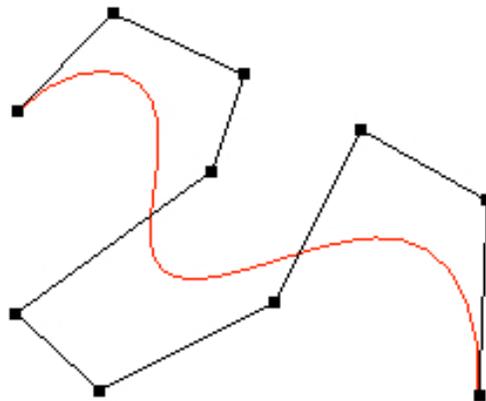


An Introduction to Bezier Curves, B-Splines, and Tensor Product Surfaces with History and Applications

Benjamin T. Bertka
University of California Santa Cruz

May 30th, 2008



1 History

Before computer graphics ever existed there were engineers designing aircraft wings and automobile chassis by using splines. A spline is a long flexible piece of wood or plastic with a rectangular cross section held in place at various positions by heavy lead weights with a protrusion called ducks, where the duck holds the spline in a fixed position against the drawing board [2]. The spline then conforms to a natural shape between the ducks. By moving the ducks around, the designer can change the shape of the spline. The drawbacks are obvious, recording duck positions and maintaining the drafting equipment necessary for many complex parts will take up square footage in a storage facility, costs that would be absorbed by a consumer. A not so obvious drawback is that when analyzed mathematically, there is no closed form solution [3].

However, in the 1960s a mathematician and engineer named Pierre Bezier changed everything with his newly developed CAGD tool called UNISURF. This new software allowed designers to draw smooth looking curves on a computer screen, and used less physical storage space for design materials. Bezier's contribution to computer graphics has paved the road for CAD software like Maya, Blender, and 3D Max. His developments serve as an entry gate into learning about modern computer graphics, which spawned a relatively new mathematical object known as a spline, or a smooth curve specified in terms of a few points.

2 Applications for Bezier curves and B-splines

To see why splines are important, let's consider the problem of designing an aircraft wing. Let's assume that the Air Force is designing the latest and greatest jet fighter plane, and the wing is currently being designed according to specifications that include and promote optimal behavior under extreme turbulence due to mach speeds. To even complicate the design further, the wing has to look nice on the rest of the jet so as to promote more military funding and generate recruits into the Air Force. There are many different possible designs for a wing, some that are more optimal than others, and some that are more aesthetically pleasing than others as well. To find a balance between optimizing the air flow around the wing and how the shape looks is quite a task.

Assume for a moment that your job as a visualization specialist is to make use of the computer system that utilizes recorded data from an aircraft testing facility. Their system is able to relate the flow of turbulence around the wing of an aircraft to the shape of the aircraft's wing. You are asked to create a piece of software using their model to allow an efficient way for a designer to specify an optimal and aesthetically pleasing shape for the wing. The relation between shape and turbulence is already completed, so it is your job to give global control to the designer. Mainly, a way of specifying smooth curves on a computer screen is required, and splines are the natural way of completing the task.

In order to effectively represent a smooth curve on a computer screen, we need to somehow approximate it. We come to this realization upon the simple fact that a computer can only draw pixels, which have a predefined width and height. If you get really close to an LCD screen and observe the tiny squares making up the outline of an image, it's easy to understand that everything represented in computer graphics is an approximation at best.

3 Bezier Curves

We start by letting the four points of a control polygon be the set P , where $P = \{ P_0, P_1, P_2, P_3 \}$. The position of these points in two or three dimensions determines the curvature of the curve. Now let $Q(u)$ be a parametrically defined vector valued function where $0 \leq u \leq 1$. As u varies from 0 to 1, the vector values of $Q(u)$ sweep out the curve [3].

