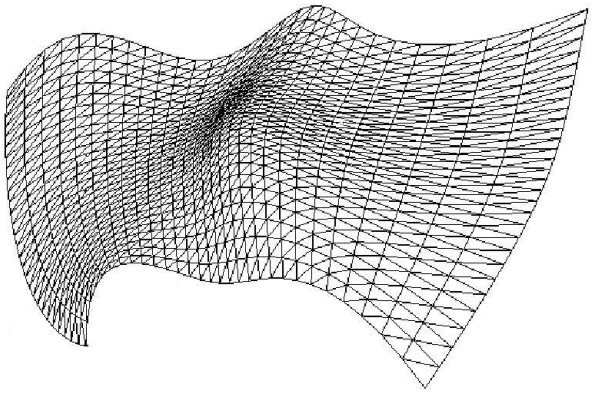
Bezier and B-spline Technology



Fredrik Andersson
fmu@home.se

Supervisor:
Berit Kvernes
beritk@cs.umu.se

June 11, 2003

Abstract

Graphical curves and surfaces are hot topics in many different areas of research and application, not least due to the rapid progression of computer aided design (CAD) and gaming technology. Two of the most popular representations for this field are the Bezier and B-spline curves and surfaces.

This thesis will describe the theory behind these topics from a programmers viewpoint, together with the developed software which is a rather intuitive application to model different types of Bezier and B-spline curves and surfaces in real-time.

The mathematics behind these elements can be quite intimidating to the normal user at first, hence this thesis is dedicated to students with some background in elementary linear algebra and variable calculus but not necessarily in computer graphics.

There are an abundance of interesting aspects to this topic of curves and surfaces, and this thesis will primarily focus on the different ways to use, implement and efficiently evaluate them. Thus, this report will serve as excellent material for students entering the field of polynomial curves and surfaces in computer graphics.

Acknowledgements

I would like to thank my supervisor Berit Kvernes for the great support, both morally and technically. Also, love goes to my parents which have assisted with finance, corrective reading and moral support.

Francois Rimasson, 3D-artist, and Andreas Johansson, fellow student, supplied me with superb visual material for the presentation of my work. This was very appreciated and I thank both of you.

Contents

List of Figures

Chapter 1

Introduction

Over the last two decades the scene of computer graphics has literally exploded with progress in all directions; the arrival of dedicated 3D hardware, computer generated animation and faster computers to name a few key events.

One of these directions focus on displaying smooth curves and surfaces, suitable for modeling landscapes, faces and other topologies of interest. Here arise the need for Bezier and, in particular, B-spline curves, two concepts with roots in the late 1950's.

These two types of curves are made up of an arbitrary amount of *control points* which provide a guideline for the evaluated curve to approximate. The approximation can be modified to suit its needs in a number of different ways; weighted control points, knot vectors, varying the degree of the curve, and so forth. All will be expanded on and explained later.

One might ask why parametric representations of curves and surfaces are to prefer over the polygonal representation. A few good reasons are tiny size, automatic detail resolution and scalability[1]. The advantages are numerous and will be thoroughly investigated.

1.1 Background

Before the 1950's, designing smooth shapes was a very tedious effort. Environment dependent aluminum strips, clumsy full scale moulding forms and a timetable that would havoc most modern companies.

In 1959, an employee at the Citroen automobile company named Paul de Faget de Casteljau came up with the simple idea of iterating affine combinations of polygon meshes to obtain smooth surfaces suitable for modeling car chassis. Almost simultaneously, Peugeot employee Pierre Bezier worked on the intersection of partial cylinders to achieve the same goals. Both versions, which produced equal curves, are today known as the Bezier curve

[1]The ability to transfer between small and large systems

since Pierre Bezier was the first to write a public paper on the subject. Paul de Faget de Casteljau was however not forgotten, as his named is etched in the Casteljau algorithm, one of the most common ways to evaluate Bezier curves. Both of these methods, and their extensions to other curves, are explained in later sections.

The Bezier curve was formally presented in [9] and has since then been a very common way to display smooth curves, both in computer graphics and mathematics.

The B-spline curve is an extended version of the Bezier curve that consists of segments, each of which can be viewed as an individual Bezier curve with some additions that will be covered in chapter 3.

1.2 Objective

The purpose of this thesis is to thoroughly investigate and explain the concept, use and implementation of Bezier and B-spline curves and surfaces to readers with little experience in the field. This is accomplished primarily with the report, but also practically aided from the developed software.

There are a number of issues that perplexes the use of parametric curves or surfaces in computer graphics. With no priorities, to name a few:

- Creating and sculpting in an intuitive manner
- Evaluation for effective real-time[2] use
- Collision detection
- Intimidating mathematics
- Approximation errors

Since computer graphics and gaming technology are appreciated topics with most computer science students (and hence, readers of this thesis), these areas will serve as a guideline for the report.

1.3 Structure of the thesis

Mathematics are fundamental to parametric curves and surfaces, so the report begins with a very thorough explanation of the Bezier and B-spline curve concepts and the related topics of interest (Hermite curves, for instance). Figures are used extensively to demonstrate and simplify the theories, with some references to the developed thesis application.

Further on, sections on the actual use of curves and surfaces in the real world followed by some discussion of advantages and disadvantages with the contemporary representations.

[2] Around 25 frames per second or above, loosely speaking

The vast field of implementation options are covered at the end of the thesis, together with some discussions about efficiency, collision detection and related topics.

1.4 Prerequisites

It is assumed that the reader of this thesis has studied elementary linear algebra, some multivariate calculus and maybe a dash of computer graphics. The two latter are not really necessary, but improves the understanding of the underlying mathematics and theories.

1.5 Software

The software developed for this thesis is a portable OpenGL[3] application that has a graphical user interface written with GLUI[4]. Supplied with the thesis are three packages:

- Compiled Linux x86 version
- Compiled Windows 2000/XP version
- Source code tarball[5]

The software has been tested on Mandrake 8.0, Mandrake 9.0, Windows 2000 and Windows XP. Since both OpenGL and GLUI exist for nearly all platforms, compiling on other systems should not be a problem. Make sure to edit the Makefile to match your system libraries.

The software itself does not have a very steep learning curve. Click and drag the control points, toggle different display information and curve modes from the graphical user interface.

1.6 Notation and Typesetting

This report uses standard LaTeX. Also, the concept of C^x and G^x continuity levels are used extensively, and should be familiar to the reader as I have observed no other formats for describing continuity in the realms of computer graphics.

[3] Portable graphics library for developing 2D- and 3D applications
[4] Portable GUI library, built completely on OpenGL
[5] Common compression format for Unix/Linux, *.tar.gz

Chapter 2

Bezier curves and surfaces

As mentioned in section 1.1, the Bezier curve was developed in the late 1950's. We will focus on the cubic polynomial curve since this is the most used in computer graphics and it is quite easy to extend this case to polynomials of higher degree later on.

We will begin by looking at the Bernstein polynomial evaluation which provide a so called *basis function* for Bezier curves. A later section covers the geometrical equivalent by the de Casteljau algorithm.

2.1 Basis function

A cubic Bezier curve consists of four control points used to position and modify the curve, P_0 through P_3. The two intermediate points P_1 and P_2 are used to specify the endpoint tangent vectors, hence the curve interpolates (passes through) P_0 and P_3 while approximating the other two control points. To accomplish this we need some sort of weighting function which tells us the influence of the control points at a given point on the curve. One can use an arbitrary function to suit the requirements but in most cases we define the *Bernstein polynomials* [1][2]

$$B_{i,n}(u) = \binom{n}{i} u^i (1-u)^{n-i} \tag{2.1}$$

or similarly

$$B_{i,n}(u) = \frac{n!}{i!(n-i)!} u^i (1-u)^{n-i} \tag{2.2}$$

where i is the control point to be weighted and n is the degree of the curve, to be the *basis function*. As we can see in figure 2.1-2.3 these polynomials are symmetric and sum to unity. The resulting Bezier curve is also symmetric, as can be seen in figure 2.4, hence the evaluation direction does not matter.

Since the understanding of the basis function is crucial to grasp this report, we will now expand on the topic.

The parameter $u \in (0, 1)$ can be viewed as a particle trying to interpolate the control points in discrete steps. As it moves along the trajectory, it is drawn towards control point i by an amount determined by $B_{i,3}(u)$, in the cubic case. Without thinking too much about equation 2.1 consider the following image which illustrates the Bernstein polynomials at degree 1 (the curve is just a straight line between the control points).

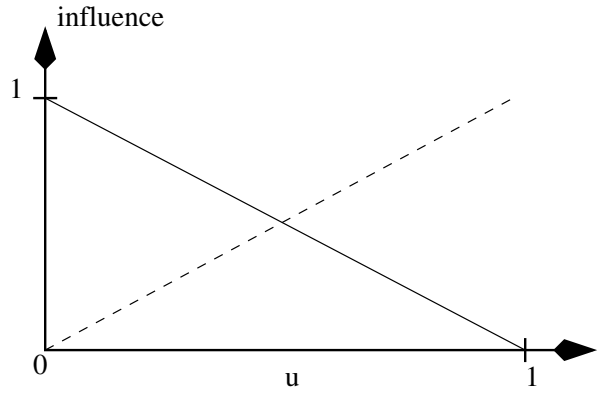


Figure 2.1: Basis functions of degree one

At $u = 0$ the second control point (the dashed line) has zero influence, but as we move along $0 \leq u \leq 1$ the first control point looses influence while the second gains. The result of this is obviously a straight line between the two control points. The basis functions used in this figure are $B_{0,1}$ and $B_{1,1}$.

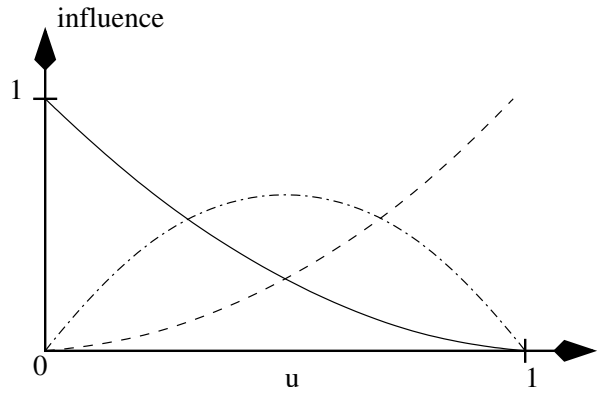


Figure 2.2: Basis functions of degree two

Figure 2.2 illustrates the case of $n = 2$, a quadratic polynomial basis function. The two previous control points get a steeper loss and gain, but never lose influence except at the end points. As we shall see later, this enables the actual Bezier curve to be concave or convex (just as regular second degree

polynomials look). The basis functions used in this figure are $B_{0,2}$, $B_{1,2}$ and $B_{2,2}$.

It is perhaps suitable to mention that there is no local control this far. Moving one control points affects the entire curve.

