An Introduction to NURBS and OpenGL

C. Ramakrishnan
MAT 200C
Winter ‘02

NURBS Can Be Cool, Once You Get To Know Them

Introduction

Non-Uniform Rational B-Splines, or NURBS as they are known to their friends, are a powerful way to specify complex curves and surfaces using a small number of control points. Following the exposition of Jones (2001), this article develops the *background* for understanding of NURBS. We briefly consider NURBS, but we will not develop an appreciation for their full power: we just focus on how to express simpler structures using NURBS.

Then we look at how to render Bézier curves and NURBS using OpenGL. We will generate visual representations, but the reader should keep in mind that NURBS need not be given visual representations — they are potentially applicable to, say, sound. In particular, NURBS are brilliant solutions to what I believe to be one of the fundamental problems of algorithmic generation of art: the problem of generating complex structures from a small number of control data. Thus, NURBS and especially Bézier curves are instructive examples even for those working outside the visual arts.

Bézier Splines

Square Interpolation

To properly understand NURBS, we must first look at their simpler cousin, the Bézier curve.

Suppose we have two points, P and S, and we want to construct a curve between them. The simplest solution is to construct the parametric line segment between the two points through simple linear interpolation:

$$P(u) = (1-u)P + uS; 0 \leq u \leq 1$$

(Audio people can think of this as just a linear cross-fade.)

The line segment we have is a start, but we want something more complex: we want a curve. With just two points, the problem is underdetermined. There are many ways to draw a curve between two points and there is no reason to pick one curve over another.

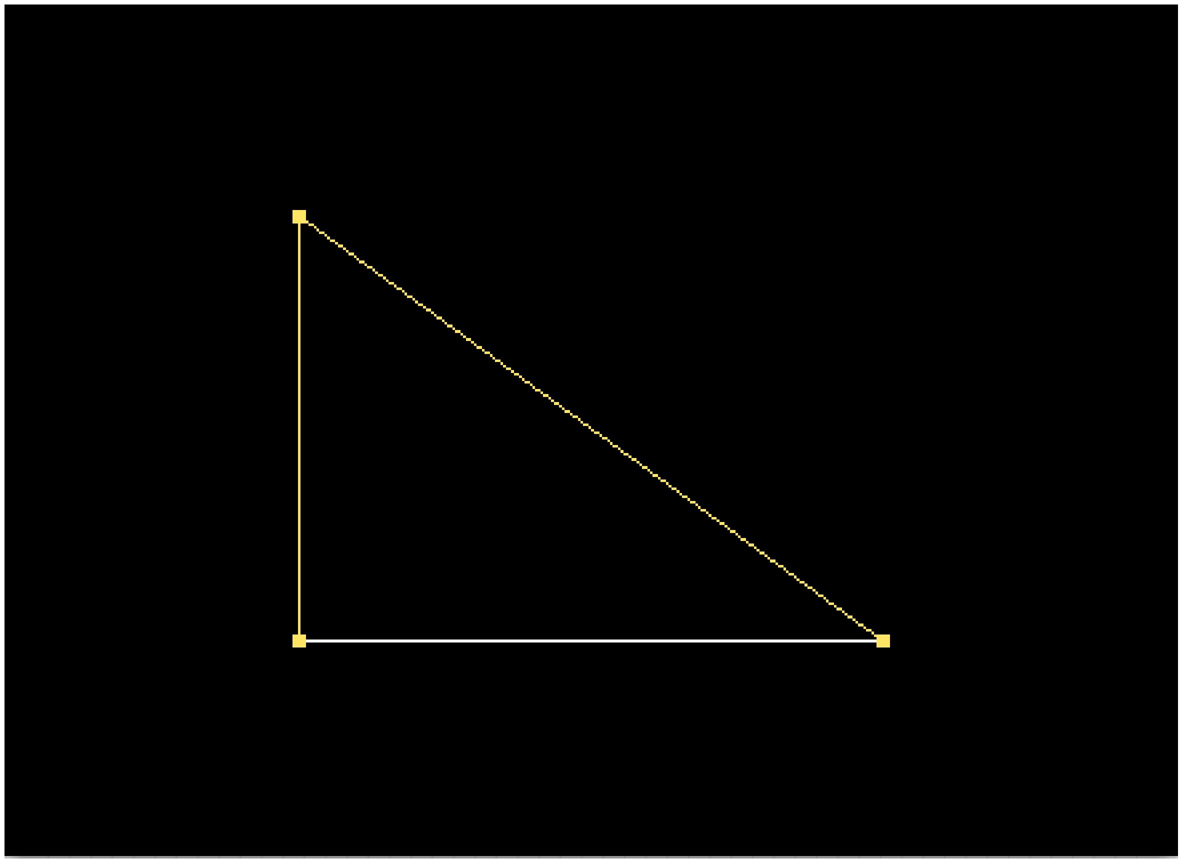
To better define the curve, we can add another control point, Q. There are still many ways to use Q to determine one curve of the many between P and S, but, let us continue with our linear interpolation approach. First, we interpolate between P and Q:

$$P(u) = (1-u)P + uQ; 0 \leq u \leq 1$$

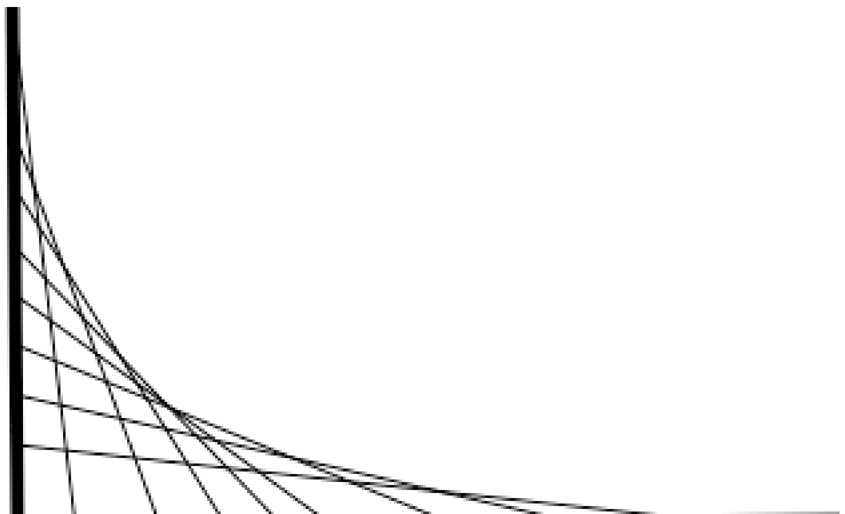
But we want to go from P to S and this only gets us part-way there. So, we interpolate between Q and S:

$$Q(u) = (1-u)Q + uS; 0 \leq u \leq 1$$

This gives us two lines (the yellow ones):



Now we are at the point where the magic happens. If you grew up in the United States, in 4th grade you quite likely made art by looping string around pins to make curves using only straight lines (children in other countries may do this too). It looks like this:



We are going to do something similar. Between pairs of points on the two lines we have defined, we are going to interpolate lines. This gives us:

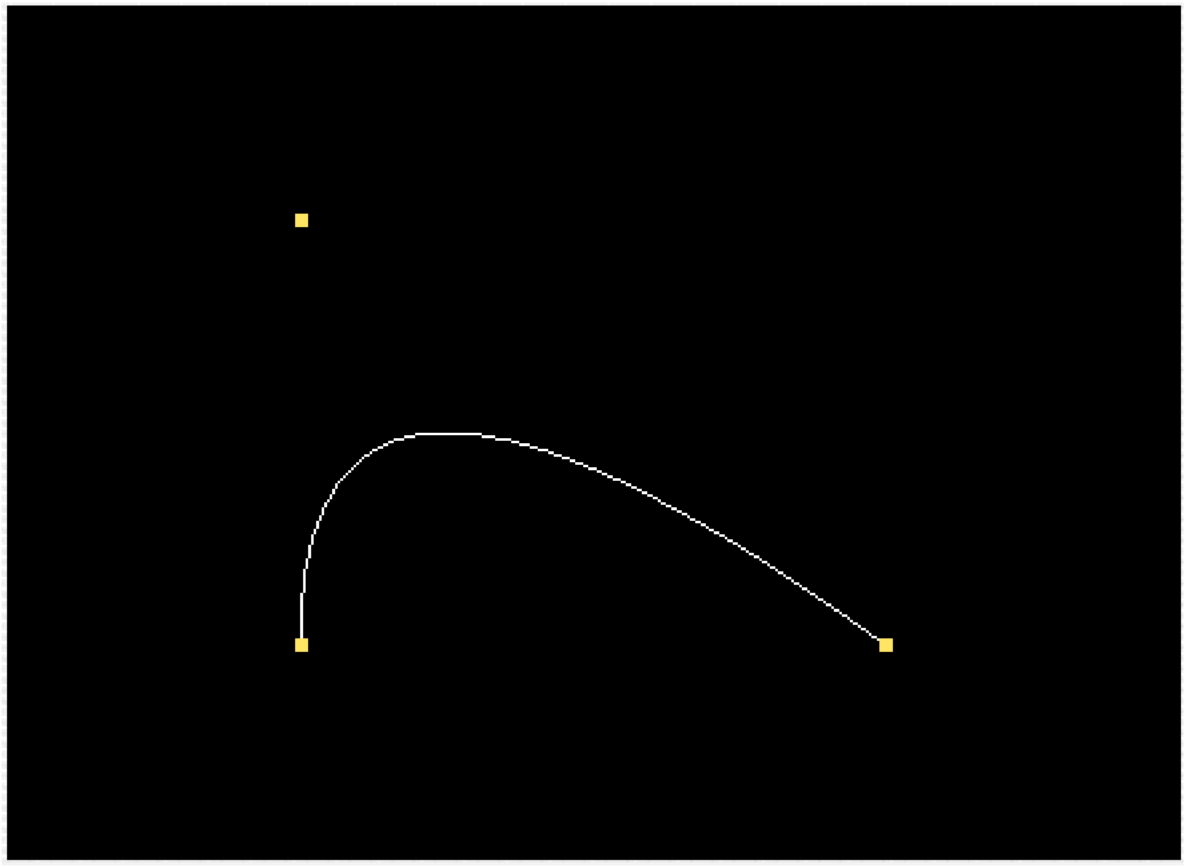
$$P'(t) = (1-t)P(u) + tQ(u); 0 <= t <= 1$$

This is a collection of lines (one for each u). But, by combining the three equations and using the same “u” in all three:

$$P(u) = (1-u)P + uQ$$
$$Q(u) = (1-u)Q + uS$$

$$P'(u) = (1-u)P(u) + uQ(u) \quad (1)$$

we get a collection of points. This collection of points defines a curve.



So, we have just defined a curve using only lines!

Note, that by substituting the equations P(u) and Q(u) into eq. 1 for P'(u), we get an equation for P'(u) in terms of P, Q, and S:

$$P'(u) = (1-u)^2P + 2u(1-u)Q + u^2S;\ 0 \le u \le 1$$

This curve does indeed go through P and S. You can see this by setting u=0, which gives us the point P. And setting u=1 gives us the point S. In general, this curve *does not* go through our control point Q. (Other than the trivial case where P, Q, and S are colinear, does the curve we have defined ever go through Q?)

Cubic Interpolation

We started off with two points, P and S, and a line between them. By adding one point, the control point Q, we were able to get a for more interesting curve that connected P and S. Continuing what we have been doing, let us add another point of control, R. By using our same interpolating strategy as before, we get the equations for three lines:

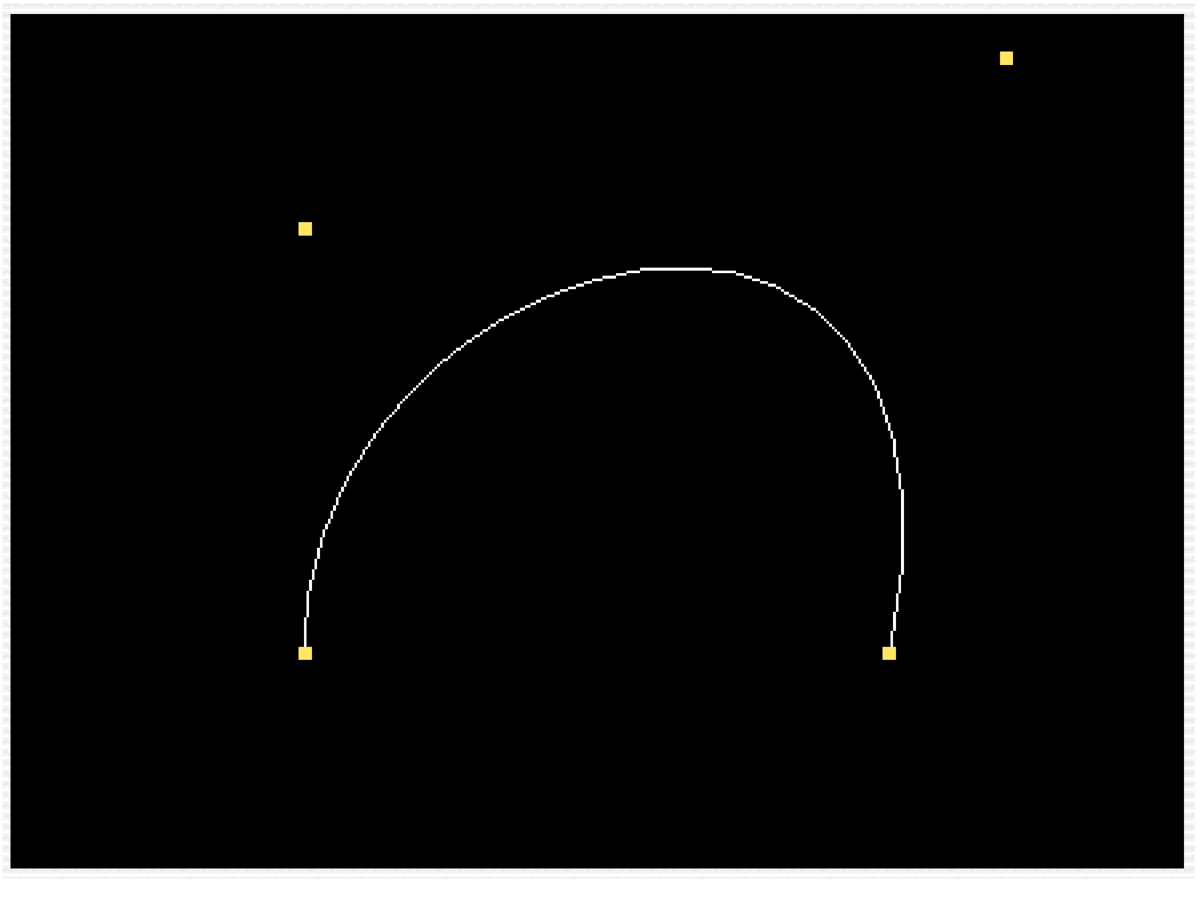
$$\begin{aligned} P(u) &= (1-u)P + uQ \\ Q(u) &= (1-u)Q + uR \\ R(u) &= (1-u)R + uS \end{aligned}$$

We also get two curves, one from P to R, and one from Q to S:

$$\begin{aligned} P'(u) &= (1-u)P(u) + uQ(u) \\ Q'(u) &= (1-u)Q(u) + uR(u) \end{aligned}$$

Now, we interpolate between these two curves, to get a final curve:

$$P''(u) = (1-u)P'(u) + uQ'(u);\ 0 \le u \le 1$$



By adding another point, we get greater control over the shape of the curve. For example, it looks like it is impossible to create a curve that crosses itself using just two endpoints (P, S) and one control point (Q) (proof?). With two control points, it is possible to create a curve that crosses itself once, but no more (proof?).

We can convert this recursive algorithm into an explicit formula for the curve (in terms of P, Q, R, and S). First, substitute P(u), Q(u), and R(u) into the expressions for P'(u) and Q'(u) to yield:

$$P'(u) = (1-u)^2P + 2u(1-u)Q + u^2R$$
$$Q'(u) = (1-u)^2Q + 2u(1-u)R + u^2S$$

Then substituting these expressions for P' and Q' into P'' gives:

$$P''(u) = (1-u)^3P + 3u(1-u)^2Q + 3u^2(1-u)R + u^3S$$

This is the cubic Bézier Spline, the most commonly used Bézier Curve.