Recall the Bernstein polynomials of degree n (we will use this in the general case of degree- n Bezier curves):

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i} \quad (1)$$

, where $i = 0, 1, 2, \dots, n$, and

$$\binom{n}{i} = \frac{n!}{i!(n-i)!} \quad (2)$$

Bezier curves use the special case of the Bernstein polynomial where $n = 3$. Since Bezier curves use the Bernstein polynomial as a basis, it is ok to use the term "Bezier/Bernstein spline" when talking about these curves. Now we can mathematically define a degree three Bezier curve. Let $Q(u)$ be such a curve:

$$Q_3(u) = B_0^3(u)P_0 + B_1^3(u)P_1 + B_2^3(u)P_2 + B_3^3(u)P_3 \quad (3)$$

, where each $B_i^3(u)$ term is scalar valued in \Re , and the control point P_i is vector valued in \Re^3 . In matrix form we can also write:

$$Q_3(u) = [B][P] \quad (4)$$

Since we are using the Bernstein basis, several properties of Bezier curves are revealed: each basis function is real; the degree of the polynomial defining the curve segment is one less than than the number of points that compose the control polygon; the first and last points on the curve are coincident with the first and last points of the control polygon; the tangent vectors at the ends of the curve have the same direction as the first and last spans of the control polygon.

Finding expressions for each basis term is straightforward for the degree three case, but can be very tedious for splines of degree greater than three. Knowing that $n = 3$ and we have 4 control points, by using (1) we compute:

$$B_0^3(u) = (1 - u)^3 \quad (5)$$

$$B_1^3(u) = 3u(1 - u)^2 \quad (6)$$

$$B_2^3(u) = 3u^2(1 - u) \quad (7)$$

$$B_3^3(u) = u^3 \quad (8)$$

Putting these terms back into (3) gives us our polynomial $Q(u)$:

$$Q_3(u) = (1 - u)^3 P_0 + 3u(1 - u)^2 P_1 + u^2(1 - u) P_2 + u^3 P_3 \quad (9)$$

Hence, for each value of u , we find point on the Bezier curve. A computer graphics library like OpenGL can take each point and connect them with lines, making it easy to create a program that allows for a designer to specify a control polygon with four clicks of a mouse. The program can easily allow for repositioning of the points via drag and drop, thus giving a designer an easy way to specify a smooth form (see Figure 1).

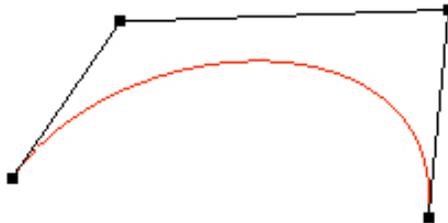


Figure 1: A Bezier curve of degree three. This curve was made using an interactive program written by the author.

Like with many polynomials, the Bezier curve has "sweet" spots along the trajectory that gives a maximum amount of "pull." We usually don't pull on the start and end points of a curve, so lets just consider the degree three case for simplicity's sake. A maximum value exists for the basis functions $B_1^3(u_1)$, and $B_2^3(u_2)$. To find the maximum value for u_i , we recall the mean value theorem:

If f is a function that is continuous and differentiable on a closed interval $[a,b]$, then there exists a number c inside the interval (a,b) such that:

$$f(b) - f(a) = f'(c)(b - a) \quad (10)$$

Let $B_i^3(u)$ be a Bernstein basis function that is continuous on the closed interval $[0,1]$. Then the second and third control points corresponds to $i=1,2$, and by (6), $B_1^3(u) = 3u(1 - u)^2$, and by (7), $B_2^3(u) = 3u^2(1 - u)$.