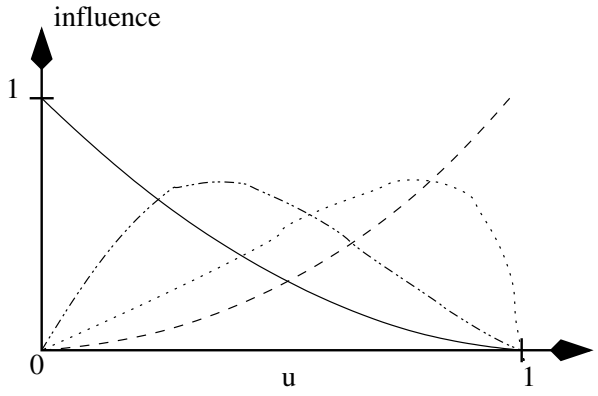


Figure 2.3: Basis functions of degree three

Above we see the case of $n = 3$, a cubic polynomial basis function. This implies that the curve can behave like any cubic curve and can be both concave and convex. This is the most comfortable degree, at least according to regular users, to work with. If we have $n < 3$ we do not have much freedom to work with, and if we have $n > 3$ the curve becomes hard to handle since it tends to "take off" where we do not want it to.

It is also important to note that the basis functions always sum to unity. Anywhere on the figures above, the different values of the curves sum up to one, regardless of curve degree.

2.2 Curve definition

Now that we are familiar with the Bezier basis functions, here is the definition of a Bezier curve of degree n

$$C(u) = \sum_{i=0}^{n} P_i B_{i,n}(u) \tag{2.3}$$

To evaluate the curve at $u \in (0, 1)$ we simply iterate the n+1 control points and compute the corresponding influence. This can also be expressed in matrix form for those more familiar with linear algebra.

$$C(u) = U M_B G_B \tag{2.4}$$

where U is a vector of $[u^n, u^{n-1}, \ldots, u, 1]$, M_B the Bezier basis matrix and G_B is a vector with the n+1 control points. The first, more analytical

description, will be used throughout this thesis since it is more suitable from an implementor's point of view.

In figure 2.4 we see how the symmetry of the Bernstein polynomials is reflected in the actual curve. This is also true if we are using the de Casteljau algorithm (see section 2.7).

From equation 2.3 we discover that no matter the degree of the curve, there is no local control property. All control points affect the curve, and this can be a disadvantage when designing smooth shapes with the Bezier curve. To remedy this flaw we introduce curves made up of several Bezier segments which is covered in the next section.

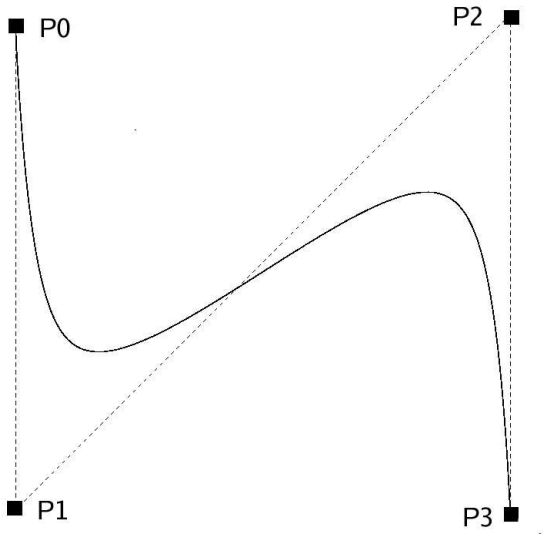


Figure 2.4: Curve and control polygon

To sum up this section, above we see a Bezier curve of degree three with four control points and an outlined control polygon. Notice how the $P0P1$ and $P2P3$ tangents specify the direction at the endpoints. The *convex hull* can be illustrated by enclosing a rubber band around the control points, and this is a fundamental property of the Bezier curve. No matter how we manipulate the control points, we can always rely on the curve to lie within the convex hull. This is a useful property that we take advantage of in collision detection, see section 4.4.

2.3 Curve segments

By concatenating several Bezier curves we can obtain local control. This is done by sharing the first and last control point with the surrounding curves. Obviously, the first and last segment only share one of these.

This will however introduce some discontinuities at the shared control point. A Bezier curve has C^1 continuity to its defined interval, but at the shared vertices we get C^0. Hence, we do not obtain a smooth transition between segments if we do not create these manually, which can be good or bad, depending on the application.

For instance, a simple square cannot be created with concatenated C^1 continuous curves unless we introduce a discontinuity at the corners of it, just as a circle requires smooth transitions and greater continuity.

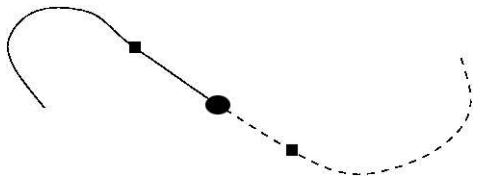


Figure 2.5: Continuity constraint

Assume we have curves $P = P_0, P_1, \ldots, P_{n+1}$ and $Q = P_{n+1}, Q_1, \ldots, Q_{m+1}$ of degree n and m respectively. To force continuity between these two curves we must have tangential continuity between P_n, P_{n+1} and Q_1. In figure 2.5 we have P_n and Q_1 marked with squares while P_{n+1} is a circle, which is the join point of these two curves. The disadvantage with this constraint is that we loose one degree of freedom in each curve since they become "locked" to provide the smooth transition.

2.4 Bezier subdivision

If a Bezier curve is not flexible enough, we can break the curve at a given point u' and create two new Bezier curves that join on u'. An algorithm for this task was presented by de Casteljau [8], and it uses a geometric construction technique.

More formally, this involves finding points L_0, L_1, L_2, L_3 and R_0, R_1, R_2, R_3 so that the Bezier curve formed by L exactly matches $0 \leq u < 0.5$ of the original curve, and R matches $0.5 \leq u \leq 1.0$. Since L and R are separate curves, they are both evaluated locally with $0.0 \leq u \leq 1.0$.

Assume that our original Bezier curve consists of P_0, P_1, P_2, P_3. We then construct a point H that divides P_0P_1, P_1P_2 and P_2P_3 in a ratio of $(1 - u')$. This creates a hull with points $L_0L_1L_2L_3$ and $R_0R_1R_2R_3$, and we have two sets of control points that lie in the hull of the original control points since L and R are weighted sums of P.

This basically is the de Casteljau algorithm which is covered in section 2.7.

2.5 Rational Bezier curves

Rational curves imply that each of the control points are extended with one coordinate which represents the *weight* or amount of influence. A heavy vertex will draw the curve towards it, and a light vertex (can be negative as well) will push the curve away.

This is of great importance when designing curves on the computer screen, which has a limited area. Instead of dragging the control point far away, perhaps outside the canvas, we can simply increase the weight.

The extension does not require much calculation and we now have:

$$C(u) = \frac{\sum_{i=0}^{n} w_i P_i B_{i,n}(u)}{\sum_{i=0}^{n} w_i B_{i,n}(u)} \tag{2.5}$$

where w_i is the weight of P_i. Both sums can be calculated within the same iteration of the control points, so for very little processing time we obtain a curve which is far more useful.

2.6 Bezier surfaces

The Bezier surface is simply an extension of the Bezier curve in two parametric directions, a tensor product of two curves. It is evaluated through

$$C(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} P_{i,j} B_{i,n}(u) B_{j,m}(v) \tag{2.6}$$

where the parametric directions have degree n and m respectively, though it is often the case that $n = m$. This yields $(n+1)*(m+1)$ control points.

Equation 2.6 suggests that the control points are stored in a matrix, but this must not necessarily be true since the order of which they are evaluated is irrelevant to the result.

$$C(u,v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} w_{i,j} P_{i,j} B_{i,n}(u) B_{j,m}(u)}{\sum_{i=0}^{n}\sum_{j=0}^{m} w_{i,j} B_{i,n}(u) B_{j,m}(u)} \tag{2.7}$$

Equation 2.7 looks intimidating, but is simply the evaluation of a rational surface, which is totally analogous to the case of one parametric direction.

Regarding surface concatenation, we often apply the same constraints as discussed in section 2.3, but this time in two parametric directions.

Figure 2.6 shows a tessellated[1] view of a cubic Bezier surface.

[1] After polygons have been constructed

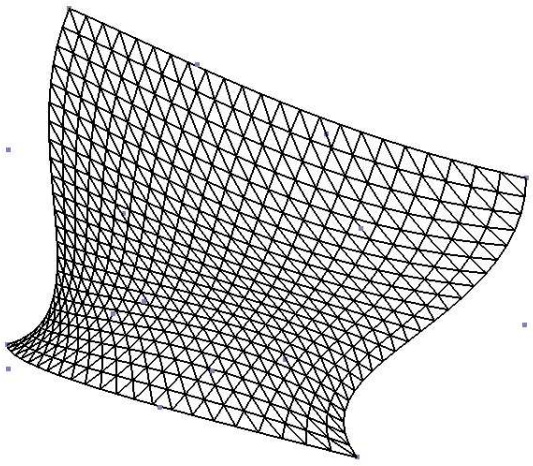


Figure 2.6: Bezier surface

2.7 The de Casteljau algorithm

As mentioned in earlier parts of the report, de Casteljau came up with a geometrical algorithm (or rather, an algorithm that we can interpret geometrically) which solves for any point on a Bezier curve of any degree.
In figure 2.7 we see how the algorithm solves for $u = 0.5$ in a cubic Bezier curve by subdividing the control points by a ratio of $1 - u$. That is, we draw a hull between lines $P_0P_1 * 0.5$, $P_1P_2 * 0.5$ and $P_2P_3 * 0.5$ and continue in the same manner with the new "control points'. Since the curve is of degree 3 we stop here and we have found our curve value. The same procedure applies to any degree but with different recursion depth, and is of course extendable to any $u \in (0, 1)$.

This algorithm can also be represented as a triangular scheme, starting with four control points and eventually reducing them to a single point.

P0			
P1	B0		
P2	B1	C0	
P3	B2	C1	D0

The math behind this scheme is obvious if we have the geometrical interpretation (see figure 2.7) Another common representation is to define a recurrence relation for the de Casteljau algorithm. This also yields a triangular scheme, but we express it as an equation.

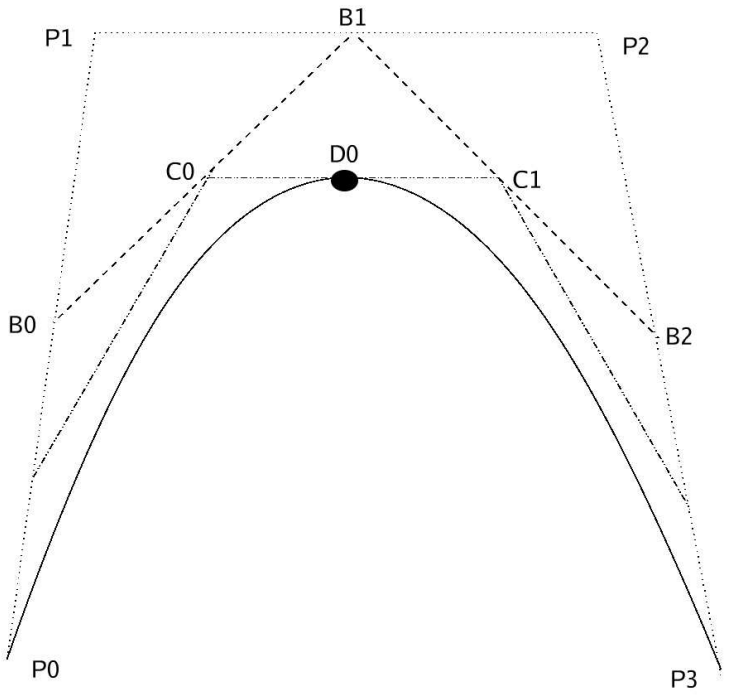


Figure 2.7: Illustration of de Casteljau's algorithm

Let $P_{0,j}$ be P_j for $j = 0, 1, \ldots, n$ where n is the degree of the curve, $i = 1, 2, \ldots, n$ and $j = 0, 1, \ldots, n - i$. Point $P_{i,j}$ is then computed as

$$P_{i,j} = (1 - u)P_{i-1,j} + uP_{i-1,j+1} \tag{2.8}$$

2.8 Real-time aspects

2.8.1 Computation

The Bezier curve is very cheap to evaluate even with a very small Δu[2]. The Δu need not be uniform, but can be increased on flat intervals with criteria from the control polygon.

