 3.0;
         else
            ctlpoints[u][v][2] = -3.0;
      }
   }
}
bgnsurface();
   nurbssurface(
      8, knots,
      8, knots,
      4*3*sizeof(double), 3*sizeof(double),
      &ctlpoints[0][0][0],
      4, 4,
      N_V3D);
   endsurface();
```

14.4.2 Color Surfaces

To specify a color surface, you use the same syntax as you do for a geometric surface, but you specify the type as `N_C4D` or `N_C4DR`. The color that you specify as a color surface has no effect when lighting is on (unless you have called `lmcolor()` with an appropriate argument), just as color commands have no effect on polygons when lighting is on. The color surface has no underlying geometry of its own; it is simply a property that is applied to the NURBS surface.

14.4.3 Texture Surfaces

To specify a texture surface, you use the same syntax as you do for a geometric surface, but you specify the type as `N_T2D` or `N_T2DR`. For texture mapping, the texture surface passed to the `nurbssurface()` call must contain an array of *s* and *t* texture coordinates that are associated with the texture surface.

The texture is applied to the NURBS surface, taking into account the current color defined by the current lighting model, in conjunction with the color defined by a color `nurbssurface()` call. You must call `texbind` to define a current texture for the texture coordinates to have any effect. For more information on defining textures, see Chapter 18, “Texture Mapping”.

Note: For color and texture surfaces, the number, spacing, or sequence of the control points is completely independent of the number, spacing, or sequence of the control points that define the geometric surface.

14.5 Trimming NURBS Surfaces

You can *trim* a NURBS surface to define its visible regions. The trimming information is mapped to model space along with the surface, as shown in Figure 14-13. A *trimming loop* defines the trim regions (shown in white) in the domain (shown in gray). Trimming loops can be composed of one or more *trimming curves.*

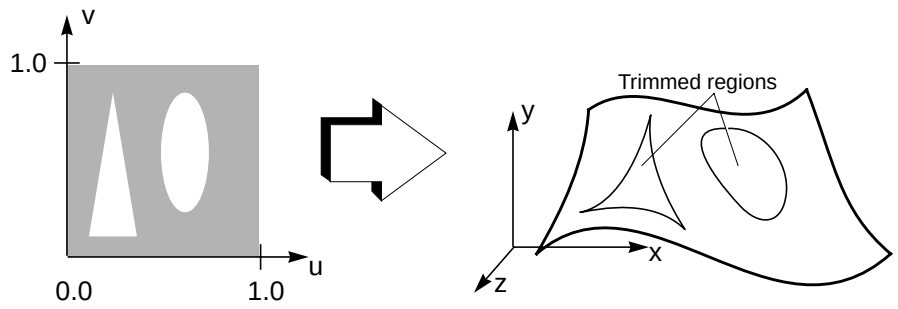

Figure 14-13 Trimmed Surface

Trimming curves, also called edges, are defined in the domain of the NURBS surface. Each edge has a head and a tail. The edge runs in the direction indicated starting at its tail and ending at its head.

You can define trimming curves with NURBS curves, using `nurbscurve()`, or with piecewise linear curves (a series of line segments), using `pwlcurve()`, or any combination of the two.

A trimming loop is a closed, oriented curve, defined in the domain, composed of edges that meet head-to-tail. Trimming loops are specified within the `bgnsurface()/endsurface()` block that defines the geometric surface.

Use the orientation of the trimming curve to indicate which regions of the surface to display:

- If the edges of the trimming curve run clockwise, the region of the NURBS surface that lies inside of the trimming curve *is not* displayed.
- If the edges of the trimming curve run counterclockwise, the region of the NURBS surface that lies inside of the trimming curve *is* displayed.

You can define a trimming loop with a single NURBS curve, or with a piecewise linear curve, or with a series of curves (of either type) joined head-to-tail.

Figure 14-14 shows some trimming curves and loops. If you nest trimming loops, as shown in Figure 14-14, you must specify the outer-most loop as counterclockwise.

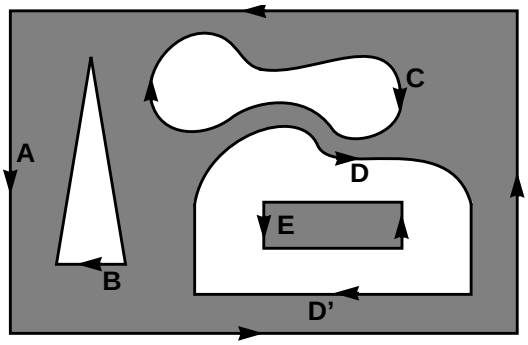

Figure 14-14 Trimming Curves

The following pseudocode creates the trimming curves shown in Figure 14-14:

```
bgnsurface();
   nurbssurface(...);
   bgntrim();
      pwlcurve(...); /* A */
   endtrim();
   bgntrim();
      pwlcurve(...); /* B */
   endtrim();
   bgntrim();
      nurbscurve(...); /* C */
   endtrim();
   bgntrim();
      nurbscurve(...); /* D */
      pwlcurve(...); /* D' */
   endtrim();
   bgntrim();
      pwlcurve(...); /* E */
   endtrim();
endsurface();
```

The following rules apply to trimming curves and trimming loops:

- Each trimming loop is surrounded by a `bgntrim()/endtrim()` pair.
- Each trimming loop must be closed; that is, the coordinates of the first and last points of the trimming curve must be identical (within a tolerance of 10^{-6}).
- Within a multi-segment trimming loop, the trimming curves must connect head-to-tail. This means that the last point of each curve segment must touch the first point of the next segment, and the last point of the last segment must touch the first point of the first segment.
- Trimming loops can neither touch nor intersect (except at their end points, which must touch).
- The outer loop of a nested trim sequence must be counter-clockwise.
- The trim space is the parameter space as defined by the knot sequence. For example, for a surface with identical u and v knot sequences (0,0,0,0,10,10,10,10), with no trimming information provided the trim region is effectively an isoparametric square with corners at (0,0) and (10,10).

To specify a NURBS trimming curve, use `nurbscurve()`, with `N_P2D` data for polynomial (*u*,*v*) curves or `N_P2DR` data for rational (*wu*,*wv*,*w*) curves.

To specify a piecewise linear trimming curve, use `pwlcurve()`:

```
void pwlcurve(long n, double *data, long offset, long type);
```

n	number of points in the trimming curve
data	array of points on the trimming curve
offset	offset in bytes between points in the array
type	type of curve, use N_ST

An offset allows the data points to be part of an array of larger structural elements. `pwlcurve()` searches for the nth coordinate pair beginning at data_array + n * offset.

The trim curve is drawn in the domain by connecting each point in the *data* array to the next point. If this `pwlcurve()` is the only curve forming a trimming loop, it is important to increment the trim point count as it is to duplicate the last point—in other words, although the last and first points are identical, they must be specified and counted twice.

14.6 NURBS Properties

NURBS properties control the rendering of NURBS curves and surfaces for the current window. You can set and query NURBS properties on a per-window basis. Use `setnurbsproperty()` to define the current NURBS properties. Use `getnurbsproperty()` to query the system for the current NURBS properties.

The ANSI C specifications for these two functions are:

```
void setnurbsproperty(long property, float value)
void getnurbsproperty(long property, float *value)
```

For maximum generality, express the value of a property in floating point. For some properties, only integer values make sense, but you must still pass them in floating point form.

Each property has a reasonable default value, but can be changed to affect the accuracy of some part of the rendering.

The NURBS properties are:

`N_PIXEL_TOLERANCE`
: A positive floating point value that bounds the maximum length (in pixels) of an edge of a line segment generated in the *tessellation* (breaking down into triangles) of a curve or a polygon generated in the tessellation of a surface.

`N_ERRORCHECKING`
: A Boolean value that, when TRUE, instructs the GL to send NURBS-related error messages to standard error.

`N_DISPLAY`
: An enumeration that dictates the surface rendering format; it affects surfaces only. Possible values are N_FILL, N_OUTLINE_POLY, and N_OUTLINE_PATCH. N_FILL instructs the GL to fill all polygons generated in the tessellation of the surface. N_OUTLINE_POLY instructs the GL to outline all polygons generated. N_OUTLINE_PATCH instructs the GL to outline the boundary of all surface patches and trim curves.

`N_CULLING`
: A Boolean value that, when TRUE, instructs the GL to discard before tessellation all patches that are outside the current viewport.

14.7 Sample NURBS Program

This sample program, *nurbs.c*, draws a lighted nurbs surface. Use the mouse to toggle the trimming curve on and off.

```
/*
 * nurbs.c
 */

#include <stdio.h>
#include <gl/gl.h>
#include <gl/device.h>

#define S_NUMKNOTS 8 /* number of knots in each dimension of surface*/
#define S_NUMCOORDS 3 /* number of surface coordinates, (x,y,z) */
#define S_ORDER 4 /* surface is bicubic, order 4 for each parameter */
#define T_NUMKNOTS 12 /* number of knots in the trimming curve */
#define T_NUMCOORDS 3 /* number of curve coordinates, (wx,wy,w) */
#define T_ORDER 3 /* trimming curve is rational quadratic */

/* number of control points in each dimension of NURBS */
#define S_NUMPOINTS (S_NUMKNOTS - S_ORDER)
#define T_NUMPOINTS (T_NUMKNOTS - T_ORDER)
/* trimming */
int trim_flag = 0;

long zfar;

double surfknots[S_NUMKNOTS] = {
    -1., -1., -1., -1., 1., 1., 1., 1.
    };

double ctlpoints[S_NUMPOINTS][S_NUMPOINTS * S_NUMCOORDS] = {
    -2.5, -3.7, 1.0,
    -1.5, -3.7, 3.0,
     1.5, -3.7, -2.5,
     2.5, -3.7, -.75,

    -2.5, -2.0, 3.0,
    -1.5, -2.0, 4.0,
     1.5, -2.0, -3.0,
     2.5, -2.0, 0.0,

    -2.5, 2.0, 1.0,
    -1.5, 2.0, 0.0,
     1.5, 2.0, -1.0,
```

```
     2.5, 2.0, 2.0,

    -2.5, 2.7, 1.25,
    -1.5, 2.7, .1,
     1.5, 2.7, -.6,
     2.5, 2.7, .2
    };

doubletrimknots[T_NUMKNOTS] = {
    0., 0., 0., 1., 1., 2., 2., 3., 3., 4., 4., 4.
    };

double trimpoints[T_NUMPOINTS][T_NUMCOORDS] = {
     1.0, 0.0, 1.0,
     1.0, 1.0, 1.0,
     0.0, 2.0, 2.0,
    -1.0, 1.0, 1.0,
    -1.0, 0.0, 1.0,
    -1.0, -1.0, 1.0,
     0.0, -2.0, 2.0,
     1.0, -1.0, 1.0,
     1.0, 0.0, 1.0,
    };

float idmat[4][4] = {
    1.0, 0.0, 0.0, 0.0,
    0.0, 1.0, 0.0, 0.0,
    0.0, 0.0, 1.0, 0.0,
    0.0, 0.0, 0.0, 1.0
    };

main()
{
 long dev;
 short val;

 init_windows();
 setup_queue();
 init_view();
 make_lights();

 set_scene();
 setnurbsproperty( N_ERRORCHECKING, 1.0 );
 setnurbsproperty( N_PIXEL_TOLERANCE, 50.0 );
 draw_trim_surface();
```