Since each basis function is continuous, they are differentiable, and we let $B_i^{\prime 3}(u)$ denote the derivative of the i^{th} basis function of $Q(u)$. Set this derivative to zero, and then (10) implies:

$$B_1^3(1) - B_1^3(0) = B_1^{\prime 3}(c_1) = 0 \quad (11)$$

$$B_2^3(1) - B_2^3(0) = B_2^{\prime 3}(c_2) = 0 \quad (12)$$

Using (1), We know for sure that (11) and (12) are true, since for an arbitrary i^{th} term of $Q(u)$:

$$B_i^3(1) - B_i^3(0) = 0 \quad (13)$$

Hence, we can find a $u_1, u_2 \in [0,1]$ such that $B_i^{\prime 3}(u_1) = B_i^{\prime 3}(u_2)$. We start by letting the four derivatives of $Q(u)$'s basis functions (5)-(8) be computed as:

$$B_0^{\prime 3}(u) = -3u^2 + 6u - 3 \quad (14)$$

$$B_1^{\prime 3}(u) = 9u^2 - 12u + 3 \quad (15)$$

$$B_2^{\prime 3}(u) = -9u^2 + 6u \quad (16)$$

$$B_3^{\prime 3}(u) = 3u^2 \quad (17)$$

Setting (15) to zero we find $u_1 = 1/3$, and evaluating (6) at u_1 , we find that the maximum value of the second basis function of $Q(u)$ to be $4/9$. Similarly, we set (16) to zero and find that $u_2 = 2/3$, and evaluating (7) at u_2 , we find that the value of the third basis function of $Q(u)$ to be $4/9$. Hence, P_1 and P_2 have the most influence or "pull", at $u_1 = 1/3$, and $u_2 = 2/3$, respectively; where the max value for both P_1 and P_2 is $4/9$. It is no coincidence that the max value for the second and third basis functions are both $4/9$. It can be shown, although not included here, that (5) and (6) are the same when evaluated at $u=1/3$, and (7) and (8) are the same when evaluated at $u = 2/3$. This is due to the symmetry of the Bernstein basis.

Below are some examples of more Bezier curves:

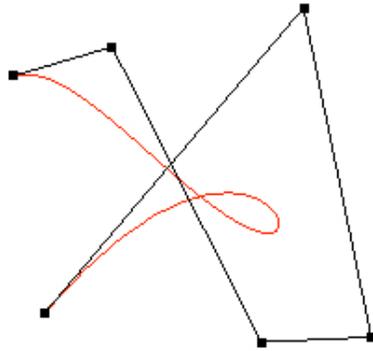


Figure 2: A Bezier curve of degree five.

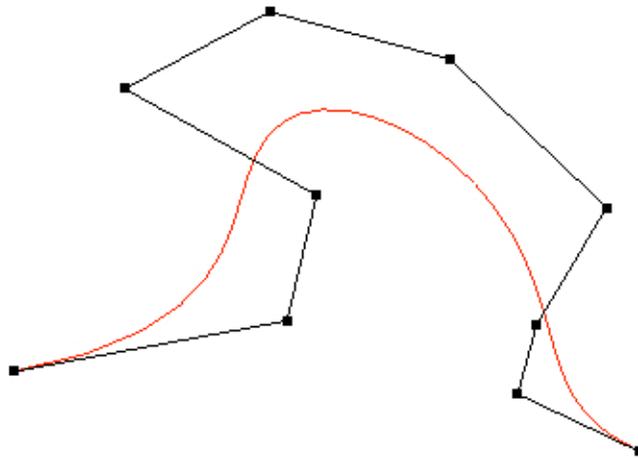


Figure 3: A Bezier curve of degree nine.

In general, a Bezier spline of degree k is defined on $n = k + 1$ control points, where the set of control points $P = \{ P_0, P_1, \dots, P_k \}$ forms a control polygon with n vertices. Let $Q(u)$ be a parametrically defined degree k Bezier spline, then:

$$Q_k(u) = \sum_{i=0}^k B_i^k(u) P_i \quad (18)$$

, where the basis $B_i^k(u)$ is defined using (1), the Bernstein polynomial, and $u \in [0,1]$.