The most common way to enhance a surface is to render it with different types of illumination or even ray-tracing, and these operations require a lot of processing time compared to the actual evaluation of surface coordinates and polygon tessellation. This huge topic is covered in many other reports and books [3][4], and does not get much room in this report.

[2]The difference between one point of evaluation and the next

2.8.2 Sculpting

A decent editor for Bezier curves and surfaces should allow options of degree and weight together with automatic C^1 continuous concatenation when adding new segments. In the case of surfaces we also need facilities for intuitive rotation and, as already mentioned, options for different types of tessellation, shading and other visual attributes.

In some modes of the thesis application, only the control polygon is drawn in real-time while manipulating the control points, since it requires too much computation to update the rendered surface continuously. This provides a rough sketch of the resulting object which is rendered once the updating stops. Another way to reduce the workload is to only render the object in its mesh form[3].

One can argue that a slow rendering frequency does not affect the actual curve or surface, which in fact it does not, but that is a topic of discussion elsewhere.

Another interesting aspect is that the computer screen is two-dimensional which complicates the editing of surfaces (or three-dimensional curves). We first have to project the curve or surface to screen space (two-dimensional), where we manipulate the control points with the mouse. Once done, we project the control points back to computation space (three-dimensional).

Of course, the surface can be rotated and hence manipulated implicitly in three dimensions, but this still poses a problem since it is very hard to determine the depth value without good visual depth cues[4].

2.9 Summary

To sum up this chapter, here are a few key properties of the Bezier curve that are good to remember in the chapters to come.

- Polynomial curve
- Evaluation by Bernstein polynomials, de Casteljau
- Number of control points and degree are dependent
- Convex hull property
- Interpolation of first and last control point
- Geometric operations on control points apply to the entire curve.

[3] No coloring or processing of polygons after tessellation

[4] Familiar object reference, shading, focus, blocking and so forth.

Chapter 3

B-spline curves and surfaces

In the last section we discovered some serious drawbacks with Bezier curves and surfaces

- No real local control
- Strict relation between curve degree and number of control points

These flaws are overcome with the B-spline curve and its bi-parametric extension, which this section will try to explain.

3.1 Comparison to Bezier curves

A B-spline curve of degree m with n control points consist of $n - m$ Bezier curve segments. These segments all have C^2 continuity at the join points. For instance, a cubic curve (degree 3) with 10 control points has 7 segments.

Any Bezier curve of arbitrary degree can be converted in to a B-spline (see section 3.4 on the basis function similarity) and any B-spline can be converted in to one or more Bezier curves.

In its unwound form, B-splines do not interpolate any of its control points, while the Bezier curve automatically clamps its endpoints. However, B-splines can be forced to interpolate any of its n control points without repeating it, which is not possible with the Bezier curve.

In general it can be stated that the B-spline curve requires more computation, but is far more flexible and pleasing to work with, which is the reason why it has become part of almost every serious graphics development environment.

The only real drawback compared to the Bezier curve is that the underlying mathematics can be quite troublesome and intimidating at first.

3.2 Knots

As explained in the previous section, a B-spline curve consists of segments. The join point between these segments are called *knots*, and play a fundamental role in the understanding of this kind of curve. In the cubic case we have $n + 4$ knot values and in the general case $n + m + 1$, which are commonly stored in a *knot vector*. This relationship will become clear over the next couple of sections.

The knot vector is used to specify values in the evaluation interval where the curve changes segment. By spacing the $n + m + 1$ intervals with equal distance we obtain a *uniform B-spline curve*, while an uneven spacing yields a *non-uniform B-spline curve*.

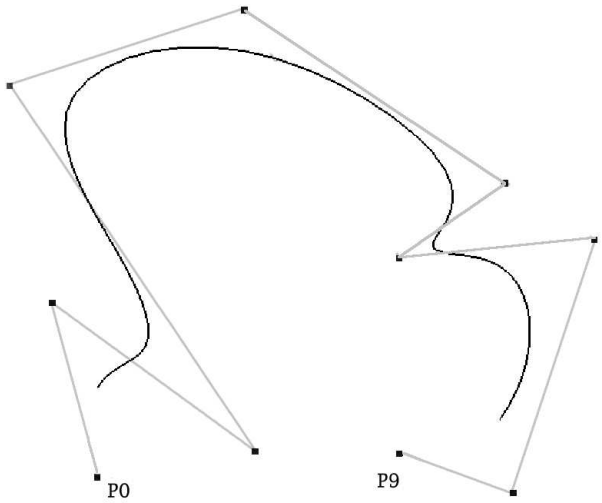


Figure 3.1: Uniform B-spline curve

Figure 3.1 shows a uniform B-spline curve of degree 3 and with 10 control points, hence we have 7 segments which are joined with C^2 continuity. The uniform part of this curve is that the distance between a knot value and the next is equal.

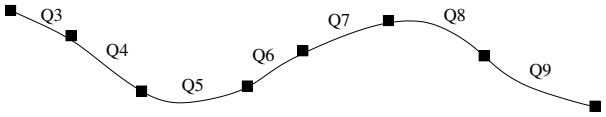


Figure 3.2: Knots and segments

Segment	Control points
3	0 1 2 3
4	1 2 3 4
5	2 3 4 5
6	3 4 5 6
7	4 5 6 7
8	5 6 7 8
9	6 7 8 9

In the table above we can see how the local control property is obtained with B-spline curves. The uniform knot vector for this curve is $[0, \frac{1}{14}, \frac{2}{14}, \ldots, \frac{13}{14}, 1]$. The reason why the segment numbering begins at 3 will become clear once we look at the basis functions and their evaluation in the next section.

By changing the values of the knot vector we make parametric segments shorter or longer, and this is one of the biggest advantages of the B-spline curve (see figure 3.2).

The easiest way to demonstrate this technique is to "clamp" the endpoints by changing the knot vector to $[0, 0, 0, 0, \frac{4}{14}, \frac{5}{14}, \ldots, \frac{10}{14}, 1, 1, 1, 1]$. This transforms the first and last segments to single points which forces the curve to interpolate the endpoints (figure 3.3)

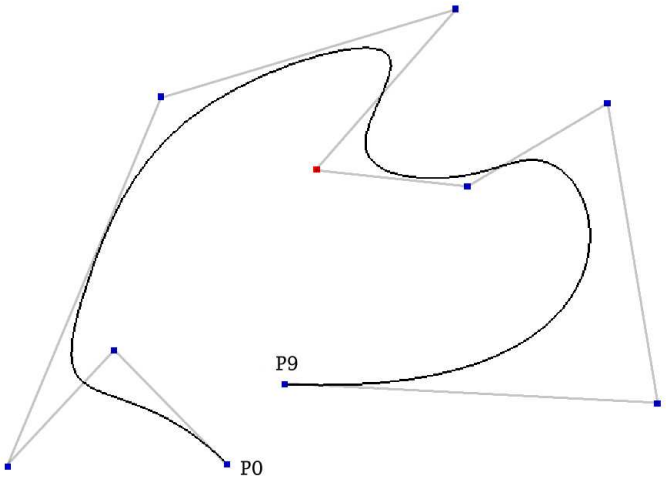


Figure 3.3: Non-uniform B-spline curve with clamped endpoints

This is the most common way to manipulate the knot vector, but of course we can clamp any point $p \in (0, 1)$. By doing this, we create a discontinuity at that particular point.

In this example, a double knot reduces the continuity to C^1 and a triple knot to C^0. The number of repeated knot values are often referred to as the

multiplicity of the knot.

Here we encounter another clever feature of the B-spline curve; several disjoint segments can be made from one knot vector and a set of control points by introducing discontinuities.

3.3 B-spline basis functions

When constructing a uniform B-spline curve the basis functions are translates of each other, yielding a very simple basis function. To complicate things we have to consider the non-uniform case, where the knot values are not equally distant from one another and this is not a very simple task.

Below we find the Cox de Boor algorithm, which can recursively compute the basis functions to any uniform or non-uniform B-spline curve of degree n. The knot values with corresponding index is denoted t_i and i is the control point to evaluate.

$B_{i,1}(t) = 1.0$ if $t_i \leq t \leq t_{i+1}$, else 0.0

$B_{i,2}(t) = \frac{t-t_i}{t_{i+1}-t_i} B_{i,1}(t) + \frac{t_{i+2}-t}{t_{i+2}-t_{i+1}} B_{i+1,1}(t)$

$B_{i,n}(t) = \frac{t-t_i}{t_{i+n-1}-t_i} B_{i,n-1}(t) + \frac{t_{i+n}-t}{t_{i+n}-t_{i+1}} B_{i+1,n-1}(t)$

This algorithm is similar to the de Casteljau algorithm covered in chapter 1 but is more general since we can keep the abstraction level on the entire curve instead of individual segments. Here we also see the need for having $n + m + 1$ control points.

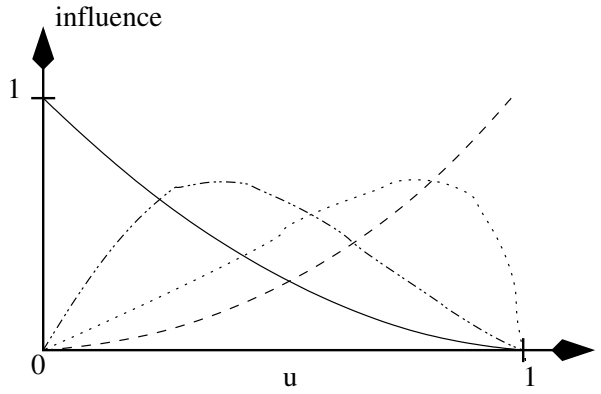


Figure 3.4: Cubic with clamped end points, Cox de Boor

By clamping the knot vector on a curve with degree 3 we obtain the same graph of influence as we did with the cubic Bezier basis function. If we have

greater multiplicity on some knot elsewhere, the graph will peak there.

3.4 B-spline formulation

There are a couple of different ways to describe a B-spline curve. In its most basic form it looks just like the Bezier formulation.

$$W^m(u) = \sum_{i=0}^{n} P_i B_{i,m}(u) \tag{3.1}$$

where P is the control point vector, B the basis function, n the number of control points and m the degree of the curve. Equation 3.1 is simple to understand, and uses the fact the B will automatically return 0 if the control point being evaluated is "out of reach" of u.

