NURBS CURVES AND SURFACES TUTORIAL

Introduction to curves and surfaces

Every curve or surface can be defined by a set of parametric functions. For instance, *(x,y,z)* coordinates of the points of the curve can be given by:

$$x = X(t),\ y = Y(t),\ z = Z(t)$$

t being the parameter and X, Y, Z being polynomial functions in t.

If X, Y and Z are 1st degree polynomials, a line segment will be defined. In that case, two knowns only (i.e. two points or a point and a slope) will be sufficient to define this curve. If X, Y, Z are 2nd degree polynomials, a parabola segment will be defined and 3 knowns will be necessary to describe it (i.e. 3 points or two points and a tangent). For higher degree polynomials, describing the curve will involve more knowns. This number of knows is what we call the order of the curve, and is always the degree of the polynomial plus 1.

Most of the time, cubic polynomials[1] are used to represent curves. Indeed, more knows are needed for higher degree polynomials, what makes modelling difficult to handle. On the other hand, lower degree polynomials describe too restrictive curves, being either lines or parabolas, which are always planar curves.

Various approaches have been imagined by mathematicians, for instance, Bézier curves, Hermite curves, Catmull-Rom splines and B-Splines. More information on those curves can be found in [CGPP].

Introduction to B-Splines[2]

Introduction

"*The term spline goes back to the long flexible strips of metal used by draftspersons to lay out the surfaces of airplanes, cars, and ships. "Ducks", weights attached to the splines, were used to pull the spline in various directions."*[3]

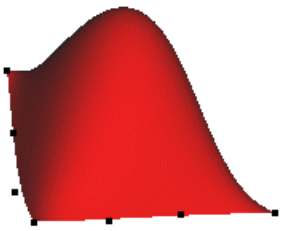
Based on this, a mathematical model was built and allows to define a curve blending control points by polynomials, introducing the concept of natural spline. The main issue with natural splines is that modifying a control point would affect the whole curve, as each polynomial coefficients depend on every control point.

---

[1] 3rd degree polynomials. 2nd degree polynomials are called quadric polynomials.

[2] This piece of text remains a basic introduction; this study, limited to real-time problems, will not deal with modelling. Thus, issues such as continuity will not be seen. A more complete description of B-Splines, including continuity issues and mathematical approach, can be found in [CGPP].

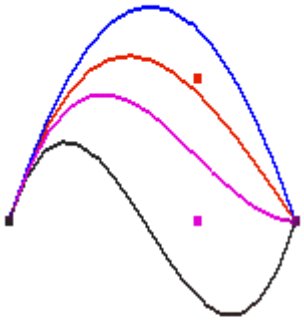
[3] [CGPP]



Surface based on B-Splines

The points are the control points (11 of 16 are visible here)

The curves we will use are based on B-Splines, consisting in several natural spline segments, each of which is defined by a reduced set of control points. Thus, polynomial coefficients will only depend on the control points of the curve segment considered. This is called local control because modifying a control point will only affect a few curve segments. Figures below[4] show the way control points affect the curve shape.



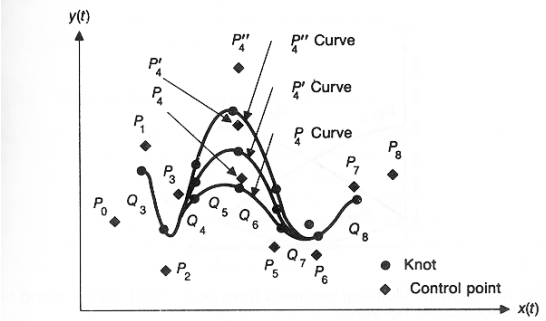


Fig. 11.23 A B-spline with control point P_4 in several different locations.

Note: we will not restrict to a certain degree for following chapters. The degree n of the curve can be any positive integer.

---

[4] Right figure is taken from [CGPP]

Various kinds of B-Splines

B-Splines blend a set of *p+1* control points: $\{P_0, P_1, ..., P_p\}, p \geq n$ and consist in *p-(n-1)* curve segments: $\{Q_n, Q_{n+1}, ..., Q_p\}$. In addition to that, we can define a common parameter *t* rather than considering *t* in the interval *[0, 1[* for each curve segment. Thus, for each curve segment Q_i, *t* will belong to the interval $[t_i, t_{i+1}[, n \leq i \leq p$. Moreover, each segment Q_i will only be affected by n control points: P_{i-n} to P_i.

