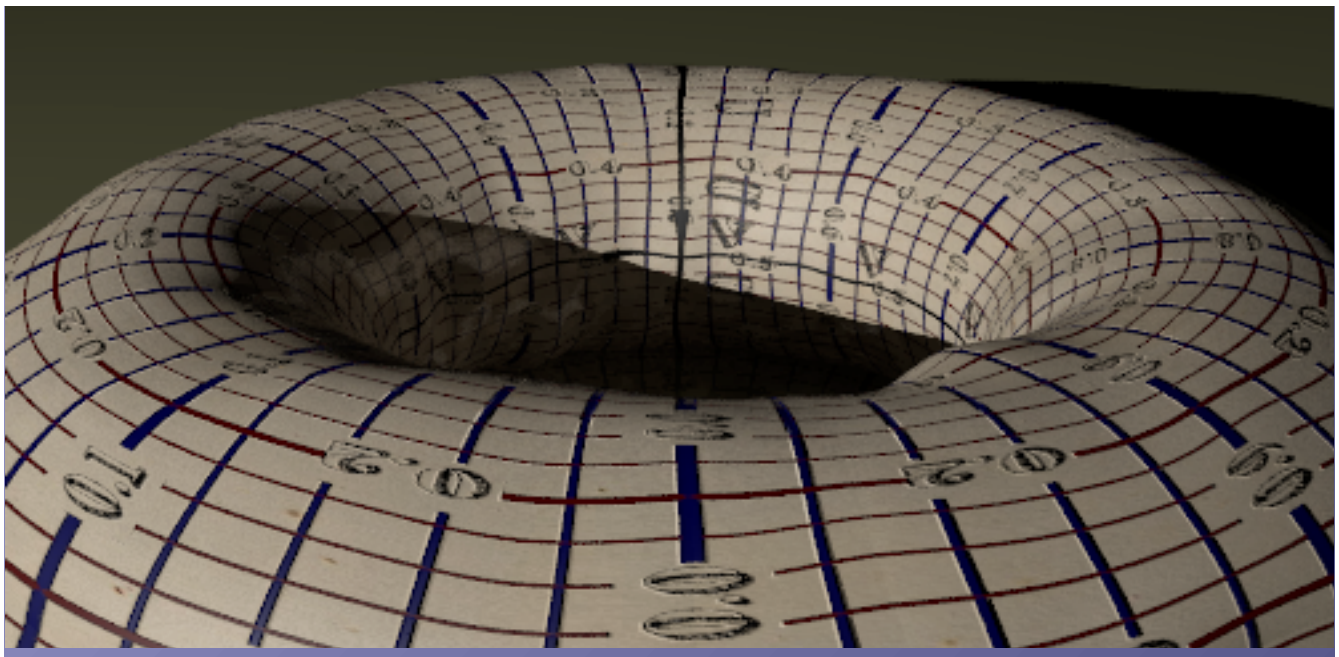
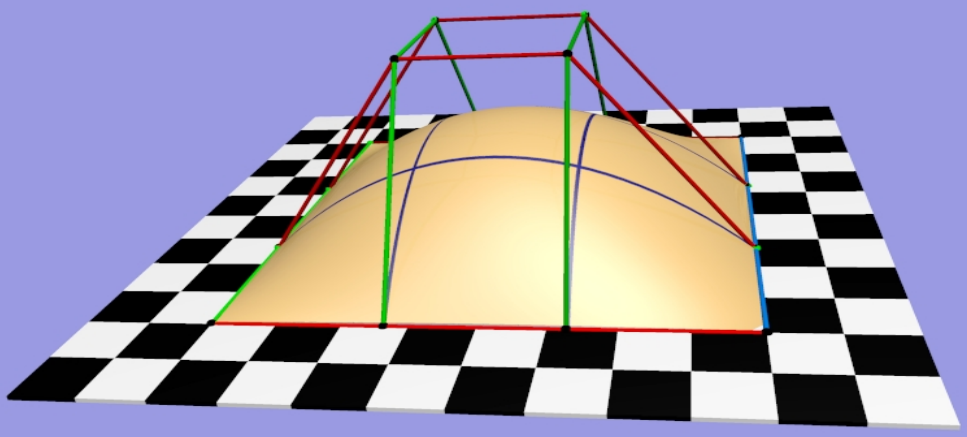