When seen in this explicit form, you can see that the curve is a sum of each control point (P, Q, R, S) multiplied by a blending function (C_0, C_1, C_2, C_3):

$$P''(u) = C_0(u)*P + C_1(u)*Q + C_2(u)*R + C_3(u)*S, \text{ where}$$
$$C_0(u) = (1-u)^3$$
$$C_1(u) = 3u(1-u)^2$$
$$C_2(u) = 3u^2(1-u)$$
$$C_3(u) = u^3$$

These blending functions have a name: the Bernstein Polynomials. This will come up again when we get to OpenGL.

We could continue to higher order Bézier splines, but there isn't any need to because you probably get the idea and see how to generalize. The basic trick is that of recursive interpolation -- interpolate between the control points, then interpolate between those line segments, and so on until you run out of things to interpolate between. If you are having problems understanding, or would like to know about the properties of derivatives of the Bézier spline and how to stitch cubic Bézier curves together, see Jones (2001).

Interlude — History

Bézier curves were formulated at around the same time, by Pierre Bézier and Paul de Faget de Casteljau (Jones, 2001, p. 259). Bézier created the formulation using blending functions; Casteljau devised the recursive interpolation algorithm. As shown above, these are equivalent.

Bézier worked at Renault and Casteljau at Citröen where they were developing early systems for computer aided design of automobiles (Gladden). For entertainment, you should check out these sites on Renault and Citröen cars. Take a look at the cars from the late 1960's and early 1970's to see what was being done with Bézier curves:

Renault: http://www.users.wineasy.se/katriina/history.htm
Citröen: http://web.ukonline.co.uk/Members/jr.marsh/cars.htm

From this point on, we are going to move a bit more quickly through the mathematics, so that we can connect it OpenGL. We cover the concepts with enough detail that you should have an understanding of how to use Bézier Surfaces, B-Splines, and NURBS in OpenGL. Once you know the relevant OpenGL functions, you will be able to play with these mathematical objects. It will probably be easier for you to develop your understanding in an environment where you can manipulate them.

Bézier Surfaces

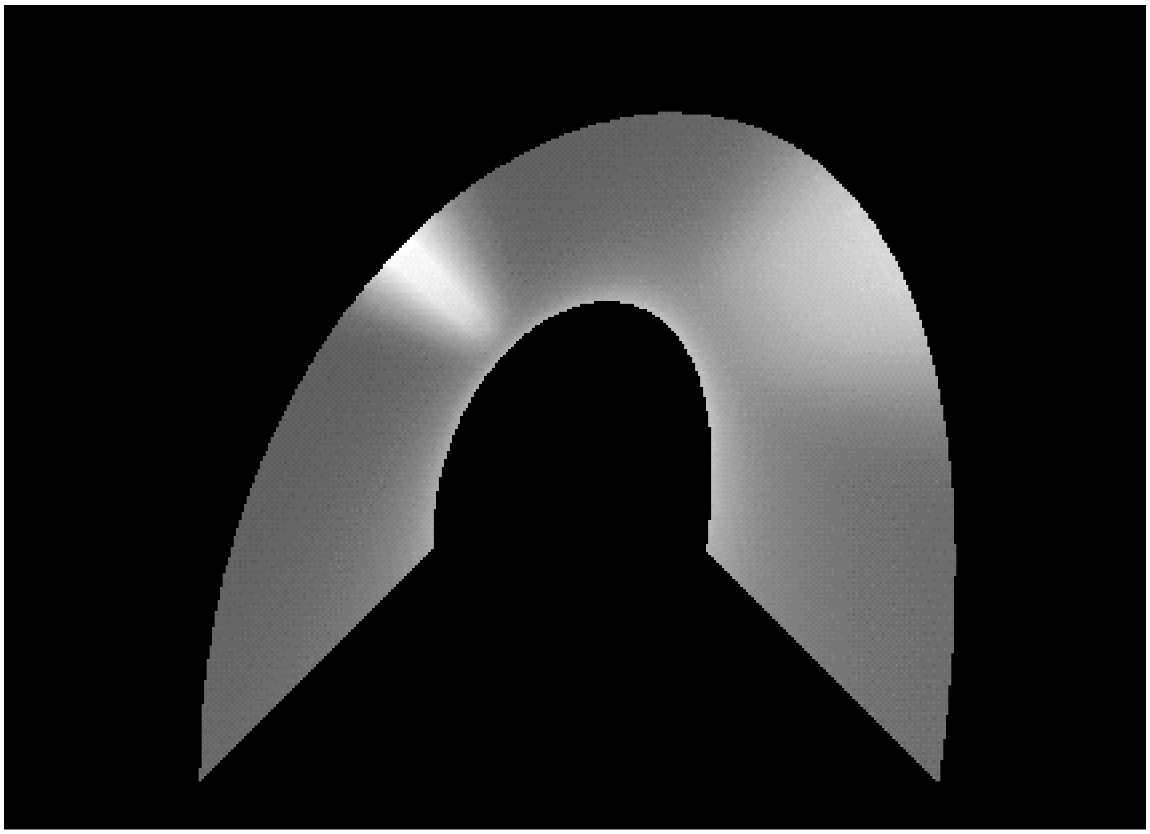
Bézier curves can be easily adapted to generating surfaces. The trick to doing this is to make the control points themselves lie on Bézier curves. In the bi-cubic case (where both of the Bézier curves are cubic):