For each $i \geq n$, there is a knot between Q_i and Q_{i+1} for the value t_i of the parameter *t*. There is a total of *p-n-2 knots* for the B-Spline. Here comes the concept of uniformity: if the knots are uniformly distributed on the interval *[0, 1[* (i.e. $\forall i \in [n, p[, t_{i+1} - t_i = t_{i+2} - t_{i+1}$), the B-Spline will be said to be *uniform*. In the other case, we will use the term *non-uniform*. It is worth mentioning the fact that those definitions imply that the knots are increasing (i.e. $\forall i \in [n, p], t_i \leq t_{i+1}$).

Now, assuming that the coordinates *(x, y, z)* of a point of the curve are given by ratios of polynomials (eq. 0), the B-Spline will be said *rational, e*lse, it will be said *non-rational.*

$$x = \frac{X(t)}{W(t)},\ y = \frac{Y(t)}{W(t)},\ z = \frac{Z(t)}{W(t)}$$

(eq. 0)

To sum up what we have just seen, we have 4 different types of B-Spline:

- Uniform Non-rational
- Non-uniform Non-rational
- Uniform Rational
- Non-uniform Rational[5]

Only the last type will be studied in this article, being the most general case[6].

---

[5] Often called NURBS (**N**on-**U**niform **R**ational **B**-**S**pline).

[6] Every other type of B-Spline is a particular case of NURBS : W(t) = 1 for non-rational and uniformity is a particular case of non-uniformity.

NURBS curves and surfaces

Mathematical approach

A NURBS curve Q can be defined by parametric equation *eq. 1a*. Similarly, *eq. 1b*, generalises equation *eq. 1a* to a 2 dimensions parametric space to define a NURBS surface.

$$Q(t) = \frac{\sum_{i=0}^{p} B_{i,n}(t).P_i.w_i}{\sum_{i=0}^{p} B_{i,n}(t).w_i}$$

(eq. 1a)

$$S(u,v) = \frac{\sum_{i=0}^{p}\sum_{j=0}^{q} B_{i,m}(u).B_{j,n}(v).P_{i,j}.w_{i,j}}{\sum_{i=0}^{p}\sum_{j=0}^{q} B_{i,m}(u).B_{j,n}(v).w_{i,j}}$$

(eq. 1b)

where P_i is a control point, w_i the weight being associated to it, and $B_{i,n}$ a basis function defined recursively by:

$$B_{i,0}(t) = \begin{cases} 1 \; if \;\; t_i \le t < t_{i+1} \\ 0 \; else \end{cases}$$

$$\forall k > 0, B_{i,k}(t) = \frac{t - t_i}{t_{i+k} - t_i} B_{i,k-1}(t) + \frac{t_{i+k+1} - t}{t_{i+k+1} - t_{i+1}} B_{i+1,k-1}(t)$$

(eq. 2)

Let's get closer and try to see what the different blocs of equations *eq. 1a* and *eq. 1b* represent. Having a denominator comes from the rational characteristic of the curve (or surface). If the curve had been non-rational, *eq. 1a* would only consist in: $Q(t) = \sum_{i=0}^{p} B_{i,n}(t).P_i$. Furthermore, we can consider basis functions $B_{i,n}$ as blending functions, allowing to modify the influence of a control point on a given point of the curve, giving more importance to control points closer of current t value.

Derivative computation

Finally, equation *eq. 1a* provides us with an easy method to plot a curve. However, how can we obtain the value of the derivative of *Q(t)* ? Indeed, knowing how to derive equation *eq. 1a* is similar to knowing how to derive $B_{i,n}$, leading to a dead-end, considering the given form of $B_{i,n}$. Same remark can be issued concerning equation *eq. 1b* as calculating partial derivatives is even more tricky. Then, why not remembering that $B_{i,n}$ can also be expressed through a polynomial. Obtaining the derivative of $B_{i,n}$ will become obvious then.

$$B_{i,n}(t) = \sum_{k=0}^{n} C_{i,n,k}(t).t^k$$

(eq. 3)