```
 while(TRUE) {
    while(qtest()) {
     dev=qread(&val);
     switch(dev) {
        case ESCKEY:
         gexit();
         exit(0);
         break;
        case WINQUIT:
         gexit();
         dglclose(-1); /* this for DGL only */
         exit(0);
         break;
        case REDRAW:
         reshapeviewport();
         set_scene();
         draw_trim_surface();
         break;
        case LEFTMOUSE:
         if( val )
            trim_flag = !trim_flag; /* trimming */
         break;
        default:
         break;
     }
    }
    set_scene();
    draw_trim_surface();
 }
}

init_windows()
/*-------------------------------------------------------------
 * Initialize all windows
 *-------------------------------------------------------------
 */
{
 if (getgdesc(GD_BITS_NORM_DBL_RED) <= 0) {
 fprintf(stderr, "nurbs: requires double buffered RGB which is "
 "unavailable on this machine\n");
 exit(1);
 /* NOTREACHED */
 }
 winopen("nurbs");
 wintitle("NURBS Surface");
 doublebuffer();
```

```
 RGBmode();
 gconfig();
 zbuffer( TRUE );
 glcompat(GLC_ZRANGEMAP, 0);
 zfar = getgdesc(GD_ZMAX);
}

setup_queue()
/*-------------------------------------------------------------
 * Queue all devices
 *-------------------------------------------------------------
 */
{
 qdevice(ESCKEY);
 qdevice(RIGHTMOUSE);
 qdevice(WINQUIT);
 qdevice(LEFTMOUSE);/* trimming */
}

init_view()
/*-------------------------------------------------------------
 * Initialize view and lighting mode
 *-------------------------------------------------------------
 */
{
 mmode(MPROJECTION);
 ortho( -4., 4., -4., 4., -4., 4. );

 mmode(MVIEWING);
 loadmatrix(idmat);

}

set_scene()
/*-------------------------------------------------------------
 * Clear screen and rotate object
 *-------------------------------------------------------------
 */
{
 lmbind(MATERIAL, 0);
 RGBcolor(150,150,150);
 lmbind(MATERIAL, 1);

 czclear(0x00969696, zfar);

 rotate( 100, 'y' );
```

```
 rotate( 100, 'z' );
}

draw_trim_surface()
/*--------------------------------------------------------------
 * Draw NURBS surface
 *--------------------------------------------------------------
 */
{
 bgnsurface();
    nurbssurface(
     sizeof( surfknots) / sizeof( double ), surfknots,
     sizeof( surfknots) / sizeof( double ), surfknots,
     sizeof(double) * S_NUMCOORDS,
     sizeof(double) * S_NUMPOINTS * S_NUMCOORDS,
     ctlpoints,
     S_ORDER, S_ORDER,
     N_V3D
     );
    /* trimming */
    if( trim_flag ) {
     bgntrim();
     /* trim curve is a rational quadratic nurbscurve (circle) */
 nurbscurve(
        sizeof( trimknots) / sizeof( double ),
        trimknots,
        sizeof(double) * T_NUMCOORDS,
        trimpoints,
        T_ORDER,
     N_P2DR
        );
     endtrim();
    }
 endsurface();
 swapbuffers();
}

make_lights()
/*--------------------------------------------------------------
 * Define material, light, and model
 *--------------------------------------------------------------
 */
{
 /* initialize model and light to default */
 lmdef(DEFLMODEL,1,0,0);
 lmdef(DEFLIGHT,1,0,0);
```

```
 /* define material #1 */
 {
 static float array[] = {
        EMISSION, 0.0, 0.0, 0.0,
        AMBIENT, 0.1, 0.1, 0.1,
        DIFFUSE, 0.6, 0.3, 0.3,
        SPECULAR, 0.0, 0.6, 0.0,
        SHININESS, 2.0,
        LMNULL
 };
 lmdef(DEFMATERIAL, 1, sizeof(array)/sizeof(array[0]), array);
 }