This is however not very efficient since we for every u must evaluate the basis function for every P_i. In the cubic case we only need to know $m+1 = 4$ relevant control points since an arbitrary point u is never affected by more than so many. This topic will be expanded on in coming sections of the report.

3.4.1 NURBS

The term NURBS stands for *Non-Uniform Rational B-Spline* and is the most common way to model curves and surfaces in computer graphics [5], hence it deserves a topic by itself in this report.

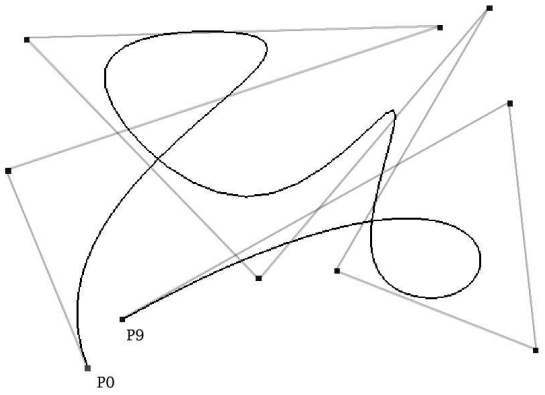


Figure 3.5: NURBS curve, $w_i = 1.0$ for all i

Just as with Bezier curves, a rational curve means that we extend the control points by one coordinate that will represent the weight, or attraction/repellation, of that particular control point. In figure 3.5 we see a NURBS curve where all control points are assigned weight 1.0, which is the default "weight-less" value.

This can be seen by inspection of the NURBS formulation (equation 3.2) as $w_i = 1.0$ does not modify any calculations or results.

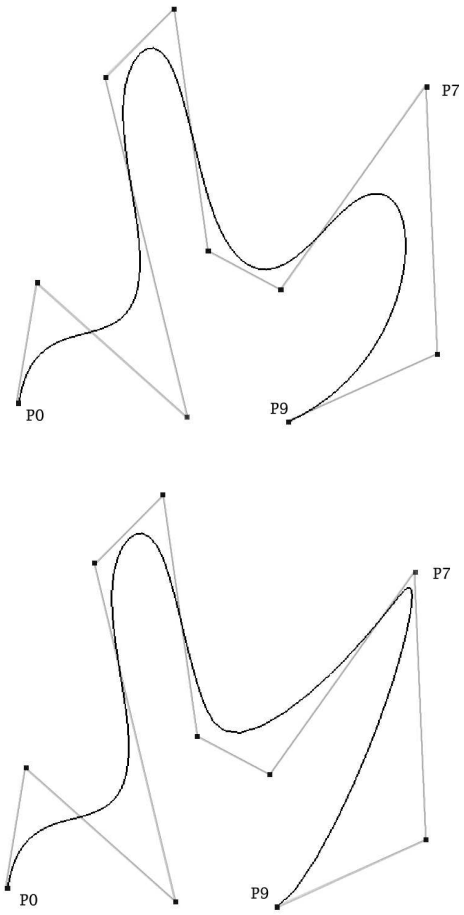


Figure 3.6: NURBS curves, $w_7 = 1.0$, $w_7 = 15.0$ respectively

In figure 3.6 we see what happens when we increase or decrease the weight. If we increase the weight enough, the curve will interpolate and eventually pass through P_7.

It should also be noted that an increase or decrease in w_i does not affect the C^2 continuity, though it may introduce some sharp corners when viewed

with poor detail or a large Δu.

This type of rational extension can cause the curve to violate the convex hull property[1], which may be of computational interest in collision detection, for instance. This is computationally handled so that the weight can only span, say, $(0.8, 20.0)$

Another interesting fact is that around $w_i = 20.0$ the curve interpolates P_i, which is perhaps more intuitive than to modify the knot vector for interpolation. The interested reader should observe *Phantom vertices*, discussed in [11], which is another method of forcing vertex interpolation.

$$W^m(u) = \frac{\sum_{i=0}^{n} w_i P_i B_{i,m}(u)}{\sum_{i=0}^{n} w_i B_{i,m}(u)} \tag{3.2}$$

The rational mathematic extension is analogous to that of Bezier curves and produces no computational difficulties, and in equation 3.2 we see the famous NURBS formulation which probably is the most popular curve in computer graphics to this date.

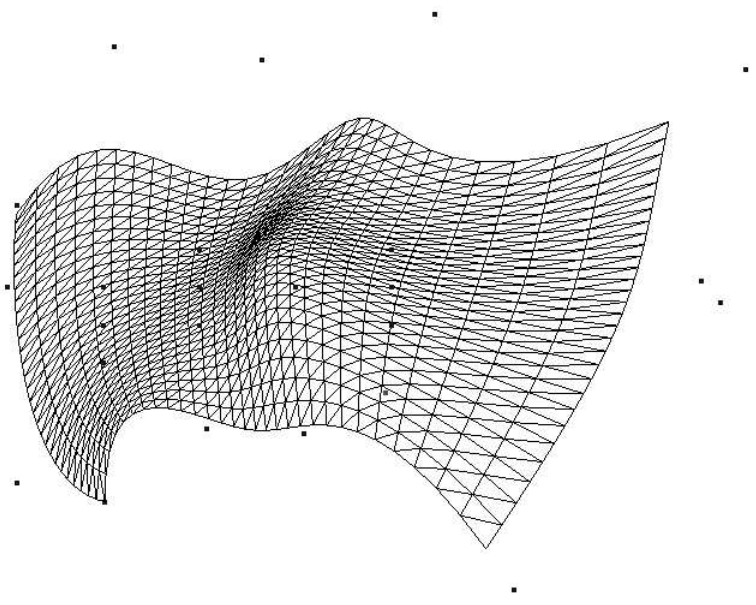


Figure 3.7: NURBS surface, 6x4 control points

[1]The curve should lie within the area enclosed by all control points

3.5 NURBS surfaces

If we extend equation 3.2 in two parametric directions we obtain a surface with the same properties as the NURBS curve. This equation is not pleasant to the eye in its mathematical form, but in its rendered.

$$W^m(u,v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{t} w_i P_i B_{i,m}(u) B_{j,m}(v)}{\sum_{i=0}^{n}\sum_{j=0}^{t} w_i B_{i,m}(u) B_{j,m}(v)} \tag{3.3}$$

The surface does not have to be of equal degree in both directions, but to keep things more understandable we settle for this now. Observe the surface in its rendered form in figure 3.7 where we clearly see the local control property.

By evaluating this equation we obtain a surface with local control, one knot vector in each parametric direction and with rational properties; virtually anything one could wish for when modeling a surface.

It goes without saying that $B_{i,m}(u)$ uses the knot vector u and $B_{j,m}(v)$ the knot vector v.

3.6 The de Boor algorithm

For the B-spline curve we also have a geometrical way to display the calculations, which is an extension of the de Casteljau algorithm covered in the previous chapter for Bezier curves. It is called the de Boor algorithm, and should not be confused with the Cox de Boor algorithm described earlier.

The only difference is that we for a point u must determine which $m+1$ control points that affect it, and there after apply the de Casteljau algorithm to that segment. See figure 2.7 for the geometrical interpretation.

This is not a hard operation since we have a co-relation between the knot and control point vectors.

In figure 3.8 we see how the de Boor algorithm would evaluate $u = 0.4$. Assume we have a knot vector $[0, 0, 0, 0, 0.25, 0.5, 0.75, 1, 1, 1, 1]$ and seven control points. In the figure we see the control polygon of these.

The point $u = 0.4$ lies in $[0.25, 0.5]$ and hence the control points affected are $P1, P2, P3, P4$. We remind ourselves that the curve formed by these points is a Bezier segment and apply the de Casteljau algorithm to calculate the point.

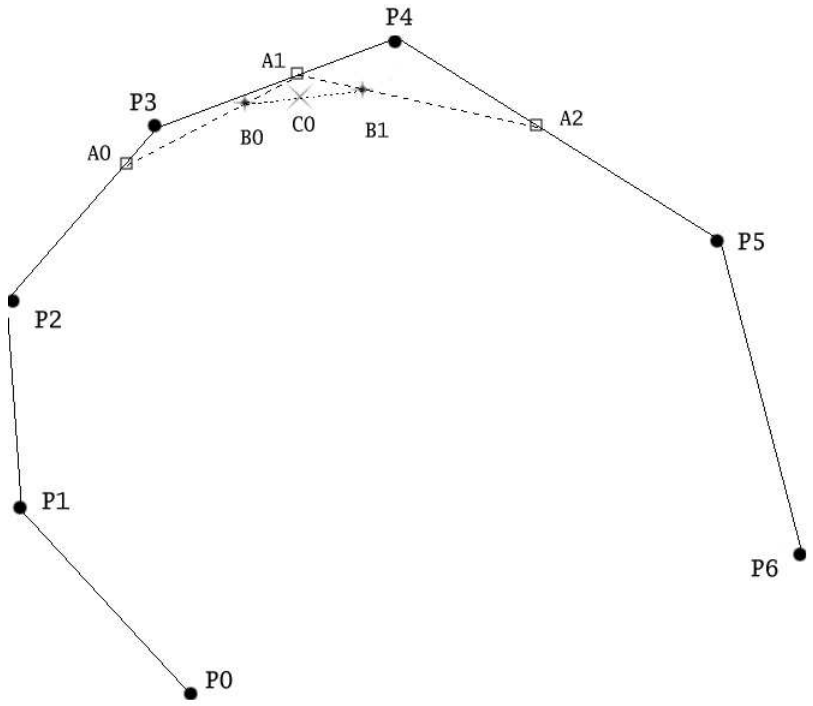


Figure 3.8: de Boor algorithm, cubic curve

Since the de Boor algorithm is very similar to the de Casteljau algorithm, we can in analogy with equation 2.8 also express it as follows:

For every segment S, let $P_{0,j}$ be P_j for $j = 0, 1, \ldots, n$ where n is the degree of the curve (and hence, the degree of an individual segment), $i = 1, 2, \ldots, n$ and $j = 0, 1, \ldots, n-i$. Point $P_{i,j}$ on segment S is then computed as

$$P_{i,j} = (1-u)P_{i-1,j} + uP_{i-1,j+1} \tag{3.4}$$

3.7 Real-time aspects

3.7.1 Computation

As mentioned before, a B-spline curve or surface can be quite expensive to calculate, especially in the non-uniform case. One method that provides faster calculations is if we force the knot values to $u_i - u_{i+1} = 1$ or $u_i - u_{i+1} = 0$. This way the Cox de Boor algorithm can be computed offline [2] and stored in matrices for fast evaluation, hence it must not be evaluated for every curve segment unless we change the knot vector.

[2] Before visual execution

In section 3.4 we briefly discussed the fact that each segment only requires input from $m+1$ control points, four in the case of a cubic curve. By evaluating the curve through *segments* we can avoid to include every control point in the calculations.

Say we have knot vector t and wish to evaluate a cubic B-spline with a control point vector P. We denote the basis function B with arguments degree, control point and parameter value, in that order.

```
for i:=3 to sizeof(P):
    for u:=t[i] to t[i+1]:
        result := 0
        result += P[i-3] * B(3,i-3,u)
        result += P[i-2] * B(3,i-2,u)
        result += P[i-1] * B(3,i-1,u)
        result += P[i-0] * B(3,i-0,u)
        plot result
    end for
end for
```

This is pseudo-code of the main loop of a B-spline curve evaluation, and it shows how we can avoid unnecessary computation.