Bézier bicubic patches

$$\mathrm{p}(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i^n(u)\, B_j^m(v)\, k_{i,j}$$

& Bernstein polynomials



Bézier Bicubic Patches and Bernstein Polynomials

William H. Walker
POV-Ray user “Bald Eagle”
New Hampshire, united States of America

10. August 2020

ABSTRACT

The fundamentals of Bézier splines and surfaces is presented, and the basics of constructing Bézier bicubic patches, adjoining them, and texturing them with uv-mapped patterns in POV-Ray is covered. Advanced methods for generating smooth curves are discussed. Sophisticated CAD-style tools for analyzing and visualizing surface curvature were developed by the author for use in POV-Ray.

MOTIVATION

In the absence of a suitable GUI modeler with the capability to export to POV-Ray’s Scene Description Language (SDL), it can be difficult for many users to implement bicubic patches in a manner that is easy and efficient enough to make using them worthwhile; this paper seeks to remedy that. It also aims to illustrate some of the practical and artistic aspects of Bézier curves and surfaces, such that users might use the examples provided as a starting point for their work, and as a useful reference during development of their scenes. It is hoped that by covering some of the basic concepts about how the curves and surfaces are generated and manipulated, that the reader may feel inspired to experiment with these objects in their scenes.

SCOPE

This work is primarily directed towards novices and hobbyists in computer graphics and is neither authoritative nor complete. It focuses on modeling surfaces in the Persistence of Vision ray-tracer (www.POV-Ray.org), and therefore code examples will employ its Scene Description Language syntax (SDL), but the material presented is applicable to generating splines and surfaces in many other software applications and programming languages. Most of the mathematics involved are the basics of what is typically presented in a high-school calculus course. The author claims no special knowledge or skill in this area, and everything in this paper was learned from scratch using nothing but information available online.

INTRODUCTION

Bézier curves are parametric curves. Rather than being implicit functions in x, y, and z, they are instead defined by a parameter *t* upon which the values

of x, y, and z are dependent. One can think of this by imagining any curve – for example a long, winding road going from Point A to Point B through a mountainous region. Any point on the road may be located by determining what percentage of the trip has been completed. At any given percentage P, one can determine where they are on the map: E-W, N-S, and elevation.

Bézier patches are surfaces defined by a 2D grid of Bézier curves and are parametric functions of parameters *u* and *v*. One might think of this like the numbers for outdoor seating, where in one direction are the aisles, and in the other the individual seats are arranged in rows. Although they may be set up on rolling hills or in an amphitheater, all of the seat coordinates are still aisle u, seat v.

Linear Curve

A linear Bézier curve is equivalent to a simple linear interpolation between two points, based on the parameter *t* over the range [0, 1]. The result can be considered as a blend, or mixing of the two points, their relative proportions being dependent on a sliding scale. Another way of saying this, is that it is a curve defined by the function F(*t*) = <x, y, z>.

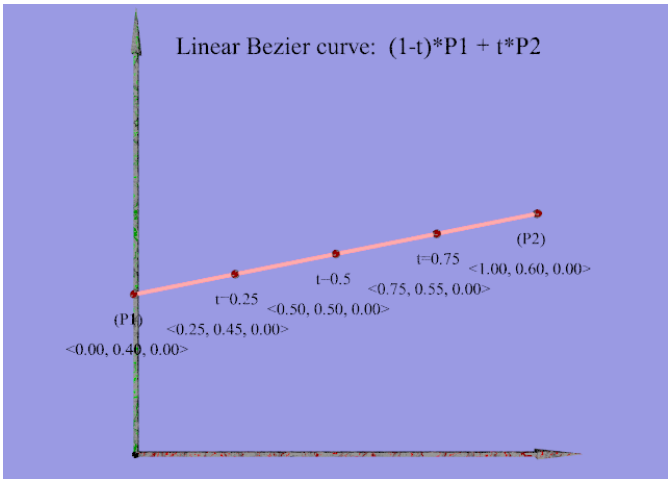


Illustration 1: Linear Bézier curve

At *t* = 0, the function evaluates to 100% P1 and 0% P2. At *t* = 1, the function evaluates to 0% P1 and 100% P2. We can easily write a macro in POV-Ray to perform this calculation, which represents a traversal along a vector.

```
#macro Lerp (_P1, _P2, T)
   #local _Pmid =
   (1-T)*_P1 + (T*_P2);
   _Pmid
#end
```

Quadratic curve

A quadratic Bézier curve is defined by three points, and is the linear interpolation of the results of two other linear interpolations.