In this general case there are some immediate consequences of using the Bernstein basis:

$$B_i^k(u) \in \Re \quad (19)$$

$$B_0^k(0) = B_k^k(1) = 1 \quad (20)$$

$$\sum_{i=0}^k B_i^k(u) = 1 \quad (21)$$

$$B_i^k(u) \geq 0 \quad (22)$$

By (20), the curve starts at $Q(0) = P_0$, and ends at $Q(1) = P_k$. By (21) and (22), it is implied that each point on $Q(u)$ is a weighted average of control points. Taken together, these features force a Bezier curve to lie entirely in the convex hull of the control polygon.

A Bezier curve of degree k lies tangent to the first and last control points. To see how this is true, we first formalize our definition of the Bezier derivative. Let $Q(u)$ be a Bezier curve of degree k with the set of $k + 1$ control points, P . Then its derivative $Q'(u)$ is defined as:

$$Q'_k(u) = k \sum_{i=0}^{k-1} B_i^{k-1}(u) (P_{i+1} - P_i) \quad (23)$$

We see that the degree is decreased by one and $Q'(u)$ is also a Bezier curve with control points $k(P_{i+1} - P_i)$

Evaluating $Q'(u)$ at 0 and 1:

$$Q'_k(0) = k(P_1 - P_0) \quad (24)$$

$$Q'_k(1) = k(P_k - P_{k-1}) \quad (25)$$

By (23), (24), and (25), we prove that the Bezier curve of degree k starts in the direction of $P_1 - P_0$, and ends in the direction of $P_k - P_{k-1}$.

4 Rational Bezier Curves

Now that we understand Bezier curves of degree k , we can consider the rational form of a Bezier curve. Graphics Processing Units, or GPU's actually need more information than just a point $\{x, y, z\} \in \mathfrak{R}^3$. The problem with just a regular point is that there is no info on how to project it. Homogeneous coordinates exist in order to make the projective transformation easier to work with [5]. The rational form has advantages in that it can represent a wide range of curves, and surfaces (more on surfaces in a little while). Curves could be in the form of circles, ellipses, parabolas, and hyperbolas; surfaces can be in the form of spheres, ellipsoids, cylinders, cones, paraboloids, hyperboloids, and hyperbolic paraboloids [1]. The only difference in Rational Bezier curves is that the coordinates that specify the curve are in one dimension higher than their nonrational counterpart.

For example, if we wish to express a Bezier curve in \mathfrak{R}^3 , we specify a control point as $\{x, y, z, \omega\}$, where ω is a weight that allows for the transition between regular and homogeneous coordinate space. Let h be a map from homogenous coordinate space to regular space, then we define h as

$$h(x, y, z, \omega) = (x/\omega, y/\omega, z/\omega) \quad (26)$$

In computer graphics homogeneous coordinates allow the designer to work with normalized points for $\omega = 1$, and also to define a point at infinity, saw when $\omega = 0$ [4].

Now we make a definition of a rational Bezier curves:

Let w_i for $i=0, \dots, k$, be the $k + 1$ weights corresponding to the control points P_i . The the rational Bezier curve of degree k becomes:

$$Q_k(u) = \frac{\sum_{i=0}^k w_i B_i^k(u) P_i}{\sum_{r=0}^k w_r B_r^k(u)} \quad (27)$$

With some rearranging we can write (27) as linear combination of rational basis functions:

$$Q_k(u) = \sum_{i=0}^k P_i \left[\frac{w_i B_i^k(u)}{\sum_{r=0}^k w_r B_r^k(u)} \right] \quad (28)$$

Where we denote the term in brackets by $R_i^k(u)$,

$$R_i^k(u) = \frac{w_i B_i^k(u)}{\sum_{r=0}^k w_r B_r^k(u)} \quad (29)$$

And we now have a familiar looking form by replacing the terms in brackets in (28) by the new term $R_i^k(u)$ as defined in (29),

$$Q_k(u) = \sum_{i=0}^k R_i^k(u) P_i \quad (30)$$

Hence we have the same curve equation as (18), but the univariate basis is rational. It is immediately implied that all the rules for the nonrational Bezier curve carry over to the rational Bezier curve [1]. Noting that if the denominator of the rational Bezier curve is one, the nonrational and rational curves are the same (except for the extra ω in the coordinates).