Combining equations *eq. 2* and *eq. 3*, we can obtain the expression of the coefficients of the polynomials:

$$C_{i,0,0}(t) = B_{i,0}(t)$$

$$C_{i,n,0}(t) = \frac{t_{i+n+1}}{t_{i+n+1} - t_{i+1}} C_{i+1,n-1,0}(t) - \frac{t_i}{t_{i+n} - t_i} C_{i,n-1,0}(t)$$

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

$$\forall k \in \{1..n-1\}, C_{i,n,k}(t) = \frac{C_{i,n-1,k-1}(t) - t_i.C_{i,n-1,k}(t)}{t_{i+n} - t_i} - \frac{C_{i+1,n-1,k-1}(t) - t_{i+n+1}.C_{i+1,n-1,k}(t)}{t_{i+n+1} - t_{i+1}}$$

(eq. 4)

Notice that those coefficients depend only on the knot span in which t is and not directly on the value of t. Thus, those coefficients have a constant value on the various knot spans. As a consequence, $B_{i,n}$ derivative is now known:

$$\frac{dB_{i,n}(t)}{dt} = \sum_{k=1}^{n} k.C_{i,n,k}(t).t^{k-1}$$

(eq. 5)

The same method can be applied to a 2 dimension parametric space, providing us with the formulas for surfaces:

$$\frac{dB_{i,n}(u)}{du} = \sum_{k=1}^{n} k.C_{i,n,k}(u).u^{k-1}$$

(eq. 6a)

$$\frac{dB_{i,n}(v)}{dv} = \sum_{k=1}^{n} k.C_{i,n,k}(v).v^{k-1}$$

(eq. 6b)

At this point, we are in possession of all the stuff needed to determine the coordinates of a point and the normal to the surface on this point. We can now think of computing everything in real-time, in an effective way. We will concentrate on surfaces as curves can easily be deduced from what will be explained.

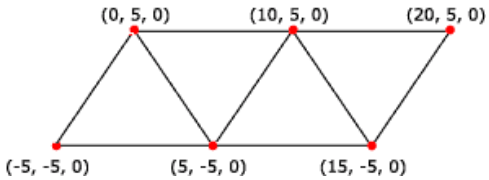
Real-time rendering of NURBS Surfaces

General remarks

Rendering of a surface is a two steps process : first, computing the points that will form the mesh of the surface and then, send this mesh to 3D API.

First step, called tessellation, consists in fact in transforming the mathematical continuous definition of the surface into a discrete definition. For some complexity and efficiency reasons, we will use uniform tessellation, that is to say that we will compute points for uniformly spaced values for parameters u and v.

Contrary to the first one, this step is dependent on the 3D API chosen. Most efficient method to represent this mesh seems to be triangle strips. Indeed, this interpretation of a set of points is the one allowing a minimal use of memory , contrary to specification of independent facets.



The mesh represented above[7] can be defined in two ways:

- By a set of triangles (4 triangles each defined by 3 vertices, thus 12 vertices sent to the API).
- By a triangle strip (on strip defined by 6 vertices, thus 6 vertices sent to the API).

In both cases, result will be the same but we clearly see that in the first case, common points will be stored for nothing. We will make use of OpenGL triangle strips in our study but Direct 3D also allows to declare such primitives. Let's focus on tessellation now, as rendering is not the critical step.

[7] Figure taken from Direct X documentation, (c) Microsoft.

Tessellation of the surface

First, let's see how to pre-calculate basis function values, as calculating them during tessellation is far too costly.

We have seen that the coefficients of the polynomial representation of $B_{i,n}$ are constant between two knots. Thus, except if we change the knot vector, we will not need to compute those coefficients in each tessellation. This is convenient for us because those coefficients require a recursive evaluation which is very costly.

Now regarding values of the basis functions $B_{i,n}$. We know that, for given values (i, j), only a few of those functions will be non-zero: to go further, this number is precisely the order of the surface. For each point to be computed, we then can store values of $B_{i,m}(u)$ and $B_{j,n}(v)$. Finally, we can pre-weight the coordinates (x, y, z) of the control points. As a result, only the calculus of the sum in *eq. 1b* remains.

To compute normal vectors, we have to compute tangents to the surface for each point, in u and v directions. Normal vector will be the dot product of those two tangent vectors. This can be done thanks to equations *eq. 1a*, *eq. 6a* et *eq. 6b*. We also pre-calculate partial derivatives of $B_{i,n}$ and we will only have a sum to do for tesselation.