Since the B-spline curve can be represented in matrix form, it can also be wise to take advantage of fast matrix multiplication algorithms, or use software like Matlab or Maple as modules for the multiplications.

As mentioned in the previous chapter we can vary Δu with B-splines as well, by inspecting the slope of P_i and P_{i+1}. A flat section of the curve does not need the same resolution as a heavily curved one.

3.7.2 Sculpting

Most properties from Bezier sculpting carry over to the B-spline. The new thing that we have to consider is the editing of the knot vector(s).

If the GUI (Graphical User Interface) supports input of vectors or strings, it is quite common to display an editable strip of text where the knot values can be edited with the keyboard.

A more refined method is to project the knot values to a line or plane that lies below the curve or surface, where the user can manipulate the values by dragging the projected icons in the fixed parametric directions, which is perhaps the most intuitive way to modify the knot vectors.

Knot multiplicity is also a detail to consider, as the software must allow insertions of several knots at the same coordinate

3.8 Knot insertion and deletion

If the B-spline curve is not flexible enough we resolve to a technique known as knot insertion. Since the size of knot vector t and control point vector P is related, the idea behind this concept is to add one or more control points without modifying the shape of the curve, thereby obtaining a knot vector of greater size and flexibility. The reason for this is the fundamental property that $m = n + p + 1$, where m and n are the sizes of the control point vector and knot vector respectively, and p is the degree. As mentioned earlier, a cubic curve with 7 control points has $3+7+1 = 11$ knot values, for instance.

Assume for the following knot insertion that we have our original control points $P_0, P_1, \ldots, P_n$ and wish to find $Q_0, Q_1, \ldots, Q_n, Q_{n+1}$ which produces the same curve of degree p.

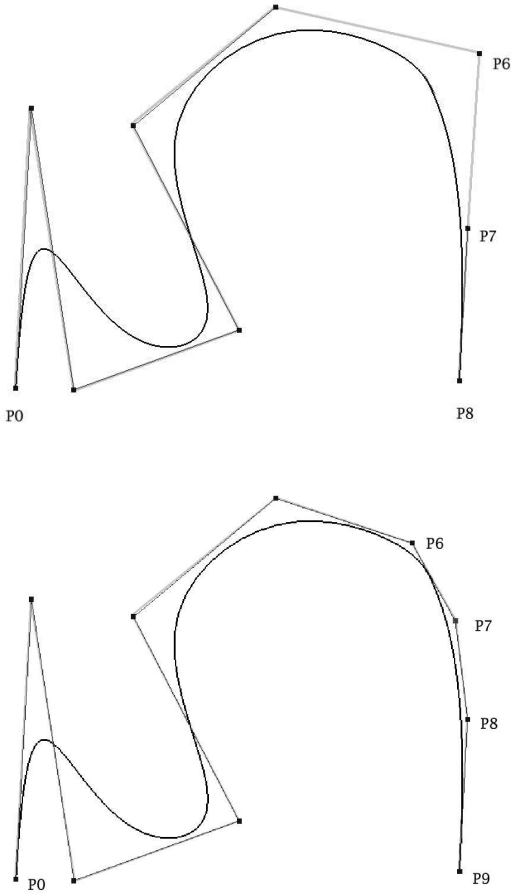


Figure 3.9: Before and after knot insertion

To the knot vector T we wish to insert a knot at u, where the value of u is in the knot vector interval $[T_k, T_{k+1}]$. Since only a part of the curve is affected by this operation, we only need to consider control points $P_k, P_{k-1}, \ldots, P_{k-p}$ which are the ones that are involved in the calculation of $[T_k, T_{k+1}]$.

Now that we have the necessary facts we can calculate point Q_i which is in between original control points P_{i-1} and P_i by

$$Q_i = (1 - a_i)P_{i-1} + a_i P_i \text{ , where}$$

$$a_i = \frac{u - u_i}{u_{i+p} - u_i} \text{ for } k - p + 1 \leq i \leq k$$

which indeed is a very simple and fast calculation. Notice the similarities to the de Casteljau and de Boor algorithms. Geometrically speaking, the control polygon is unaffected over the entire curve except between $P_k, P_{k-1}, \ldots, P_{k-p}$. This method naturally works for inserting knots at existing values with any multiplicity, but this is more efficiently done with the Oslo algorithm [6] which can insert several knots in one operation.

The reverse operation, the deletion of a knot, requires deletion of a control point and hence it does not preserve the shape of the curve. See [7] for further reading.

In figure 3.9 we see how the exact same curve can be represented by a different amount of control points. Here we have inserted a knot at $u = 0.8$, on the right half of the curve. This will provide us with greater local control, and can be repeated if necessary.

3.9 Concatenation

There are algorithms for merging both B-spline curves and patches. The simplest way to a accomplish a concatenation is to simply create a new larger curve or patch which contain all the control points, but since we in most cases work with clamped end points we can apply the same technique as shown for Bezier curves in the previous chapter. That is, to introduce a continuity constraint between the last and first couple of points on each curve.

3.10 Other spline forms

So far we have covered a few of the most common types of curves and surfaces. In this section we look at some other useful types that aid us in common computer graphics tasks.

3.10.1 Natural spline

A natural spline is the real-life model of the spline curves used in computer graphics. They are strips of flexible metal (aluminum, for instance) that are C^2 continuous, but they do not posses local control. Moving one control point affects the entire curve.

3.10.2 Hermite curves

The Hermite curve, named after Charles Hermite, is similar to the Bezier curve where we specify two endpoints and two other points to determine the tangents and these endpoints. However, it uses a different set of basis functions and behaves differently, see figure 3.10. It also interpolates all of its control points, but does not have a convex hull property.

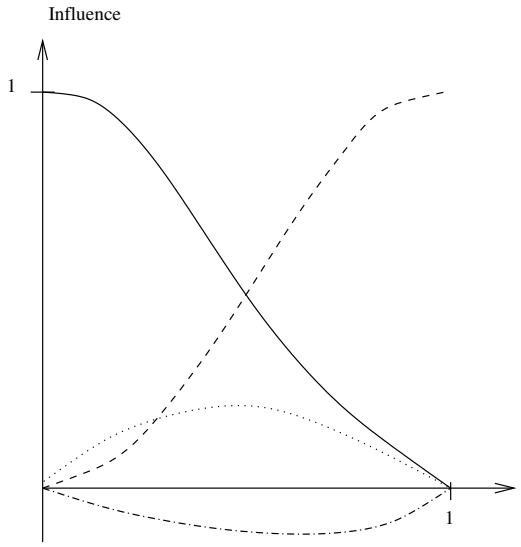


Figure 3.10: Cubic Hermite basis functions

3.10.3 Catmull-Rom splines

The Catmull-Rom splines [12], also called Overhauser splines [13], is a set of splines that interpolate all or most of their control points, hence this is a very common type of parametric curve that we use to model exact representations.

The downside to this type of parametric curve is that it does not posses the convex hull property, which complicates some applications (collision detection, for instance).

3.10.4 Uniformly shaped β-spline

This type of curve works like a B-spline curve but with two new variables in the calculations called *bias* and *tension*. In addition to the rational extension, we now have two further "weights" to modify our curve which provide a higher degree of flexibility. It should be noted that these two parameters are not exclusive to each control points, but are defined globally. The curve was presented by Barsky and Bartel [10][11].

The only problem with this spline is that it is only C^0 continuous at the segment join points (knots). This complicates some of the applications involving movement discussed in the next chapter. As with the Catmull-Rom spline, the β-spline does not have a convex hull property, neither does it have local control.

3.11 B-spline summary

Presented below are a few key terms to remember regarding B-spline curves and their use.

- Local control
- No relation between degree and number of control points
- Knot insertion and the knot vector
- Uniformity and non-uniformity
- de Boor algorithm
- Cox de Boor algorithm
- NURBS

Chapter 4

Applications

The need for curves and surfaces is very obvious, and here we will cover at least some areas of application where they are used extensively.

4.1 Camera movement

One of the first areas in 3D computer graphics to be exploited by polynomial curves was the non-linear movement of a camera or object through a virtual environment. If we create a linear path between some key positions the camera will be jerky in its movements. By interpolating the path with a spline curve we obtain a smooth curve which fits the task perfectly.

Within the spline data-type we can also store rotation, acceleration and other properties at certain key points, still with very small storage requirements.

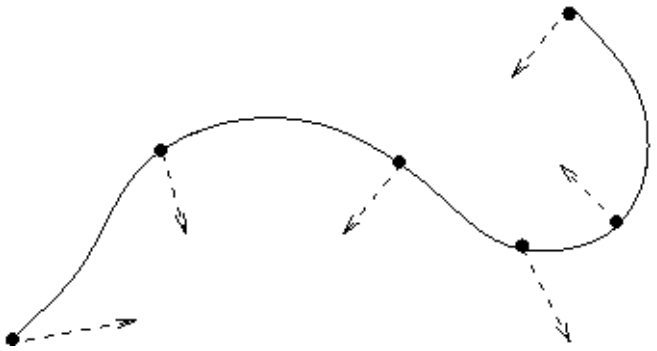


Figure 4.1: Key rotations on a spline

Above we have a spline with certain key positions that contain a vector of direction. In an off-line operation we interpolate between these key rotations by, for instance, spherical interpolation, and simply store the results along

the curve. In these key positions it is also common to store an increase or decrease in acceleration, camera zoom, lens properties and so forth.

Put shortly, the spline curve is a necessary tool for creating nice camera movements without having to do frame-by-frame animation.

4.2 Topologies

In visualization software we often need to model topologies of different types. In computer games we crave natural-looking bodies, faces and level surfaces for our characters to travel on. This is not seldom done by using NURBS surfaces, which provide excellent control both for smooth and sharp surfaces.

Here we can benefit from the non-discrete property that allows for different resolutions depending on the application. We can compute a polygon mesh of rough size for the physics while computing a finer resolution for the visual output by using the same set of control points.

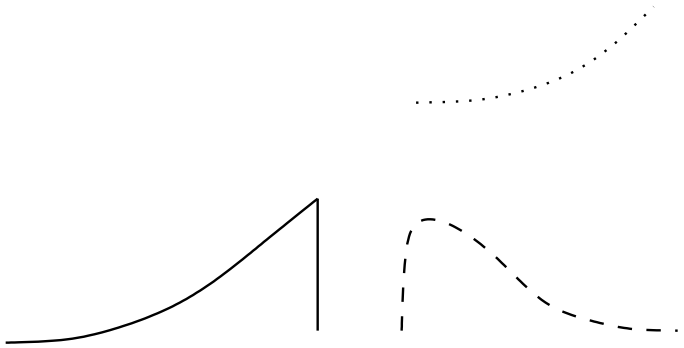


Figure 4.2: Game level side-view

Figure 4.2 shows how we can create a simple jump in a racing game by using disjoint segments in one spline (done through the knot vector) or by using several splines.

This method of level creation is extremely useful when modeling a virtual version of a real scenario. Prior to implementation we can measure the topology of the real racing track (or whatever it is we model), scale these points to the virtual world and use them as control points to a NURBS surface.

Another common method for building tracks is to use a NURBS curve and to specify tilt, yaw and pitch[1] at the key points (see figure 4.1).