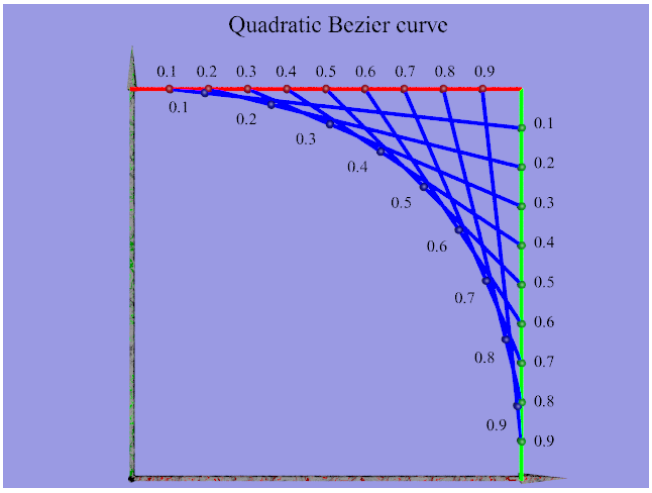


Illustration 2: Quadratic Bézier curve

As *t* progresses from 0 to 1, the red points are interpolated between P1 and P2, and the green points are interpolated between P2 and P3 to form the endpoints of the series of blue lines. A final interpolation is performed along the blue lines to calculate the blue points which actually form the quadratic Bézier curve.

```
#for (T, 0, 1, 0.1)
 RedPoint = Lerp (P1, P2, T);
 GreenPoint = Lerp (P2, P3, T);
 BluePoint =
 Lerp (RedPoint, GreenPoint, T);
#end
```

Higher-order Bézier curves are constructed in an analogous manner, simply with more control points and more interpolation steps. It is important to note at this point that Bézier curves and surfaces are approximations, not interpolations. Excluding the linear case, the curve generated by the Bézier curve only intersects the endpoints, not the “internal” control points.

Cubic Bézier curve

A cubic Bézier curve is defined by four points, and requires 6 linear interpolations. The red, green, and blue points are linear interpolations between the four control points that construct the two sets of gray lines. Two sets of gray points are linear interpolations along the gray lines that form the endpoints of the blue lines. A final linear interpolation along the blue lines gives the black points which lie along the cubic Bézier curve.

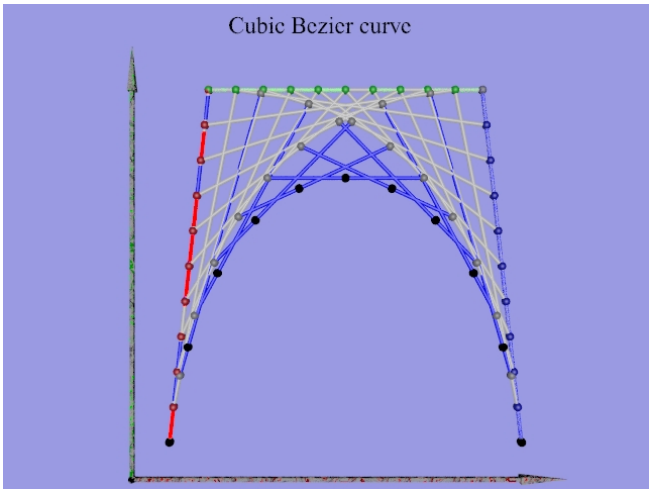


Illustration 3: Cubic Bézier curve

This method of layering linear interpolations on top of previous interpolations, was invented by Paul de Faget de Casteljau in 1963 (based on a summation of equations developed by Sergei Natanovich Bernstein in 1912[1]) at

the Citroën automobile company[2] and was subsequently adopted as the primary computational method for efficiently and accurately describing the complex curves of auto bodies, using only a small number of data points.