5 Rational B-Splines

B-splines are like Bezier curves because they both use a control polygon to define the curve, and are helpful due to their control points' local control of the resulting shape. The B in B-spline stands for "basis," and the basis is specified by the Cox-de Boor formula for computing the basis function. What uniquely sets them apart from Bezier curves is that a vector of scalars called a knot vector is figured in to the computation of the basis functions. When these knots are spaced evenly, the B-spline is said to be uniform, and non-uniform otherwise. The basis function considers the knot vector in every computation.

In general, a B-spline can be rational or non-rational, depending on the use of homogeneous coordinates. Let $C_{k,n}(u)$ denote a uniform rational B-spline of order k (degree $k - 1$), where $k \leq n$. Let ω_i for $i = 1, \dots, n$ be the n weights corresponding to homogeneous control points $\{P_1, P_2 \dots P_n\}$. Let \vec{x} be a knot vector such that $x_1 \leq x_2 \leq \dots \leq x_{n+k}$, then the rational B-spline becomes:

$$C_{k,n}(u) = \frac{\sum_{i=1}^n \omega_i N_i^k(u) P_i}{\sum_{r=1}^n \omega_r N_r^k(u)} \quad (31)$$

where the basis functions N are defined using the Cox-de Boor algorithm [6]:

Set $N_j^1(u) = 1$ if $x_j \leq u < x_{j+1}$, and zero otherwise.

Let the order $k = p + 1$, then $N_j^k(u)$ becomes $N_j^{p+1}(u)$, where

$$N_j^{p+1}(u) = \frac{u - x_j}{x_{j+p} - x_j} N_j^p(u) + \frac{x_{j+p+1} - u}{x_{j+p+1} - x_{j+1}} N_{j+1}^p(u) \quad (32)$$

We may regroup the terms in (31) as we did in the previous sections as

$$C_{k,n}(u) = \sum_{i=1}^n P_i \left[\frac{\omega_i N_i^k(u)}{\sum_{r=1}^n \omega_r N_r^k(u)} \right] \quad (33)$$

where we denote the term in brackets by $U_i^k(u)$, where

$$U_i^k(u) = \frac{\omega_i N_i^k(u)}{\sum_{r=1}^n \omega_r N_r^k(u)} \quad (34)$$

And substituting (34) back into our equation (33), we get the compact expression for $C_{k,n}(u)$,

$$C_{k,n}(u) = \sum_{i=1}^n U_i^k(u) P_i \quad (35)$$

Hence the B-spline is defined by its basis, where the basis is heavily influenced by the knot vector. If the knots were not evenly spaced apart, the curve would be a non-uniform rational B - spline, a.k.a. NURB.

Computing a B-spline is a formidable task. And we explain below the computation in detail to have a good understanding of it.

5.1 B-spline Computation

Normally, the B-spline is computed by a computer, since a smooth curve needs small time steps. To make comprehension easier for the novice in computer graphics, or non-programmer, my best attempt at describing how the computation is in terms of a regular uniform B-spline.

We start by choosing an order for the spline. In our case, let the order be k . We specify a set of n control points. We call this set the control set. Depending on desired smoothness, the number of points on our curve can vary, and denote this set of points to be the curve set. With this information we may now begin computing.

The first thing we need to do is specify a uniform knot vector \vec{x} of length $n+k$. We compute it as follows. Set each knot x_i to be zero. Then for each $1 < i \leq n+k$, check to see if both conditions, $i > n$ and $i < n+2$ hold. If they do, set x_i to the value of $x_{i-1} + 1$. If the conditions do not hold, just set x_i to the value of x_{i-1} . As an example, If we choose $k = n = 4$, we find that our knot vector $\vec{x} = \{0, 0, 0, 0, 1, 1, 1, 1\}$, which also happens to be the correct knot vector to make a Bezier curve out of a B spline.