Both of the explained methods provide very intuitive sculpting, both for developers and end users wanting to create their own levels. Thus it is

[1]Common descriptors for specifying rotation and orientation

not hard to realize why the NURBS curve have become so popular in game design over the last couple of decades.

Speaking of surface fitting, this is also a much used model for creating virtual versions of real characters. We scan the topology of the human face and adapt the results as control points in a NURBS surface, but often this needs "tweaking" since the surface does not interpolate all of its control points.

4.3 Character movement

If we gaze back at the amusement applications developed during the early 1980's we often see that characters move in straight lines, or even in fixed directions (up, down left, right). As soon as the processing power of computers started to elevate, developers began to use curve interpolation and approximation to get smooth movements.

If we combine key points on NURBS curves, good-looking rotational interpolation (spherical interpolation, for instance), inverse kinematics[2] and random movements (behavioral noise) we basically have today's concept for creating decent character translocation.

Specifying paths of movement through splines have several advantages. If the spline passes through an obstacle we modify the curve to obtain a smooth motion around it, a very fast operation which can be done in real-time. For instance, if we want an object to move between point A and B we first create a straight line (spline) and try it for intersection with other objects in the scene, which is a cheap operation (see chapter 4.4). If it collides with anything we manipulate the curve.

In racing games we perhaps have the last 3-4 recorded locations of an adversary player. Once we receive a new location (control point) we create a spline path with these points and obtain a smooth non-linear motion.

4.4 Collision detection

The basic problem behind the topic *collision detection* is to determine when objects collide, with as little processing power as possible, while maintaining a solid behavior.

This is a huge topic by itself and we will concentrate on the advantages of using parametric representations for curves and surfaces and how to use them in an interactive application.

To start things off, we have a couple of primitives that are used extensively in collision detection. These consist of planes, spheres, lines and so forth. The idea is to encapsulate or compose an object with these primitives

[2]Popular way to animate movement through bone structure

so that an intersection between two primitives can be determined quickly. There are rapid algorithms for most combinations such as sphere-line, box-plane and plane-line which we will use to make a *bounding primitive collision detection* between objects.

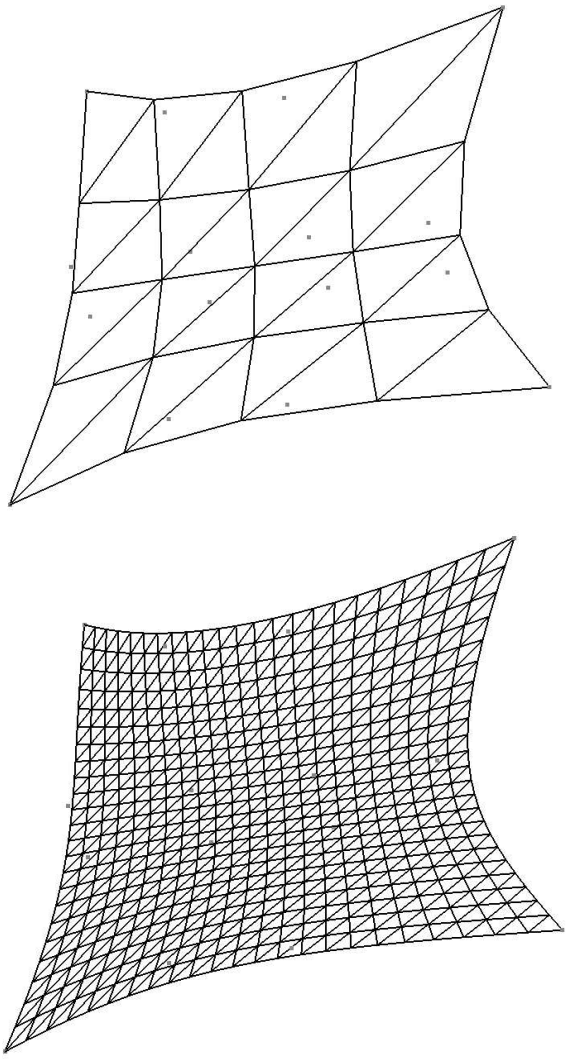


Figure 4.3: Different resolutions

In an off-line process we read the control points and create two versions of the surface, one with the visual output resolution and one with resolution suited for the collision detection engine. In figure 4.3 the control points remain in the same position but we tessellate the surface at different resolution.

This tactic is often combined with a tree structure. In this tree we break

the surface in to several hierarchic bounding volumes where every vertex except root and leafs have volumes as parent and children. See picture 4.4 below.

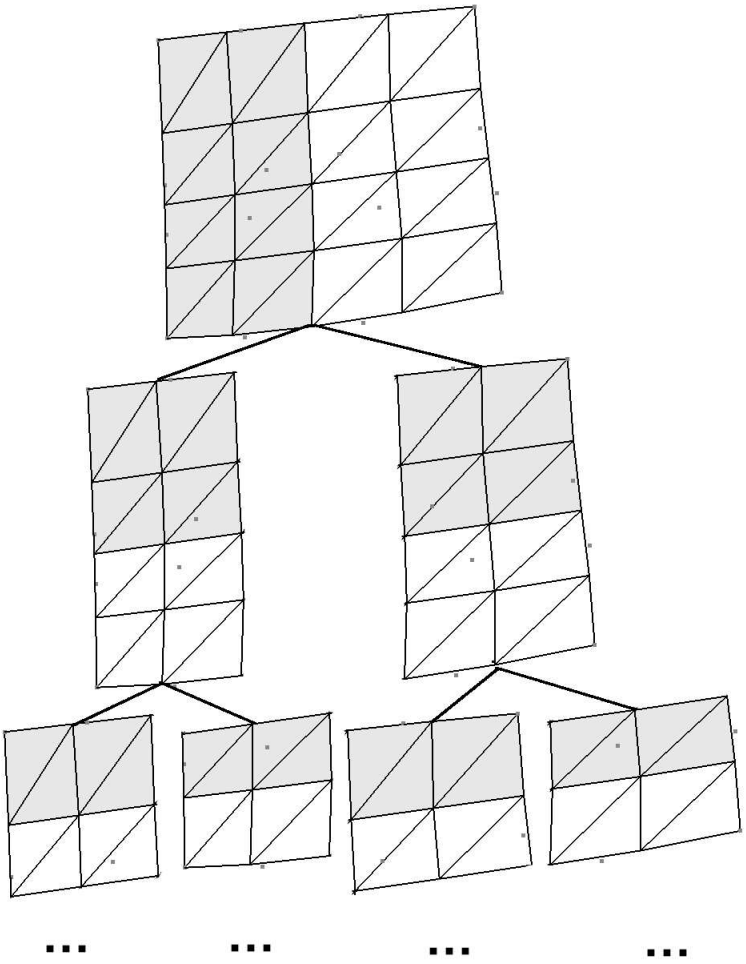


Figure 4.4: Bounding volumes

This is a classical technique for eliminating the need for unnecessary calculations. First we try the collision with the top volume. If it does not intersect, we can abort with only a few calculations "wasted". If it does, we traverse down the tree and repeat the process.

This tree can be computed off-line and the only difficulty is to anticipate the depth of the tree, which is very dependent on the application. If this surface is a flag that we drive by at high speed, we may only need the two top levels. One the other hand, it could be part of something that is closely

inspected by the user, say a piece of paper with clues in an adventure game.

In the latter case we usually want to make the volumes such that grouped polygons lie approximately in the same plane or box, so that we can use documented intersection algorithms that solve very quickly.

One should observe the discussion in earlier topics about using adaptive subdivision, but now applied to the low-resolution surface. By using this technique wisely and checking for flatness criteria we can save tremendous amounts of processing power for other important tasks. Since half of the game developer's work consists of clever tricks and cheats for greater performance, we often trade a little bit of accuracy for fewer collision computations, in a case like this.

This tree-structure technique is of course also applicable to curves where we create bounding boxes to snare smaller and smaller parts of the curve.

4.5 Soft objects

A very tricky subject in the realms of physics and computer graphics is to model good-looking cloth, blankets and other soft deformable objects. Often we get a good result by suspending a surface with a number of springs that we use to calculate the resulting forces. These forces can then be applied to the surface control points. In figure 4.5 a spring is attached to every control point.

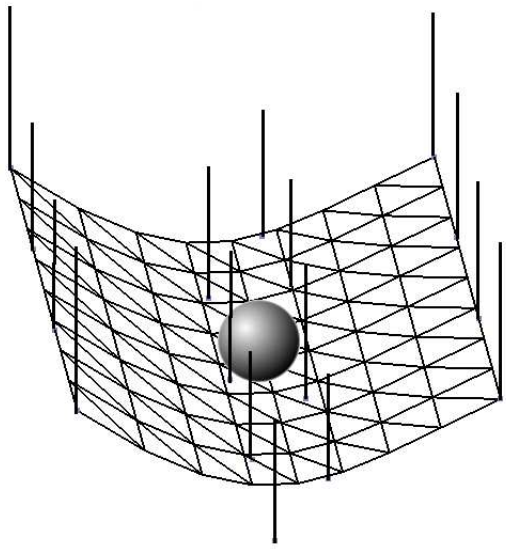


Figure 4.5: Bezier surface with springs

4.6 Automatic LOD

LOD (Level Of Detail) determines how detailed a particular object is. An oft-used trick that developers use is to render several versions of an object off-line. This object can be a picture, model, surface or anything with a polygonal representation. As the camera moves closer to the object, we switch between different levels of LOD. A close-up of the object may contain, say, 500 polygons while a very distant view only contains five.

This is very similar to the technique described in section 4.4, see figure 4.6 below.

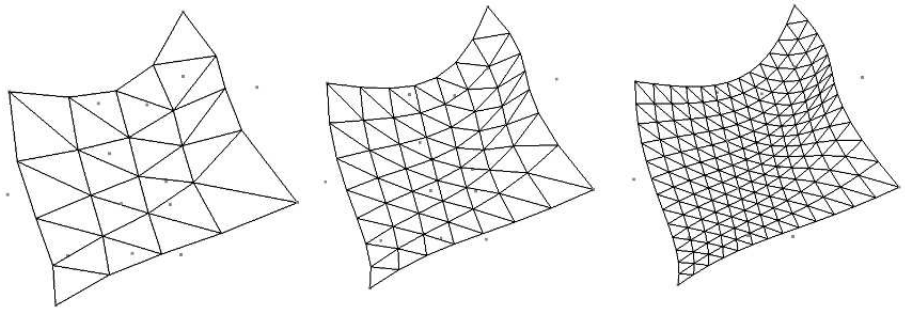


Figure 4.6: Bezier patch, different LOD

There are several topics to expand on regarding LOD computations, and these concern how the transition between different resolutions is made. The important aspect here is that we acknowledge the power of a parametric representation.

However, if we do not have a parametric representation available, there are other techniques generally known as *imploding*, where we reduce the number of polygons by some set of rules involving the distance to the viewer.

4.7 Object modeling

One last topic that should be mentioned is the ability to create in-game objects, which is a very common application. This is often done in 3D modeling software but can be done within the executing applications as well.

Consider the look of a curved door passage in an ordinary computer game. There is a large possibility that the frame of this door is done by a U-shaped spline that is tessellated with a certain thickness and then textured.