Rather than a long recursive algorithm, a more direct method of determining the points on the curve is by mathematically evaluating a series of parametric polynomial equations. This was the achievement of Pierre Bézier at Renault, and independently reproduced de Casteljau's result though by a different method.[3] The same cubic curve generated by four levels of interpolation is obtained by summing the four terms of the equation:

$$A(1-T)^3+3BT(1-T)^2+3CT^2(1-T)+DT^3$$

This is the same equation that is used internally by POV-Ray to evaluate a cubic Bézier spline.

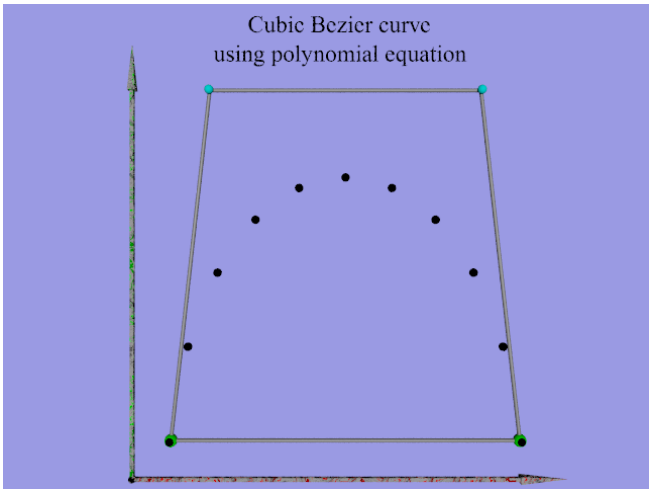


Illustration 4: Cubic Bézier point positions calculated using the Bernstein polynomial

Properties of Bézier curves

Bézier curves are often described in the literature as being affinely invariant. All this means is that you can rotate and translate it all you want, but it will otherwise still be the same shape. This is because unlike a mathematical curve like a parabola, whose shape is determined by the exact x value, a Bézier curve is not an implicit function of its specific spatial position, but a parametric function of the control points used to define it. Its control points can be moved as a set anywhere in space, but the curve is defined only by their relative positions. Moving the control points as a unit only changes the position or orientation of the surface, it does not change its shape. Returning to our simple linear curve, it is intuitively self-evident that a line between two points will not change if the end points are rotated or translated as a unit. Since the Bézier curve is, in a sense, the result of a series of progressive linear interpolations, then it too retains its shape, regardless of the overall transformation of it control points, because the curve is defined by the relative positions of the control points to each other, not their absolute position in space.

The curve begins and ends at the first and last control point, and resides inside of the polygon

formed by the control points, or the “convex hull”. Changing any control point affects the entire curve, to varying degrees, but there is no localized control of any given section of the curve.

Of special importance for modeling, the beginning and end portions of the curve are tangent with the sides of the polygon where they intersect with the control endpoints. The curve always intersects these two endpoints, but never intersects any intermediate control points. This is important because if two curves needed to be joined smoothly, like the top and bottom portions of an “S”, then all that needs to be done is have the last two control points of the first curve, and the first two control points of the second curve all lie on the same line. They will therefore have the same slope, and the curves defined by those control points will not only be continuous, but smoothly connected as well.

Bicubic patches

If we look at the arch formed by the cubic Bézier curve in Illustration 4, we can imagine 3 more copies of that curve distributed in the z-direction, with the same points on each curve being connected with a straight line to generate a mesh like an arched tunnel. This mesh would be cubic in one dimension (the splines), and linear (the connecting lines) in the other. To form a bicubic mesh, we could connect the arches in the z-direction not with straight lines, but with other cubic Bézier curves that are composed of four control points oriented in the z-direction instead, replacing all of the control points in the copy arches with these new points. By doing so, we create a series of Bézier curves in the x,y-plane that are simultaneously a series of Bézier curves in the z,y-plane. This results in a surface that is the tensor product of eight Bézier curves. Now, when these different Bézier curves are connected, we get not a straight tunnel, but a complex surface as the control points of the Bézier curve in one dimension are swept through the Bézier curves in the other.