 /* turn on lighting */
 lmbind(LIGHT0, 1);
 lmbind(LMODEL, 1);
 lmbind(MATERIAL, 1);
}
```

Suggestions for Further Reading

For a basic introduction to curves and surfaces:

Foley, J.D., A. van Dam, S. Feiner, and J.D. Hughes, "Representing Curves and Surfaces," in *Computer Graphics Principles and Practice*, 2d ed., Addison Wesley Publishing Company Inc., Menlo Park, 1990.

For a comprehensive treatment, including derivations:

Farin, G., *Curves and Surfaces for Computer Aided Geometric Design*, Academic Press, 2d ed., New York, 1990.

14.8 Old-Style Curves and Surfaces

This section describes the methods used for drawing curves and surfaces prior to the 4D1-3.2 release of the Graphics Library software. These techniques are still supported for compatibility with programs written for earlier versions of the software.

Note: Use of these statements is not recommended. All new development should use the GL NURBS method.

The techniques presented in this section have limited capabilities. Old-style curves are restricted to *cubic splines* (degree 3, order 4) only. Each polynomial segment is presented to the drawing subroutines one at a time.

Old-style surfaces are restricted to *bicubic wireframe* surfaces only, and each surface patch is presented to the drawing subroutines one at a time.

14.8.1 Old-Style Curves

The curves in most applications are too complex to be represented by a single curve segment and instead must be represented by a series of curve segments joined end-to-end. To create smooth joints, you must control the positions and curvatures at the endpoints of curve segments.

The shape of the curve segment is determined by a function of a set of four control points. The IRIS approximates the shape of a curve segment with a series of straight line segments.

A set of constraints expresses how the shape of the curve segment relates to the control points. For example, a constraint might be that one endpoint of the curve segment is located at the first control point, or that the tangent vector at an endpoint lies on the line segment formed by the first two control points. This constraint information is used to define a *basis matrix* for the curve.

The old-style IRIS curve facility is based on the *parametric cubic curve*. Three classes of cubic curves are available:

- Bezier
- Cardinal spline
- B-spline

The characteristics of each of these curves are summarized next.

Bezier Cubic Curve

A *Bezier c*ubic curve segment passes through the first and fourth control points and uses the second and third points to determine the shape of the curve segment. Of the three kinds of curves, the Bezier form provides the most intuitive control over the shape of the curve.

Cardinal Spline Cubic Curve

A *Cardinal spline* curve segment passes through the two interior control points and is continuous in the first derivative at the points where segments meet. The curve segment starts at the second point and ends at the third point, and uses the first and fourth points to define the shape of the curve. The mathematical derivation of the Cardinal spline basis matrix can be found in James H. Clark, *Parametric Curves, Surfaces and Volumes in Computer Graphics and Computer-Aided Geometric Design*, Technical Report No. 221, Computer Systems Laboratory, Stanford University.

B-Spline Cubic Curve

In general, a *B-spline* curve segment does not pass through any control points, but is continuous in both the first and second derivatives at the points where segments meet. Thus, a series of joined B-spline curve segments is smoother than a series of Cardinal spline segments.

14.8.2 Drawing Old-Style Curves

You can create complex curved lines by joining several curve segments end-to-end. The curve facility provides the means for making smooth joints between the segments.

You draw an old-style curve segment by specifying:

- a set of four control points
- a basis, which defines how the system uses the control points to determine the shape of the segment

To draw an old-style curve segment:

1. Define and name a basis matrix with `defbasis()`:

   ```
   void defbasis(short id, Matrix mat);
   ```

   The matrix *mat* is saved and is associated with the identifier *id*. Use *id* in subsequent calls to `curvebasis()` and `patchbasis()`.

2. Select a defined basis matrix (defined by `defbasis()`) as the current basis matrix with `curvebasis()`:

   ```
   void curvebasis(short basisid);
   ```

3. Specify the number of line segments used to approximate each curve segment with `curveprecision()`:

   ```
   void curveprecision(short nsegments);
   ```

   When `crv()`, `crvn()`, `rcrv()`, or `rcrvn()` executes, a number of straight line segments (*nsegments*) approximates each curve segment. The greater the value of *nsegments*, the smoother the curve, but the longer the drawing time.

4. Draw the curve segment using the current basis matrix, the current curve precision, and the four control points with `crv()` (`rcrv()` draws a rational curve).

   ```
   void crv(Coord geom[4][3]);
   ```

When you call `crv()`, a matrix M is built from the geometry G (the control point array), the current basis M_b, and the current precision n, as shown below:

$$\begin{bmatrix} \frac{6}{n^3} & 0 & 0 & 0 \\ \frac{6}{n^3} & \frac{2}{n^2} & 0 & 0 \\ \frac{1}{n^3} & \frac{1}{n^2} & \frac{1}{n} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} M_b G$$

The bottom row of the resulting transformation matrix identifies the first of n points that describe the curve.

A forward difference algorithm is used to iterate the matrix to generate the remaining points in the curve. Each iteration draws one line segment of the curve segment. At each iteration, the first row is added to the second row, the second row is added to the third row, and the third row is added to the fourth row. The fourth row is then output as one of the points on the curve.