Another great example are vines, which must be the most suited thing to model with a spline or Bezier curve, since they by definition respond very similar to their real aliases.

A very famous example of the power of parametric surfaces is the "Utah teapot" which is, like the name implies, a virtual model of a teapot and was one of the first real objects to be replicated with surface patches. By using a few Bezier- or even fewer B-spline surfaces we can construct a very realistic teapot. Consider the polygonal mesh version that requires at least a couple of hundred definition points while the parametric version does the same job with considerably fewer points (say, 40-50). Add to that the non-discrete property that allows us to change the resolution just by changing the evaluation interval.

Chapter 5

Implementation

This chapter will deal with topics that emerged while developing the thesis application and some interesting areas of research that are connected to the actual implementation.

5.1 Simple evaluation

So far we have described the evaluation of curves and surfaces as a "brute force" task where we set an interval Δu and evaluate the curve accordingly. This is common but not very efficient since the shape varies over the parameter interval. It may be completely flat, only to make a short peak somewhere, and that is a waste of processor time.

5.2 Forward difference

Most computer scientists and mathematicians have encountered the evaluation method *forward difference*, which is one way to adapt Δu according to some given criteria, in this case flatness of the curve.

The forward difference $\Delta f(t)$ of the function $f(t)$ is given by

$$\Delta f(t) = f(t + \delta) - f(t) \tag{5.1}$$

This we can write as

$$f_{n+1} = f_n + \Delta f_n \tag{5.2}$$

where f is evaluated a uniform interval with size δ. For a cubic polynomial, which is the most common degree for parametric curves and surfaces, we have

$$f(t) = at^3 + bt^2 + ct + d \tag{5.3}$$

so the forward difference becomes

$$\Delta f(t) = a(t+\delta)^3 + b(t+\delta)^2 + c(t+\delta) + d - (at^3 + bt^2 + ct + d) \quad (5.4)$$

To further ease the computation we can now apply the same technique to $\Delta f(t)$. Equation 5.2 and 5.4 now gives us

$$\Delta^2 f(t) = \Delta f(t+\delta) - \Delta f(t) \quad (5.5)$$

$$\Delta^2 f(t) = 6a\delta^2 t + 6a\delta^3 + sb\delta^2 \quad (5.6)$$

This yields

$$\Delta^2 f_n = \Delta f_{n+1} - \Delta f_n \quad (5.7)$$

or similarly

$$\Delta f_n = \Delta f_{n-1} + \Delta^2 f_{n-1} \quad (5.8)$$

Since we in this example are working with cubic polynomials the last and third forward difference will be a constant, and we calculate it just as shown above. In summary we know have

$$f_{n+1} = f_n + \Delta f_n$$

$$\Delta f_n = \Delta f_n n - 1 + \Delta^2 f_{n-1}$$

$$\Delta^2 f_{n-1} = \Delta^2 f_{n-2} + 6a\delta^3$$

This may seem a bit much just for calculating a line segment, but we have now reduced the calculations per 3D point from ten additions and nine multiplications to just nine additions, if we use the Bezier curve as an example. Though, we have to calculate the initial conditions when using forward difference, but this only has to be done once.

5.3 Recursive subdivision

Another method for improved efficiency is to use a recursive subdivision technique as suggested in earlier chapters of the report. This is best explained in pseudo-code.

```
Draw(CURVE c, ERROR e)
   if(Straight(c,e))
      then DrawLine(curve)
   else
```

```
      SubdivideCurve(c, leftCurve, rightCurve)
      draw(leftCurve,e)
      draw(rightCurve,e)
   end if
end draw
```

This Divide-and-Conquer algorithm is simple to implement and halts when the divided curve segment (if any) is flat enough to be drawn as a straight line. This test can be, for instance, the quadratic area enclosed by the curve end points, the area beneath it or the angle between the end points.

5.4 Look-up tables

Look-up tables are often used when we know the evaluation resolution of a function. In games we most often create them for trigonometric functions, but we can use them for parametric curves and surfaces as well. For each control point, we build a table that holds indices to the basis function return value at each interval. When we show or manipulate the surface in real-time, we never have to access the cumbersome weighting functions.

As the tables can be come quite large, the trade-off associated with this trick is storage size. Usually, processor time is of more value than the memory usage, so it is very common that we implement this feature.

5.5 Portability

As claimed earlier in the report, portability is one of the strongest features of parametric curves and the algorithms that deal with them. If we combine this fact with an object-oriented approach we get a class that can be very general. Consider the class diagram[1] figure below which is based on the Fly3D graphical engine.

[1] Common way for displaying class hierarchy and content

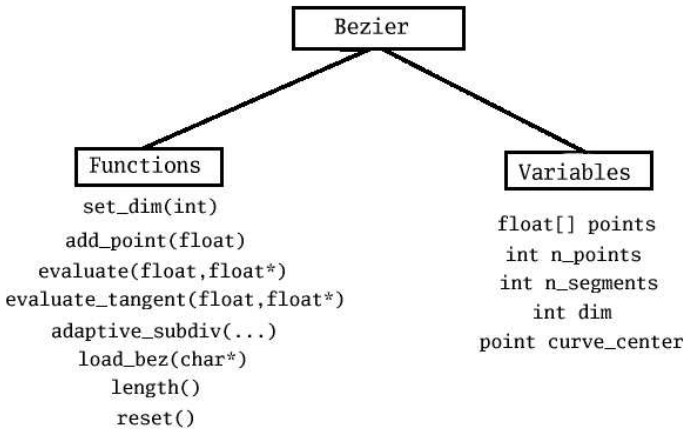


Figure 5.1: Bezier class diagram

This is a basic class that holds all the algorithms and functions needed for a Bezier curve evaluation. All functions are platform independent, with a minor exception for the load_bez function because file reading can differ on some platforms. What each function does and what each variable holds should be of no difficulty to understand.

Similar classes are easily done for B-spline curves and surfaces, with addition of the knot vector operations and algorithms. The only thing needed to display (if that is our goal) the curve is a plotting function, which often varies over different platforms.

5.6 Implementation comparison

In this section we shall look at a few comparisons between different implementations and how well they scale. The test involves the calculation of a typical curve with different resolutions, and the time is measured with a system profiling tool.

It should be noted that the tests use a system-wide clock, which is not devoted to one process only. The values are averages of three separate sessions.

Following each chart is a short text that discusses the differences obtained and why they occurred.

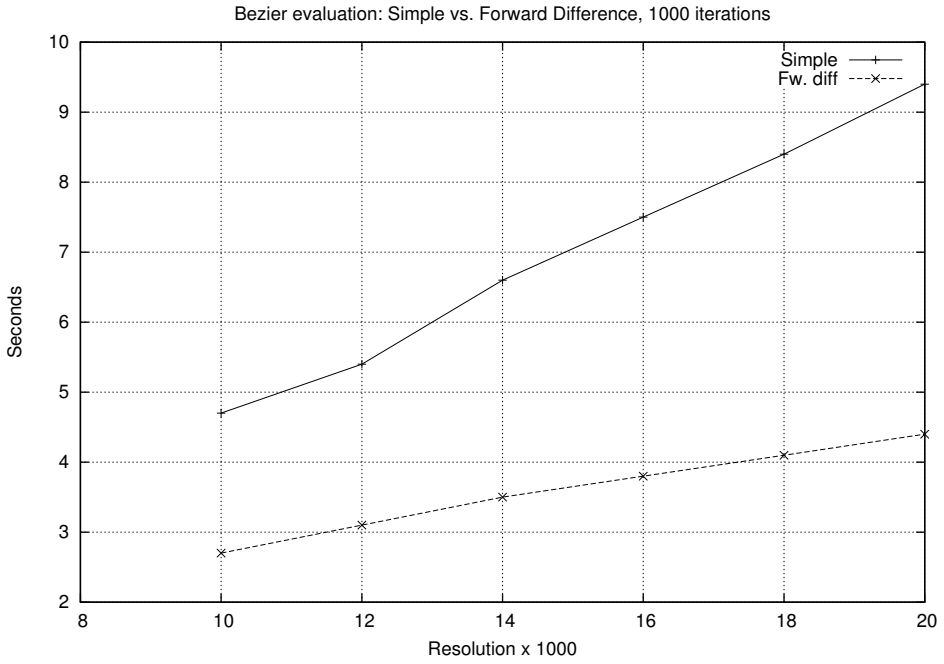


Figure 5.2: Simple evaluation vs. Forward difference

As explained earlier in the report, the forward difference method reduces the amount of calculations needed per 3D point from ten additions and nine multiplications to just nine additions. This can be clearly seen in figure 5.2.

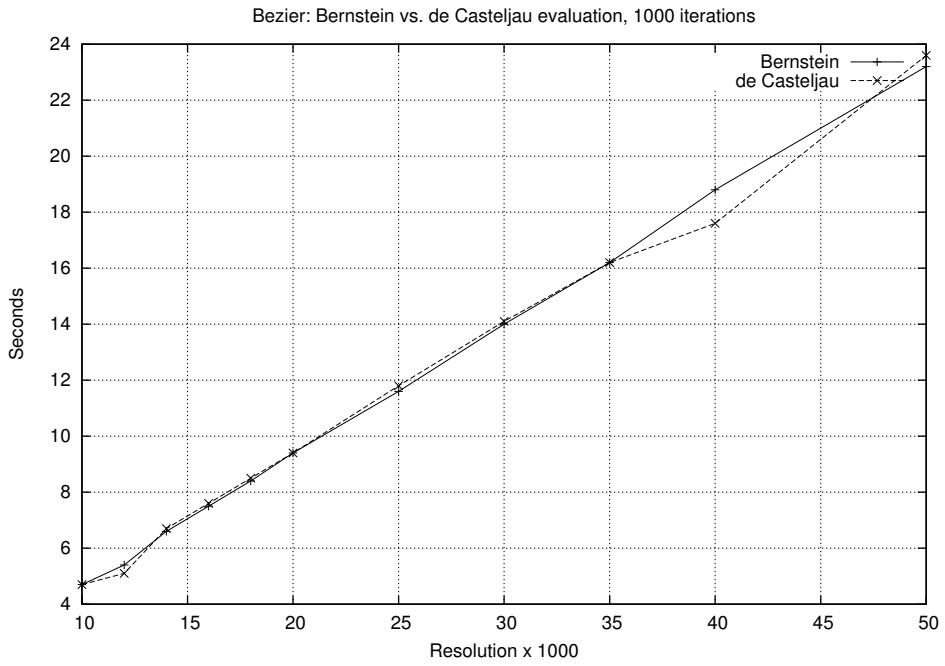


Figure 5.3: Bernstein polynomials vs. de Casteljau

The only difference between these two is the theoretical view; the de Casteljau algorithm is interpreted geometrically while the Bernstein polynomials are handled analytically. They contain the same number of multiplications, compute the same points, but as mentioned in the introduction their history is quite different. In the end, it comes down to which is more easy to implement. The small local differences are visible in figure 5.3.