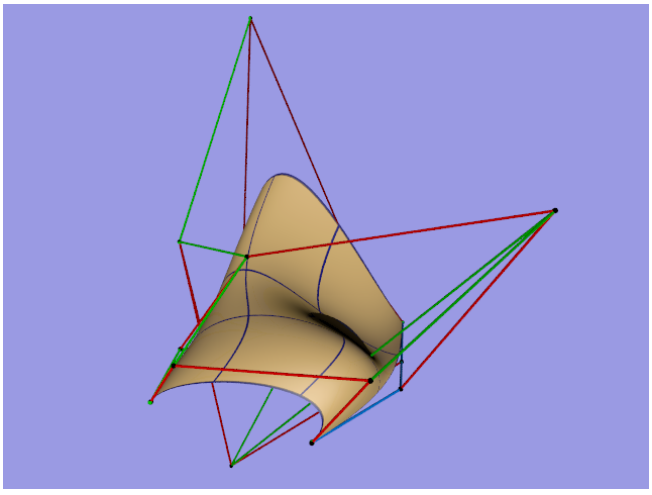


Illustration 5: A Bézier patch used to model a complex surface

There is nothing special about this, it is just the sum of a few simple functions that are the

coefficients of vector quantities (the control points). Indeed, a Bézier patch can be created using a parametric {} object in POV-Ray by using the *u* and *v* parameters in three separate equations for the *x*, *y*, and *z* components of the surface.

```
parametric {
   function {BezierX (u, v)}
   function {BezierY (u, v)}
   function {BezierZ (u, v)}
   <0, 0>, <1, 1>
contained_by {box {bMin, bMax}}
}
```

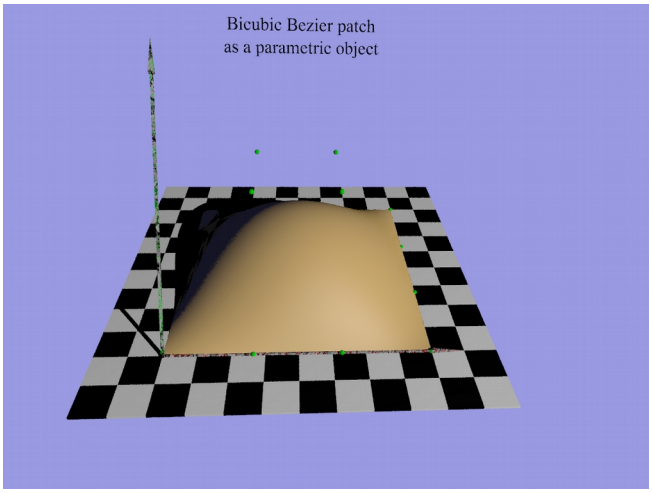


Illustration 6: A Bézier patch rendered using POV-Ray's parametric {} object

uv-Mapping

The uv-mapping of Bézier patches is easy to understand because the interpolation in one direction of the Bézier patch corresponds to the image or texture data in the *u* dimension, and the same interpolation in the orthogonal direction of the patch similarly corresponds to the same texture data in the *v* dimension.

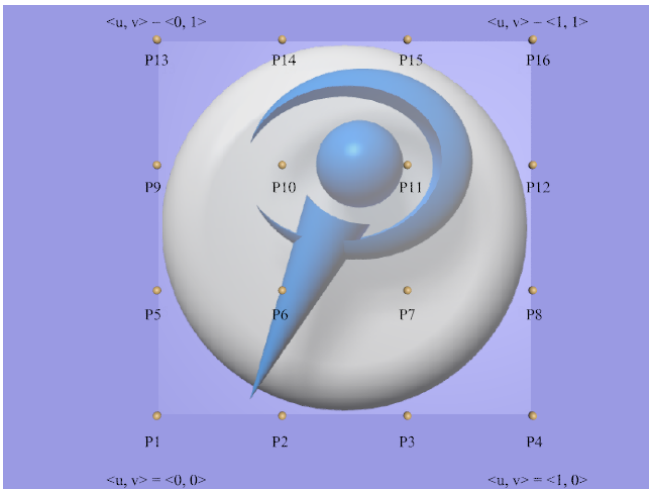


Illustration 7: uv-mapping of a simple Bézier surface

Selecting a specific portion of the texture to be mapped is also easily done, as the uv coordinates of the control points can be defined, and thus the patch can be fully covered with only a sub-rectangle of an image map or other planar texture.