$$S(u, v) = C_0(u)*P(v) + C_1(u)*Q(v) + C_2(u)*R(v) + C_3(u)*S(v);\quad 0 \le u \le 1,\ 0 \le v \le 1$$

where C_0 are the Bernstein Polynomial blending functions defined above, and P(v), Q(v), R(v), S(v) are themselves cubic Bézier curves:

$$\begin{aligned}
P(v) &= C_0(v)*P_0 + C_1(v)*Q_0 + C_2(v)*R_0 + C_3(v)*S_0\\
Q(v) &= C_0(v)*P_1 + C_1(v)*Q_1 + C_2(v)*R_1 + C_3(v)*S_1\\
R(v) &= C_0(v)*P_2 + C_1(v)*Q_2 + C_2(v)*R_2 + C_3(v)*S_2\\
S(v) &= C_0(v)*P_3 + C_1(v)*Q_3 + C_2(v)*R_3 + C_3(v)*S_3
\end{aligned}$$

This defines a surface based on the 4 Bézier curves, which are each defined by 4 control points. So, the surface is defined by 4*4 = 16 control points.



This is the important insight here: just as you can generate a curve by interpolating between points, you can generate a surface by interpolating between Bézier curves. And to do so, the number of control points equals the number of control points in the Bézier curves (which are all assumed to have the same number of control points) times the number of curves.

B-Splines

Bézier curves and surfaces are useful and powerful, but one potential problem with them is that the control points have global scope. Thus, a change in one control points effects the global shape of the curve, which could be undesirable.

One way around this is to stitch together multiple Bézier curves together. This works, assuming we respect some constraints in order to ensure that the resulting curve is smooth (i.e., differentiable). Another way to do it is with B-Splines, which is just a more formal way of doing the same thing (proof?).

The "B" in B-Spline stands for basis. In this context, a basis is just a blending function, as discussed above, by another name. However, there is a little difference between the blending functions used in B-Splines and those used in Bézier curves. In B-Splines, the blending function can be zero outside a particular range. The effect of defining such a range is that it limits the scope over which a control point has influence.

Thus, a B-Spline is defined by control points *and* the range in which each control point is active. These ranges are specified, indirectly, through something called a knot vector (I don’t know why it is called a knot vector). The knot vector is a non-decreasing sequence of numbers between 0 and 1.

$$\text{Knot Vector} = \{u_0, ..., u_m \mid \text{for all } i \; 0 \le u_i \le 1 \text{ and } u_{i-1} \le u_i\}$$

Given a knot vector with “m” knots as above, and assuming we have defined “n” control points (P_0, ..., P_n), let us define the degree of the curve as p = m - n - 1. Then the B-Spline, B, is defined as follows:

$$B(u) = \sum_{i=0}^{n} f_i^{(p)}(u)*P_i$$

where $f_i^{(p)}(u)$ is defined recursively as:

$$f_i^{(p)}(u) = \frac{u - u_i}{u_{i+p} - u_i} f_i^{(p-1)}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} f_{i+1}^{(p-1)}(u)$$

$$f_i^{(0)}(u) = \begin{cases} 1 \text{ for } u_i \le u \le u_{i+1} \text{ and } u_i < u_{i+1} \\ 0 \text{ otherwise} \end{cases}$$

The greater local control of B-Splines comes at the cost of greater complexity. Bézier curves are defined by only the control points. B-splines are defined by control points and a knot vector. This is necessary because the knot vector is used to define the domain of influence of each control point. (Actually, in both the Bézier curve and B-Spline, there is an implict power for the curve, though this is determined by the number of control points (in Bézier curves) or by the number of control points and the size of the knot vector (in B-Splines).) You should take into consideration that B-Splines require more information specified when deciding whether to use B-Splines or Bézier curves.