Before we can calculate the points in the curve set, we need to decide on an appropriate step value for the parameter u . The proper step is found by dividing the value of knot x_{n+k} by one less than the number of points on the curve set (19 steps plus the *zeroth* step is 20 steps). Using the knot vector from the previous example, if we had 20 points in our curve set, and $n+k=8$, then the step would be $1/19$ which is about 0.052632.

We now begin the task of computing the curve set points on the B-spline. We need to keep in mind that the number of basis functions we compute for each time step is going to equal that of the number of points in the control set. So for the entire spline, the number of basis computations is going to be equal to the product of the magnitudes of the control and curve sets. So in our running example, we have 20 points on the curve set to compute and four control points, so there is 80 basis functions to compute!

At each step, use the Cox-de Boor algorithm to compute the value of the basis function. During each iteration of the steps, the entire knot vector \vec{x} is considered in with the step value u . We see that the Cox-de Boor algorithm is recursive, so when using this system special care has to be made in order to compute the basis correctly. Once we have a set of n basis functions for the control points at a specific step u , we find the coordinates for the curve point at the step by multiplying the i^{th} basis function to the i^{th} control point. The resulting values are inserted into the curve set, and we generate a collection of points that when plotted together, and connected with line segments can aesthetically resemble a curve.

The number of computations required to make a B-spline are really large, so it is obvious that a computer is needed for the computation of splines. As a concrete example, let a set of four control points be as follows: $P_4 = \{(1, 1, 1), (2, 3, 1), (3, 3, 1), (5, 1, 1)\}$. Letting $n = 4$, $k = 4$, knot vector $\vec{x} = \{0, 0, 0, 0, 1, 1, 1, 1\}$, we use a computer program for computing the following basis functions for 20 points on the control set using the previously mentioned step (see screen shot from my computer).

```

bbertkas-computer:program bbertka$ ./a.out

Basis Calculation

step: 0.052632

u: 0.000000, basis: 1.000000 0.000000 0.000000 0.000000
u: 0.052632, basis: 0.850270 0.141712 0.007873 0.000146
u: 0.105263, basis: 0.716285 0.252807 0.029742 0.001166
u: 0.157895, basis: 0.597172 0.335909 0.062983 0.003936
u: 0.210526, basis: 0.492054 0.393643 0.104972 0.009331
u: 0.263158, basis: 0.400058 0.428634 0.153084 0.018224
u: 0.315789, basis: 0.320309 0.443505 0.204695 0.031491
u: 0.368421, basis: 0.251932 0.440881 0.257180 0.050007
u: 0.421053, basis: 0.194052 0.423385 0.307917 0.074646
u: 0.473684, basis: 0.145794 0.393643 0.354279 0.106284
u: 0.526316, basis: 0.106284 0.354279 0.393643 0.145794
u: 0.578947, basis: 0.074646 0.307917 0.423385 0.194052
u: 0.631579, basis: 0.050007 0.257180 0.440881 0.251932
u: 0.684210, basis: 0.031491 0.204695 0.443505 0.320309
u: 0.736842, basis: 0.018224 0.153084 0.428634 0.400058
u: 0.789474, basis: 0.009331 0.104972 0.393643 0.492054
u: 0.842105, basis: 0.003936 0.062983 0.335909 0.597171
u: 0.894737, basis: 0.001166 0.029742 0.252807 0.716285
u: 0.947368, basis: 0.000146 0.007873 0.141712 0.850269
u: 1.000000, basis: 0.000000 0.000000 0.000000 1.000000

Control Set

1.000000 1.000000 1.000000
2.000000 3.000000 1.000000
3.000000 3.000000 1.000000
5.000000 1.000000 1.000000

Curve Set

1.000000 1.000000 1.000000
1.158041 1.299169 1.000000
1.316956 1.565097 1.000000
1.477621 1.797784 1.000000
1.640910 1.997230 1.000000
1.807698 2.163435 1.000000
1.978860 2.296399 1.000000
2.155271 2.396122 1.000000
2.337804 2.462604 1.000000
2.527337 2.495845 1.000000
2.724741 2.495845 1.000000
2.930894 2.462604 1.000000
3.146668 2.396122 1.000000
3.372941 2.296399 1.000000
3.610584 2.163435 1.000000
3.860475 1.997230 1.000000
4.123487 1.797784 1.000000
4.400495 1.565098 1.000000
4.692374 1.299170 1.000000
5.000000 1.000000 1.000000
bbertkas-computer:program bbertka$ █