Finally, we will be able to use dynamic tessellation, that is to say adapt tessellation value to the distance of the surface and its volume : if a surface is very far, we only need a few vertices whereas if the surface is close to the viewer, we will need extra vertices to display a smooth surface. This is a vital process when one has to display a large amount of surfaces, so as to send only useful information to the graphics card.

Conclusion

Rendering a surface will then consist in 4 main steps:

- Determining tessellation value (dynamic tessellation).
- If this value differs from previous one, pre-compute basis function values and its derivatives once again.
- Tessellate surface.
- Send vertices to the graphic card using the 3D API and triangle strips.

We did not talk about texture mapping: here, we used uniform planar mapping. This method is simple and efficient. However, it can show limitations for some surfaces where deformations can be observed. Anyway, this is sufficient for our application.

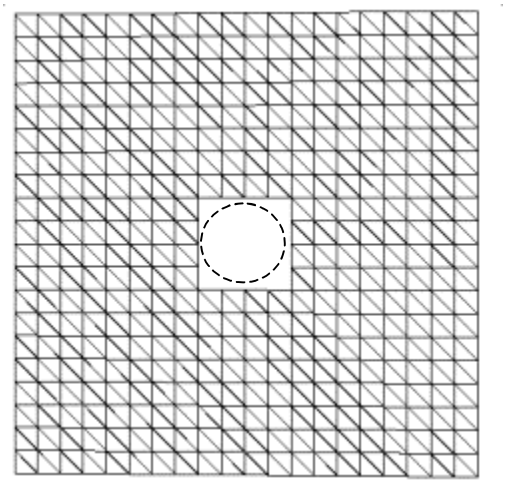
We also considered that we would not have to tessellate the surface each frame, which is true for worlds where the user moves rarely or slowly. We then compute each point and store it in a vertex buffer. If surface had to be tessellated each frame, for instance in the case of a race game or plane simulator, it would be more efficient to send triangle strips as we calculate the points, gathering steps 3 and 4.

Surface trimming

We can think of trimming the surface in two ways: building the mesh taking the holes into account or building the untrimmed mesh and then trimming it. First idea is often based on the calculus of intersection of the surface with the hole. Then, a set of points is computed and mesh is finally built. However, computing the intersection between the surface and the hole is not a simple thing, and doing it in real-time is still the object of thesis that I could not find. We then have the second idea which appears to be quite straight forward to implement and very efficient, even if it has some limitations in a few cases.

Trimming the surface

The idea is the following: during tessellation, we will remember the points that are on the surface and those that are not. To do so, we simply reuse previous tessellation algorithm, adding a test for each point checking whether the parameters are in a hole or not. In that case, we remember the hole, else, we indicate that this point is on the surface, by memorising -1 for instance. We also have to modify the function sending those points to the graphics card. The idea is still simple, I will not go into further details, as the source code is self-explanatory. At this stage of trimming, we have the following surface with, in dot line the hole has it should appear.



The result is not really sufficient but we have a trimmed surface. Next step will be to refine this surface to obtain a round hole.

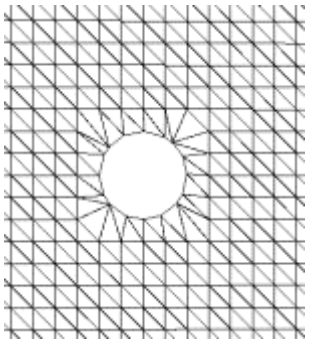
Note: you will have noticed that the hole are defined in the parametric plane and not in object space. This allows us to keep hole description independent from every parameter of the surface (control points, order).

Contour refinement

Idea behind refinement is simple: we have to move the points that are on the border of a contour and modify their coordinates in order to approximate the shape of the hole.

First question is: how to identify "frontier points" and which one shall we modify, surface points or hole points we eliminated ? First, we can notice that a point is on the border of a hole if at least one of its 8 neighbours does not have the same state[8] than him. This implies to know the state of its 8 neighbours and thus, that trimming and building the surface is not possible at the same time (as we know the state of 5 of the neighbours in a primary version of the program). Second part of the answer is nearly obvious: refine the frontier points that are inside the hole would lead to a loss of accuracy in the approximation of the shape of the hole. Indeed, those points are less numerous than frontier points on the surface.