Three facts you’ll want to know about B-Splines:

— The power of the curve (p), the number of knots (m) and the number of control points (n) are related by the formula: m = n + p + 1.

— In general, a B-Spline will not pass through the two end control points. The curve will pass through these points if the first p+1 knots are 0 and the last p+1 knots are 1 (where p is the power of the curve).

— A Bézier curve is a B-Spline with the first p+1 knots = 0, the last p+1 knots = 1 and no other knots. This means that n, the number of control points, has to equal p + 1.

Did you get all that!? I'm guessing it looks a little abstruse. A good way to get a better handle on this material is to define B-Splines in OpenGL and then play around with the definition. Try keeping the control points fixed and tweaking the knot vector, for example.

NURBS

But wait... there's more. NURBS are generalizations of B-Splines that are invariant under the projective transformation. This is nice because it means that if you draw two objects that touch, they will still touch when drawn in perspective. NURBS also can *exactly* model conic sections.

NURBS are just B-Splines with modifications made to accommodate points specified using homogeneous coordinates. Without getting into a tangent about homogeneous coordinates, let's just say that homogeneous coordinates exist to make the projective transformation easier to work with.

NURBS, like B-Splines, are defined by some control points and a knot vector, the difference being the control points are specified in homogeneous coordinates.

If interested, the equations for NURBS can be found here:

<http://mathworld.wolfram.com/NURBSCurve.html>
<http://mathworld.wolfram.com/HomogeneousCoordinates.html>

Ok, let us get on to OpenGL!

NURBS and OpenGL (with a bias to OS X)

Introduction

This is not intended to be an intro to OpenGL, or even an intro to OpenGL on OS X, but I think it is worthwhile to provide a little background.

OpenGL, as you likely know, is a 2D and 3D graphics API originaly developed by Silicon Graphics. There are many great online references on OpenGL, and the *OpenGL Programming Guide* is an excellent paper reference.

On OS X, the easiest way to get working with OpenGL is to subclass the NSOpenGLView. Drop your subclass into your window, and you are ready to start going.

Some potentially-debugging-time-saving tips are:

— Some properties that are generally set programmatically in OpenGL can be set in InterfaceBuilder by looking at the properties of your view. For example, you can set the depth buffer (e.g., is your image 3D) in InterfaceBuilder

— You can get man pages on OpenGL functions in Terminal.app by doing “man function-name”. But you need to know that the function name is not usually what you type in your program! In your program you will generally type a function name+the type of arguments you are passing in, e.g., glMap1f (the “f” means that the arguments are floats). To get the man pages, though, you should type “man glMap1”.

— Shading of objects in OpenGL is done on a per-vertex basis. So, if you have defined surfaces that are equivalent, but have a different number of vertices, they will be shaded differently (see Kilgard).

The provided (OS X) sample code handles all the necessary setup code, so you can play around with it, even if you do have much background in OpenGL.

Evaluators

There are two ways to draw Bézier curves in OpenGL. One is to use “evaluators”, which is the more primitive option. The other is to use the NURBS renderer, which is part of the OpenGL Utilities library of functions (prefixed with “glu”). First, we will look at evaluators.

In this discussion, we will just talk about curves, not surfaces, but everything generalizes in a straightforward manner. You can also look at the code examples for surfaces.

The part of the OpenGL API we are concerned with is made up of 4 functions: glMap1, glEvalCoord, glMapGrid, glEvalMesh.

glMap1 is used to define a one dimensional evaluator. An example from the provided source is:

```
        glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, number_of_ctlpoints, &ctlpoints[0][0]);
```

The first parameter it takes is a “target”. It was mentioned above that Bézier curves need not be given visual representations. This is something the designers of OpenGL were well aware of. This target parameter specifies what kind of a representation you want for your Bézier curve — is it going to define a curve drawn in 3-space, or is it going to represent color values, etc.