```

Figure 4: Computing 20 points on a B-spline for $n=4$, $k=4$, $\vec{x} = \{0, 0, 0, 0, 1, 1, 1, 1\}$

6 Tensor Product Surfaces

Now that we have discussed rational Bezier curves and B-Splines, it is time to introduce tensor product surfaces. For both the rational and non rational case, properties of their cohabitant curves apply. A tensor product surface is defined by a control net, similar to that of a control polygon for a curve, only we parameterize in two different directions, namely $u \in [0,1]$ and $v \in [0,1]$. This control net is specified by a set of points, where if we use homogeneous coordinates the surface is referred to a rational tensor product surface. For simplicities sake, we use the rational form of the tensor product surface with a Bernstein basis for our definition.

Let us define a rational tensor product surface $Q_{k,l}(u, v)$ as being a degree k in the u direction and degree l in the v direction. Then our rational tensor product surface becomes

$$Q_{k,l}(u, v) = \frac{\sum_{i=0}^k \sum_{j=0}^l \omega_{ij} P_{ij} B_i^k(u) B_j^l(v)}{\sum_{r=0}^k \sum_{s=0}^l \omega_{rs} B_r^k(u) B_s^l(v)} \quad (36)$$

where each control point in the set P of three dimensional points that make up the control net will be denoted in homogeneous coordinates as $P_{ij}^\omega = \{\omega_{ij}x_{ij}, \omega_{ij}y_{ij}, \omega_{ij}z_{ij}\}$. And the basis function $B_i^k(u)$ is the Bernstein polynomial from (1).

Just like we rearranged the rational curve in (28), we rearrange the rational surface to be

$$Q_{k,l}(u, v) = \sum_{i=0}^k \sum_{j=0}^l P_{ij} \left[\frac{\omega_{ij} B_i^k(u) B_j^l(v)}{\sum_{r=0}^k \sum_{s=0}^l \omega_{rs} B_r^k(u) B_s^l(v)} \right] \quad (37)$$

where the term inside the brackets is denoted by $R_{ij}^{kl}(u, v)$

$$R_{ij}^{kl}(u, v) = \frac{\omega_{ij} B_i^k(u) B_j^l(v)}{\sum_{r=0}^k \sum_{s=0}^l \omega_{rs} B_r^k(u) B_s^l(v)} \quad (38)$$

Thankfully we are able to make the replacement of expression (34) within (31), and $Q_{k,l}(u, v)$ is rewritten in a way that highlights the bivariate basis function $R_{ij}^{kl}(u, v)$

$$Q_{k,l}(u, v) = \sum_{i=0}^k \sum_{j=0}^l P_{ij} R_{ij}^{kl}(u, v) \quad (39)$$

Hence, rational Bezier curves and tensor product surfaces provide an intuitive way of approximating a designer's models based on control polygons, and control nets. By merely changing the weights, we can make a Bezier curve become a rational curve, and a rational tensor product surface a Bezier surface. A potential problem with Bezier curves and surfaces is that their control points have global control. While computing is efficient, just moving one control point has a radical effect on the shape's image. This can be bad when only minor adjustments may be needed when trying to create a certain shape.