In order to change the way a patch is mapped, one need only change the uv-vectors. Since the texture to be mapped runs, by definition, from 0 to 1 in both the u and v dimensions, specifying uv-vectors with intermediate values results in mapping only the portion of the texture that lies in the region defined by the uv-vectors. This may result in compression or expansion of the texture.

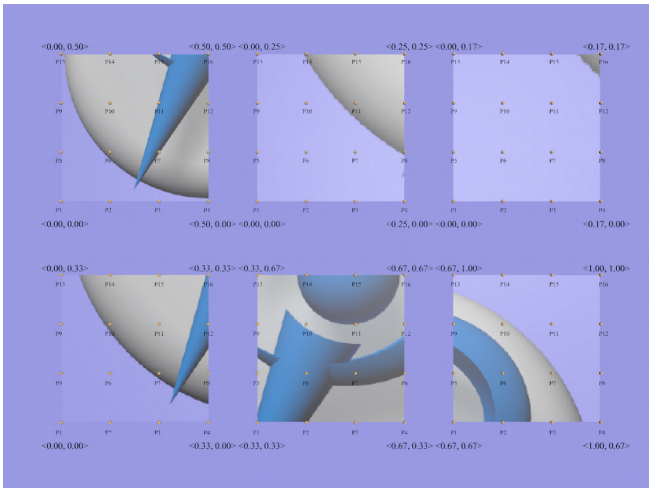


Illustration 8: Effect of changing the range of the uv-vector coordinates

Since the uv-vectors are independent of the geometric coordinates of the Bézier patch, any arrangement is possible, and the texture may be flipped, sheared, rotated, or twisted as a result of the choice of specific uv-coordinates and their ordering in the Bézier patch definition.

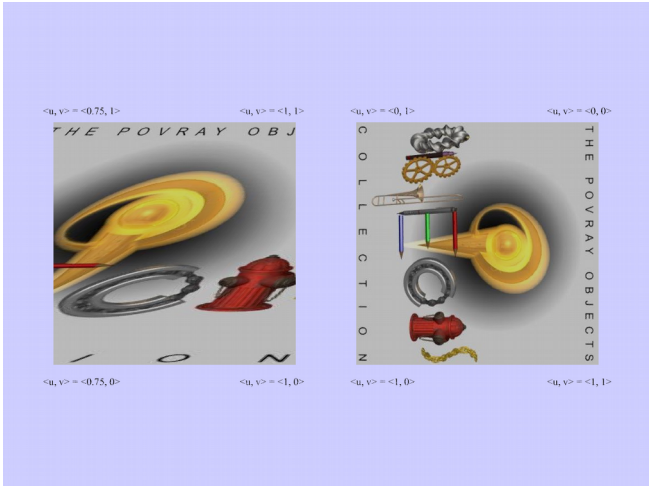


Illustration 9: A shearing effect and a rotation due to uv-vector values

It is important to note that it is the orientation of the texture being uv-mapped to the surface that is changed, not the spatial coordinates of the bicubic patch itself. So by switching the right and left or top and bottom pairs of vectors, the texture can be flipped or mirrored yet the original surface normal is retained. Assigning values outside of the 0-1 range results in either a smaller uv-mapped texture if the *once* keyword is specified in an image_map pigment, or else a repetetive tiling of the uv-mapped pattern.

Illustration 10: Tiling a pattern and twisting the right half 180 deg

Odd arrangements of uv-vectors can give odd and unexpected results which may be puzzling. A naïve expectation might be that the uv-vectors of the texture are simply translated to the corners of the patch and everything in

between is “stretched”. But a more thorough analysis and visualization of the process of how the resulting uv-map is generated explains how initially unexpected results are entirely logical, once one understands the exact process.

Consider the following situation where the uv-vectors form a concave hull. Linearly interpolating up the sides from the bottom shows how the *v* isolines form a deCasteljau-style pattern and include an area outside of the region enclosed by the uv-vectors. Even more suprising is that the isolines double-back over the red and green pencils, causing them to be sharpened on both sides in the final image!