The next two parameters represent the range of values over which the evaluator should be evaluated. The third parameter is the "width" of your coordinates — e.g., 3D coordinates or 4D (homogeneous) coordinates.

The last two parameters specify the number of control points, and then a pointer to the control points themselves.

Internally, the effect of setting up a map is to create a function that is each control point multiplied by the appropriate Bernstein polynomial.

After you specify a map, you need to turn it on. You can have multiple maps defined (one for each target) and the functions that evaluate maps (glEvalCoord, glMapGrid, and glEvalMesh) simultaneously evaluate all maps that are currently enabled. To enable and disable maps, call glEnable or glDisable with the target as the argument. E.g., “glEnable(GL_MAP1_VERTEX_3);”.

There are two ways to evaluate evaluators. One way is with glEvalCoord which evaluates the map at the value specified. So, to create a curve you need to do something like:

```
        glBegin(GL_LINE_STRIP);
            for (i = 0; i <= curveResolution; i++)
                glEvalCoord1f((GLfloat) i/(GLfloat) curveResolution);
        glEnd();
```

If you are going to evaluate the map over an evenly spaced grid (as we do in the above code example), there is a shortcut using glMapGrid and glEvalMesh. glMapGrid creates an evenly spaced grid and then glEvalMesh repeatedly evaluates the map over that grid. E.g.,

```
        glMapGrid1f(curveResolution, 0.0, 1.0);
        glEvalMesh1(GL_LINE, 0, curveResolution);
```

The first call creates a grid with “curveResolution” number of points between 0 and 1. The second call repeatedly evaluates the map starting at the 0th element of the grid up to the curveResolution-th element.

NURBS

Another way to draw Bézier curves and B-Splines is to use the NURBS interface. Internally, the NURBS interface is built on top of the evaluators we just looked at. From the programmers point of view (vs. the OpenGL implementors), the NURBS interface is simpler to use.

The NURBS interface requires a NURBS context object that is passed into each call (the poor man's object-orientation). So, the first thing to do is to create a NURBS context object:

```
nurbs = gluNewNurbsRenderer();
```

If you remember, a NURBS is defined by control points *and* a knot vector. So, to use the NURBS interface, you need to create a knot vector. But you may also remember that a Bézier curve is a B-Spline where the first p+1 knots = 0, the last p+1 knots = 1.

gluNurbsCurve is the function that actually evaulates the NURBS. The first argument is the NURBS context object. The next two are the number of knots and a pointer to the knot vector. Then comes the width of each control point is (3D vs. 4D) and a pointer to the control points.

The next argument is the “order” of the curve. The order is the degree + 1. And, looking back above, the order is equivalent to the number of knots minus the number of control points. The final argument is the "type" which is the same as the evaluator's target argument.

Summing up, the code should look like:

```
GLfloat knots[8] = {
      0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0
};

gluBeginCurve(nurb);
      gluNurbsCurve(nurb,
                        8, knots,
                        3,
                        &ctlpoints[0][0],
                        4,
                        GL_MAP1_VERTEX_3);
gluEndCurve(nurb);
```

Et Voila! Now you are ready to start experimenting with Bézier curves and NURBS using OpenGL.

References

Gladden, Jonathan. Literature Review: Curves and Surfaces. http://www.accad.ohio-state.edu/~jgladden/GradCourses/ComputerGraphicsHistory/LitReview/LitReview01.html

Jones, Huw. (2001). *Computer Graphics through Key Mathematics*. London: Springer-Verlag.

Kilgard, Mark J. Avoiding 16 Common OpenGL Pitfalls.
http://www.opengl.org/developers/code/features/KilgardTechniques/oglpitfall/oglpitfall.html

Weisstein, Eric. B-Spline from Mathworld. http://mathworld.wolfram.com/B-Spline.html

Woo M., Neider, J., Davis, T., Shreiner, D., OpenGL Architecture Review Board. (1999). *OpenGL Programming Guide*. Reading, MA: Addison-Wesley.
and older version is available online: http://fly.cc.fer.hr/~unreal/theredbook/