If we choose local control, we would want a B-spline surface. A B-spline tensor equation is going to look very similar to the Bezier tensor equations so it is omitted. However, if we wished to translate a Rational Bezier surface into a Rational B-spline surface, we would have similar parameterization in "u-v" space, use uniform knot vectors, and homogenous coordinates, hence creating an URB. If the knots are not evenly spaced, we would have a NURB, an industry standard for use in surface representation.

7 Sample Data Structure

The images in this paper were generated by a computer program written by the author. This program was written in C++ and uses OpenGL libraries, and specifically `glMap1()` and `glEvalue()` helper commands to create the curve (see OpenGL documentation for more detailed explanation). The data structure to handle the Bezier object is in the form of a display list, and illustrates the ease at which a curve is generated. While the interactive program is too long to include in this paper, the vital part of the generation of the curve is shown here.

```
void struct_Bezier(void){  
  
    int i; //used for indexing through control sets  
  
    //openGl specific code for evaluating basis  
    glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, VertexCount, &ControlSet[0][0]);  
    glEnable(GL_MAP1_VERTEX_3);  
  
    //begin the Bezier object  
    glNewList(bezier, GL_COMPILE);  
  
    //set color to RGB black  
    glColor3f(0.0, 0.0, 0.0);  
  
    //draw control points  
    glPointSize(5.0);  
    glBegin(GL_POINTS);  
        for (i = 0; i < VertexCount; i++)  
            glVertex3fv(&ControlSet[i][0]);  
    glEnd();  
  
    //draw the control polygon  
    glBegin(GL_LINE_STRIP);  
        for (i = 0; i < VertexCount; i++)  
            glVertex3fv(&ControlSet[i][0]);  
    glEnd();  
  
    //set color to red  
    glColor3f(1.0, 0.0, 0.0);  
  
    //compute and draw the 30 segments of the curve  
    glBegin(GL_LINE_STRIP);  
        for (i = 0; i <= 30; i++)  
            glEvalCoord1f((GLfloat)i/30.0);  
    glEnd();  
  
    glEndList();  
}
```

Figure 5: A Bezier object in the form of a display list. Understanding the mathematics helps us better understand the helper functions.

8 Conclusion

A brief history of Bezier curves and B-splines was presented, and in the context of Computer Aided Geometric Design, real world applications were proposed. We have reviewed the Bernstein polynomial and detailed Bezier curves in both the rational and non-rational case. Using a computer program created by the author, various images of Bezier curves were generated and screen captures included as examples. We have covered the rational and non rational, uniform and nonuniform, cases of B-splines. The Cox-de Boor algorithm was presented and computation of B-splines discussed in a detailed manner. A computer program was created and run to show the step by step computations performed by a B-spline computation. Tensor product surfaces were detailed in the Bernstein case, and the B-spline surface introduced. A sample data structure in the form of a display list from the author's OpenGL program was shown.

References

- [1] Brian A. Barsky. Acm/siggraph '90 course 25: Parametric bernstein/bezier curves and tensor product surfaces, Dallas, TX. Aug. 7th 1990.
- [2] Robert C. Beach. *An Introduction to Curves and Surfaces of Computer-Aided Design*. Van Nostrand Reinhold, 1991.
- [3] Samuel R. Buss. *3D Computer Graphics – A Mathematical Introduction with OpenGL*. Cambridge University Press, 2003.
- [4] L. Piegl. Infinite control points—a method for representing surfaces of revolution using boundary data. *IEEE (Institute of Electrical and Electronics Engineers)*, March 1987.
- [5] C. Ramakrishnan. An introduction to nurbs and opengl, 2002.
- [6] Wayne Tiller. Rational b-splines for curve and surface representation. *IEEE (Institute of Electrical and Electronics Engineers)*, September 1983.