```
/* This is the forward difference algorithm */
/* M is the current transformation matrix */
move (M[3][0]/M[3][3], M[3][1]/M[3][3], M[3][2]/M[3][3]);
/* iteration loop */
for (cnt = 0; cnt < iterationcount; cnt++) {
   for (i=3; i>0; i--)
      for (j=0; j<4; j++)
         [j] = M[i][j] + M[i-1][j];
   draw(M[3][0]/M[3][3], M[3][1]/M[3][3], M[3][2]/M[3][3]);
}
```

Each iteration draws one line segment of the curve. For the precision matrix shown, these points are generated:

$$(0, 0, 0, 1), \left(\left(\frac{1}{n}\right)^3, \left(\frac{2}{n}\right)^2, \frac{1}{n}, 1\right), \left(\left(\frac{2}{n}\right)^3, \left(\frac{2}{n}\right)^2, \frac{2}{n}, 1\right), \left(\left(\frac{3}{n}\right)^3, \left(\frac{3}{n}\right)^2, \frac{3}{n}, 1\right), \ldots$$

The iteration loop of the forward difference algorithm is implemented in the Geometry Pipeline. You can use `curveit()` to access the forward difference algorithm, making it possible to generate a curve directly from a forward difference matrix:

```
void curveit(short iterationcount);
```

`curveit()` iterates the current matrix (the one on top of the matrix stack) *iterationcount* times. Each iteration draws one of the line segments that approximate the curve. `curveit()` does not execute the initial move in the forward difference algorithm. A `move(0.0,0.0,0.0)` call must precede `curveit()` so that the correct first point is generated from the forward difference matrix.

Note: To get the numbers to use for the basis matrix M_b, see the sample program *patch1.c* at the end of the next section.

Drawing a Series of Curve Segments

You use `crvn()` to draw a series of cubic spline segments using the current basis, precision, and a series of *n* control points. Calling `crvn()` has the same effect as programming a sequence of `crv()` calls with overlapping control points.

```
void crvn(long n, Coord geom[][3]);
```

The control points specified in *geom* determine the shapes of the curve segments and are used four at a time. If the current basis is a B-spline, Cardinal spline, or a basis with similar properties to these types of curves, the curve segments are joined end-to-end and appear as a single curve.

When you call `crvn()` with a Cardinal spline or B-spline basis, it produces a single curve. However, calling `crvn()` with a Bezier basis produces several separate curve segments. As with `crv()` and `rcrv()`, a precision and basis must be defined before calling `crvn()` or `rcrvn`. This is true even if the routines are compiled into objects. See Chapter 16 for more information on graphical objects.

Rational Curves

The IRIS graphics hardware actually works in homogeneous coordinates *x*, *y*, *z*, and *w*, where 3-D coordinates are given by *xw*, *yw*, and *zw*. The homogeneous character of the system is not obvious because *w* is normally the constant 1.

The *w* coordinate can also be expressed as a cubic function of the parameter *t*, so that the 3-D coordinates of points along the curve are given as a quotient of two cubic polynomials. The only constraint is that the denominator for all three coordinates must be the same. When *w* is not the constant 1, but some cubic polynomial function of *t*, the curves generated are usually called *rational cubic splines*.

The basis definitions for rational cubic splines are identical to those for cubic splines, as are the precision specifications. The only difference is that the geometry matrix must be specified in four-dimensional homogeneous coordinates.

Use `rcrv()` to draw rational curves:

```
void rcrv(Coord geom[4] [4]);
```

`rcrv()` draws a rational curve segment using the current basis matrix, the current curve precision, and the four control points specified in its argument. `rcrv()` is analogous to `crv()` except that *w* coordinates are included in the control point definitions.

`rcrvn()` takes a series of *n* control points and draws a series of rational cubic spline curve segments using the current basis and precision:

```
void rcrvn(long n, Coord geom[][4]);
```

The control points specified in *geom* determine the shapes of the curve segments and are used four at a time.

14.8.3 Drawing Old-Style Surfaces

A *surface patch* appears on the screen as a wireframe of curve segments. A set of user-defined control points determines the shape of the patch. You can create complex surfaces by joining several patches into one large patch. You can also create a complex surface consisting of several joined patches by using overlapping sets of control points and the B-spline and Cardinal spline curve bases.

The method for drawing old-style surfaces is similar to that for drawing curves. You draw an old-style wireframe surface patch by specifying:

- a set of 16 control points
- the number of curve segments to be drawn in each direction of the patch
- the two bases that define how the control points determine the shape of the patch

The IRIS old-style surface facility is based on the *parametric bicubic surface.* Bicubic surfaces can provide continuity of position, slope, and curvature at the points where two patches meet.

The points on a bicubic patch are defined by varying the parameters *u* and *v* from 0 to 1. If one parameter is held constant and the other is varied from 0 to 1, the result is a cubic curve. Thus, you can create a wireframe patch by holding *u* constant at several values and using the IRIS curve facility to draw curve segments in one direction, and then doing the same for *v* in the other direction.

To draw a surface patch:

1. Define the appropriate curve bases using `defbasis()`. A Bezier basis provides intuitive control over the shape of the patch. The Cardinal spline and B-spline bases allow smooth joints to be created between patches.
2. Specify a basis matrix (defined by `defbasis()`) for the *u* and *v* parametric directions of a surface patch with `patchbasis()`:

   ```
   void patchbasis(long uid, long vid);
   ```

   The *u* basis and the *v* basis do not have to be the same. `patch()` uses the current *u* and *v* when it executes.
3. Specify the number of curve segments to be drawn in each direction with patchcurves:

   ```
   void patchcurves(long ucurves, long vcurves);
   ```

   A different number of curve segments can be drawn in each direction.
4. Specify the precisions for the curve segments in each direction with `patchprecision()`:

   ```
   void patchprecision(long usegments, long vsegments);
   ```

   The precision is the minimum number of line segments approximating each curve segment and can be different for each direction. The actual number of line segments is a multiple of the number of curve segments being drawn in the opposing direction. This guarantees that the *u* and *v* curve segments that form the wireframe actually intersect.
5. Draw the surfaces with patch or `rpatch()`:

   ```
   void patch(Matrix geomx, Matrix geomy, Matrix geomz);
   void rpatch(Matrix geomx, Matrix geomy, Matrix geomz, Matrix geomw);
   ```

`rpatch()` is the same as patch, except it draws a rational surface patch. The curve segments in the patch are drawn using the current linestyle, linewidth, color, and writemask.

The arguments *geomx*, *geomy*, and *geomz* are 4x4 matrices containing the coordinates of the 16 control points that determine the shape of the patch. *geomw* specifies the rational component of the patch to `rpatch()`.

The sample program *patch1.c* shows the number to use for each type of basis matrix discussed and it uses them to draw a surface patch.