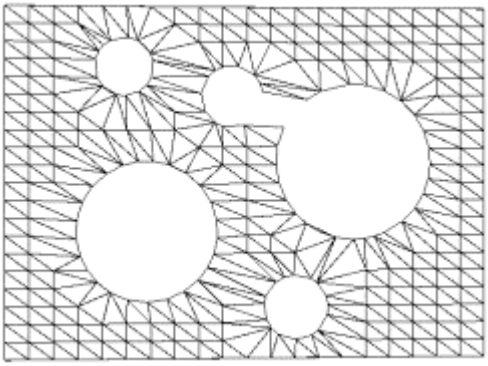
Now that we know which points to move, how could we determine their coordinates to approximate the shape of the hole ? We will simply search for the contour point that is the closest of the point to move, keeping in mind that this is done in the parametric plane. We will need equation *eq. 1b* which allows to compute the coordinates of a point whatever are the parameter values. The point we will obtain will be on the border of the hole and the resulting trimmed surface will have the correct shape.



Side-effects and solutions that were given up

The first side-effect occurs when holes are too close. This comes from the fact that between those two holes, we do not have any point of the surface to enable us to refine the contour, tessellation value being too big.

[8] Remember, we stored the state of a vertex before (the hole he was in or else, -1).



Solutions to this problem could be:

- Use the method consisting in refining frontier points that are inside the holes (the method we gave up in previous paragraph).
- Increase tessellation value in order to have points between the holes.
- Change our algorithm: you might have had the idea to compute each point of the hole and then connect it to the surface. This method was given up because of the difficulty to connect contour points and surface points in real-time. However, this method would be the more accurate and would correct this side-effect.

Real-time rendering of trimmed NURBS Surfaces

Critical steps

When I obtained the first draft of my program performing rendering of trimmed surfaces, two main bottlenecks were identified[9]:

- The test indicating whether a point belongs to a contour.
- Computation of the refined point coordinates.

Indeed, the test was based on the most intuitive method: a point is inside a polygon if the sum of the angles with the points of the polygon equals 2 PI radians. A more efficient way to do this test has been found in the FAQs of the forum comp.graphics.algorithms. However, the second bottleneck was far more costly. This part of the algorithm was inefficient because it made use of the recursive formula of the equation *eq. 2*. Given the fact that contours have a small number of points, pre-computation of surface points appeared naturally and was quite simple to implement.

[9] I take the opportunity to advice you to use profilers to identify bottlenecks in you programs. This is really useful.

Getting rid of the two passes

A problem is still remaining: how to refine the surface at the same time that we tessellate it ? Indeed, at present we are computing it in two passes (computing the surface and trimming it). This is quite inefficient, and we can just notice that from the 3rd column and the 3rd row, we know the 8 neighbouring vertices of the vertex located one row and one column before current one. We then can refine most of the vertices of the surface during the computation. We refine remaining vertices (first and last columns and rows) in a shorter and faster second pass.

Conclusion

Here we are, this is finally the end of this (long) tutorial about trimmed NURBS surfaces. I am sure that plenty of optimisations are still possible. A sample program is provided with this tutorial, documented with Doxygen (very useful tool that I recommend). I would be glad to receive some feedback and / or suggestions from readers. Contact information follows the bibliography.

Bibliography

[CGPP] Foley, VanDam, Feiner et Hughes, "*Computer Graphics: Principles & Practice - 2nd edition*", Addison-Wesley, 1996

[VRML] Grahn, Volk et Wolters, "*NURBS in VRML*", Blaxxun Interactive

[UNRT] Dean Macri, "*Using NURBS Surfaces In Real-time Applications*", www.gamasutra.com, Nov. 1999

Files included, greets, contacting me

Folder /prog	: sample program (English version only). Visual C++ project and makefile. It should be a portable program, at least I did my best for that.
Folder /prog/bin	: executables, surface files and dll (precompiled)
Folder /prog/doc	: documentation of source code (.chm file)

Lot of ideas where taken from [UNTR] concerning untrimmed surfaces in real-time. To me, I did a good job of clarification (and translation for the French version). Credits are given now and I hope that I did not make anything wrong.

I want to thank Allergy for his patience, and the linux version of the sample program. Visit his web site www.alrj.org for more tutorials (French language only).

You can e-mail me at vprat@ifrance.com and give a look to http://vprat.ifrance.com