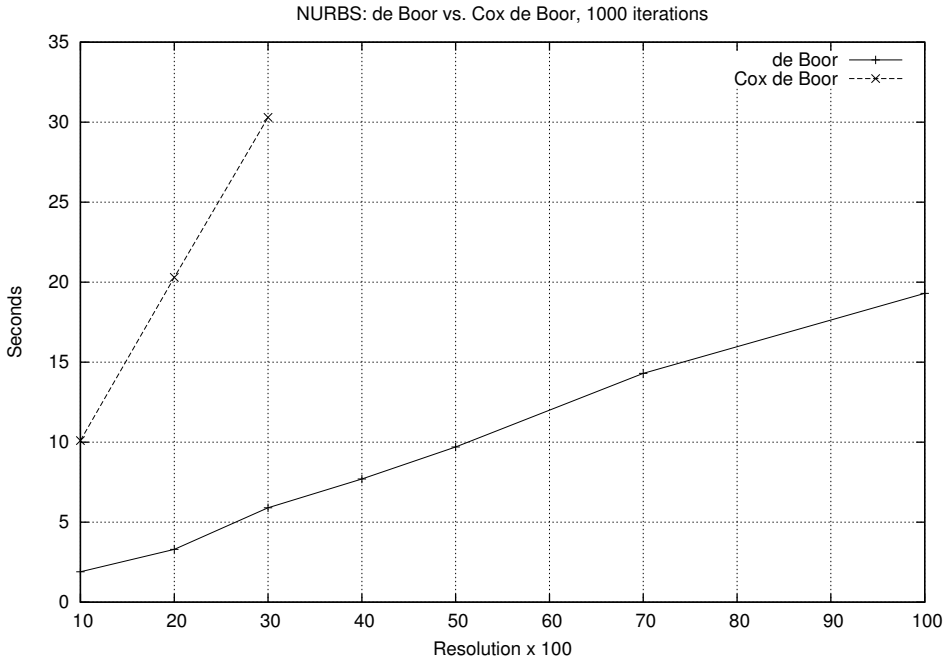


Figure 5.4: de Boor vs. Cox de Boor

As mentioned in chapter 3, the Cox de Boor algorithm is not optimized "by default". We have to extend it so that it does not include unnecessary control points, and also it helps to force successive knot values to zero or one. That enables us to store a small number of matrices that are computed off-line for fast on-line evaluation. Figure 5.4 shows two common methods for evaluating B-spline curves. It was included to demonstrate the importance of clever algorithms. The actual modifications to the Cox de Boor algorithm are not very hard to implement.

The Cox de Boor graph was only tested up to a resolution of 3000, where it simply took to long to evaluate any further.

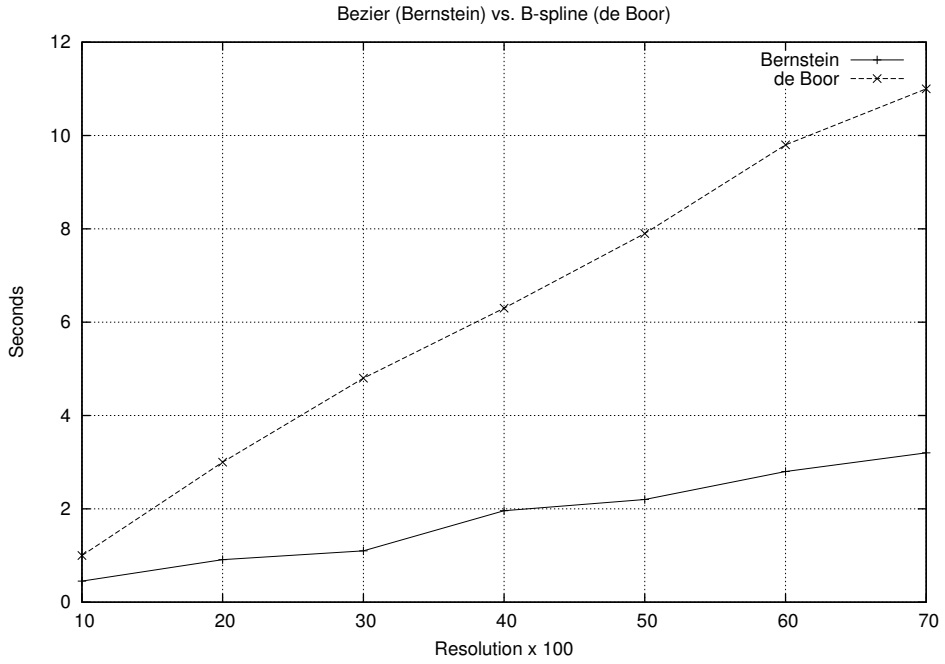


Figure 5.5: B-spline vs. Bezier

Lastly, in figure 5.5 we see why we often trade flexibility for performace in time-critical applications. It shows a comparison between a Bezier and a B-spline curve, both of degree three and four control points.

In most applications where high flexibility is not required, we use the Bezier curve due to its rapid evaluation and sculpting simplicity.

Chapter 6

Closing remarks

This thesis was intended to be a solid material for beginners in the field of parametric curves, offering a simple and intuitive sculpting tool and a walk-through of the theory behind them.

Only a fraction of the applications have been covered here, mostly aspects applicable to computer games and multimedia. The CAD section would be a thesis of its own, with volumetric sweeps, constructive solid geometry and so forth. Hopefully the reader has gained some understanding and interest to explore this popular field in computer graphics.

6.1 Personal comments

During earlier experiences with parametric curves, terms like NURBS and rational curves were intimidating, not least on the mathematical side. Many students feel the same way, so my effort has been to explain the common concepts in the easiest way possible. Hopefully this is something that computer graphics classes can take advantage of and hand this material to their students.

Since mathematics has never been my strong side, it was a good exercise to implement the software and hopefully provide a good explanation for other students. Some of the implementation may not be entirely correct, but at least the sculpting software works like intended and will serve as excellent hands-on experience with the curves and surfaces.

- Fun
- Fast
- File I/O
- Intuitive
- Portable code

These were the goals of the software, and I feel quite satisfied with the re-

sults. Of course, software like this can be expanded on forever and upgraded with new functionality.

6.2 Future work

- More types of curves and surfaces
- Surface subdivision
- Stabilize algorithms
- Output graph of basis functions
- Change GUI platform
- Rotational sweeps
- Advanced rendering options
- ...

Since surface subdivision is a huge topic and a good alternative for creating polygon meshes and solid objects, this is also a high priority if the software is continued.

The choice of GUI can also be discussed, since the GLUI package is rather basic. For instance, it would be convenient to have sliders in the parametric direction(s) for the knot insertion. This is a very common way to modify the knot vector, and would be easier to implement if another GUI package was used (GTK or Qt, for instance).

Regular shading or "flat shading" implies that we specify one normal per polygon. This means that the polygon will be rendered with equal intensity across its span, hence it requires a large amount of polygons to get a smooth appearance.

There are two major techniques to create interpolated shading, Phong- and Gourad shading. In the case of Phong, the polygon is illuminated pixel-per-pixel by interpolating the vertex normals. In its basic form it is very slow and not suitable for real-time. There are some clever versions which approximate the Phong model, and these are also considered to be of high priority since they enhance the visual experience tremendously.

In Gourad shading, we instead interpolate the color of each vertex across the polygon. This also yields a very pleasing result, but as with Phong shading it requires some clever tricks while rendering the parametric surface. We need to have some kind of list at each vertex that holds pointers to the surrounding polygons or vertices, and this is not very trivial if it is to be done in real-time.

6.3 Using the software

The thesis software has been developed on Linux Mandrake 9.1 with OpenGL 1.4, Glut 3.7 and GLUI 2.1. It should compile on any platform which sup-

ports the three latter (Windows, GNU/Linux, Solaris, among others).

Using the software is painless and self-explanatory. To the right, a strip of options are available which enables the user to switch between different types of curves and surfaces, load and save files and so forth.

Knot insertion is done by scrolling to the appropriate parametric value and pressing "Insert". The results of the insertion can be viewed by the "Show knots" option which renders the knot vector values on the canvas.

The field "Area" is used to specify a maximum area of the rendered triangles. If any are too large, they are subdivided to several smaller triangles. This option does not have to be used. We can also specify the uniform evaluation resolution by the "Resolution" scroller. Note that most of these values (resolution, rotation speed) are integers, but are internally handled as floats (1/integer).

The "Load" and "Save" buttons are used to store curves and surfaces on file. Just type a filename and hit the button. The file formats are easy to understand and can be manipulated with any text editor.

The "Weight" field applies to any rational curve or surface and changes the weight of the currently selected control point.

Rotation of the surfaces are done by pressing the X, Y or Z keys. This will start a rotation with a velocity given in the field "Rot.Speed", and halts when another rotation is specified.

References

1. Bernstein, S. "Dmonstration du thorme de Weierstrass fonde sur le calcul des probabilities." Comm. Soc. Math. Kharkov 13, 1-2, 1912.

2. Lorents, G. "Bernstein polynomials". Toronto Press, 1953.

3. Foley, J D. van Dam, A. Feiner S K. Hughes J F. "Computer Graphics principles and practice". Addison/Wesley, 1997.

4. Watt, A. Policarpo, F. "3D games Real-time rendering and Software technology", Addison/Wesley, 2001.

5. Farin, G E. "NURBS from Projective Geometry to Practical Use". A K Peters, 1999.

6. Lyche, T., K. Moerken and K. Stroem,Conversion between B-spline bases, in "Knot Insertion and Deletion Algorithms for B-spline Curves and Surfaces", R. N. Goldman and T. Lyche (eds.), SIAM, Phil, 1993, 135–153.

7. Lyche, T., "Knot removal for spline curves and surfaces", in "Approximation Theory VII", E. W. Cheney, C. K. Chui, and L. L. Schumaker, (eds.), Academic Press, Boston, 1993, 207–227.

8. de Casteljau, "F. Outillage Methodes Calcul", Anre Citroen Automobiles SA, Paris 1959.

9. Bezier, P. "Emploi des Machines a Commande Numerique", Masson et Cie, Paris 1970.

10. Barsky, B. "Computer Graphics and Geometric Modeling Using Beta-splines", Springer-verlag, New York 1988.

11. Bartels, R., J. Beatty, and B.Barsky, "An introduction to Splines for Use in Computer Graphics and Geometric Modeling, Morgan Kaufmann, Los Altos, CA, 1987.

12. Catmull, E., and R.Rom. "A Class of Local Interpolating Splines" in Barnhill. R., and R. Riesenfeld, eds., *Computer Aided Geometric Design*, Academic Press, San Francisco, 1974, 317-326.

13. Brewer, H., and D. Andersson, "Visual Interaction with Overhauser Curves and Surfaces", SIGGRAPH 77, 132-137.

Non-referenced but useful resources

Ching-Kuang Shene, "Ching-Kuang Shene's Homepage", [Online], Accessed 23 May 2003. Available from World Wide Web: http://www.cs.mtu.edu/~shene

Information Literacy Group, "Numeric style referencing", [Online], Accessed 23 May 2003. Available from World Wide Web: http://www.leeds.ac.uk/library/training/referencing/numeric.htm

Paul Rademacher, "GLUI User Interface Library", [Online], Accessed 23 May 2003. Avalible from World Wide Web: http://www.cs.unc.edu/~rademach/glui/

Mark Kilgaard. Nate Robbins, "Glut, OpenGL Utility Toolkit", [Online], Accessed 23 May 2003. Avalible from World Wide Web: http://www.xmission.com/~nate/glut.html