```
#include <gl/gl.h>

#define X       0
#define Y       1
#define Z       2
#define XYZ     3

#define BEZIER   1
#define CARDINAL 2
#define BSPLINE  3

Matrix beziermatrix = {
   {-1.0, 3.0, -3.0, 1.0},
   { 3.0, -6.0, 3.0, 0.0},
   {-3.0, 3.0, 0.0, 0.0},
   { 1.0, 0.0, 0.0, 0.0}
};

Matrix cardinalmatrix = {
   {-0.5, 1.5, -1.5, 0.5},
   { 1.0, -2.5, 2.0, -0.5},
   {-0.5, 0.0, 0.5, 0.0},
   { 0.0, 1.0, 0.0, 0.0}
};

Matrix bsplinematrix = {
   {-1.0/6.0, 3.0/6.0, -3.0/6.0, 1.0/6.0},
   { 3.0/6.0, -6.0/6.0, 3.0/6.0, 0.0},
   {-3.0/6.0, 0.0, 3.0/6.0, 0.0},
   { 1.0/6.0, 4.0/6.0, 1.0/6.0, 0.0}
};

Matrix geom[XYZ] = {
   { { 0.0, 100.0, 200.0, 300.0},
   { 0.0, 100.0, 200.0, 300.0},
   {700.0, 600.0, 500.0, 400.0},
   {700.0, 600.0, 500.0, 400.0}, },

   { {400.0, 500.0, 600.0, 700.0},
   { 0.0, 100.0, 200.0, 300.0},
   { 0.0, 100.0, 200.0, 300.0},
   {400.0, 500.0, 600.0, 700.0}, },
```

```
   { {100.0, 200.0, 300.0, 400.0},
   {100.0, 200.0, 300.0, 400.0},
   {100.0, 200.0, 300.0, 400.0},
   {100.0, 200.0, 300.0, 400.0}, },
};

main()
{
   int i, j;

   prefsize(400, 400);
   winopen("patch1");
   color(BLACK);
   clear();
   ortho(-100.0, 800.0, -100.0, 800.0, -800.0, 100.0);

 /* define 3 types of bases */
 defbasis(BEZIER, beziermatrix);
 defbasis(CARDINAL, cardinalmatrix);
 defbasis(BSPLINE, bsplinematrix);

 /*
 * seven curve segments will be drawn in the u direction
 *  and four in the v direction
 */
 patchcurves(4, 7);

 /*
 * the curve segments in u direction will consist of 20 line
segments
 * (the lowest multiple of vcurves greater than usegments) and
the curve
 * segments in the v direction will consist of 21 line
segments (the
 * lowest multiple of ucurves greater than vsegments)
 */
 patchprecision(20, 20);

 /* the patch is drawn based on the sixteen specified control
points */
   patchbasis(BEZIER, BEZIER);
   color(RED);
   patch(geom[X], geom[Y], geom[Z]);

 /*
```

```
 * another patch is drawn using the same control points but a
different
 * basis
 */
   patchbasis(CARDINAL, CARDINAL);
   color(GREEN);
   patch(geom[X], geom[Y], geom[Z]);

 /* a third patch is drawn */
   patchbasis(BSPLINE, BSPLINE);
   color(BLUE);
   patch(geom[X], geom[Y], geom[Z]);

 /* show the control points */
   color(WHITE);
   for (i = 0; i < 4; i++) {
       for (j = 0; j < 4; j++) {
           pushmatrix();
           translate(geom[X][i][j], geom[Y][i][j],
geom[Z][i][j]);
           circf(0.0, 0.0, 3.0);
       popmatrix();
       }
   }

 sleep(10);
 gexit();
 return 0;
}
```