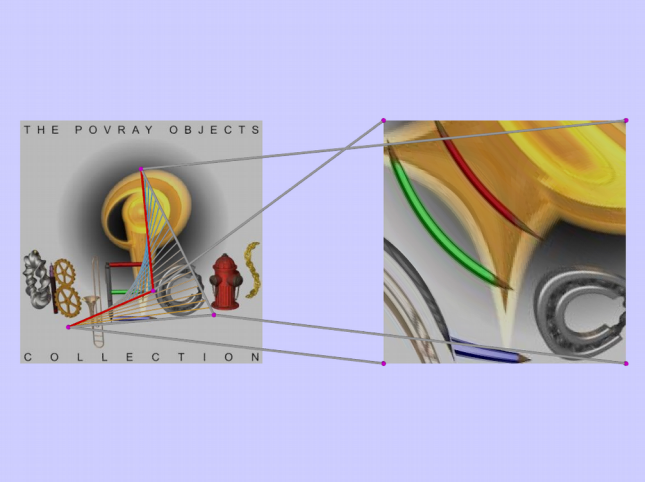


Illustration 11: Unexpected and surprising results can be obtained by odd uv-mappings

Smoothly joining Bézier patches

In order to join or “stitch” Bézier patches together smoothly to form a continuous surface, it is necessary to understand a few basic concepts. The smoothness of a curve can be expressed mathematically, and there are two sets of metrics used to describe the “fairing” of a curve. G^n is a geometric continuity concerned with how two curves behave at the point where they touch. C^n is a more stringent metric concerned with the differentiability of the two underlying functions of the curves where they are joined.

G^0 continuity means that the zeroth derivatives of two curves are the same at their intersection. This just means that the endpoints of two separate Bézier curves meet at the same point.

G^1 continuity means that the direction of the tangent vectors are identical.

G^2 continuity means that the curvature of the two curves where they meet is the same.

To assist in creating smoothly connected Bézier splines and patches, a number of useful techniques have been developed.

Tangent vector direction and size

Since Bézier curves intersect their endpoints, it is trivial to concatenate multiple Bézier splines to form a continuous curve. To make them flow smoothly without corners or cusps, it is useful to know that Bézier curves are tangent to their control polygon at either endpoint. Simply extending the vector defined by the last two

control points in the first segment by some amount gives us coordinates of a second control point necessary for both curves to be G^1 continuous.

Now, for the curves to actually be C^1 continuous, the tangent vectors need to be equal in both direction and magnitude. To do this is fairly simple, requiring only that we add the same vector defined by the last two control points in the first segment to the first control point of the second segment. This gives us the coordinates of the second control point necessary for both curves to be C^1 continuous. In mathematical terms, differentiating the Bernstein polynomials of each spline gives us the first derivative $F'(t)$, that we can use to calculate the instantaneous slope of both curves at the endpoints. These are the tangent lines where the curves meet, and not only will the geometric direction of the tangent lines be the same, the value of the parametric equation $F'(t)$ for each curve will be identical at the point where they meet.

Joining Bézier patches even more smoothly

Although two curves can meet with tangency, there is still an abrupt quality to the surface formed. This can result in a visible line or crease at the joint, or in noticeable sharp aberrations in the highlights and reflections of the surface.

In order to create an even smoother transition, one more additional degree of smoothness must be achieved. A typical example of G^1 tangency is the ¼-round edge of a rounded square or cube, formed by the union of circles and lines, or spheres, cylinders and planes. This corner-rounding is often used in wooden articles or simple geometric models, and the line where the rounding terminates is a pronounced transition. This can be described mathematically by comparing the curvature of the two curves that meet. A circle, cylinder, or sphere has a finite radius, and a straight line or flat surface can be thought of as a circle, cylinder, or sphere with an infinite radius. That's a big jump in curvature.

Both a ¼-circle and a line can be adequately represented by Bézier splines, and can meet with G^1 continuity. By adjusting the control points one step farther away than was done to achieve tangency, we can create two curves that are not only tangent, but which have the same radius of curvature where they meet (G^2).

With tangency, it is easy to use the control polygon to visually and arithmetically join the two curves at that level of smoothness. Since curvature is a more advanced level of modeling that requires more math to decribe and get right, there are

several tools available for the designer and modeler to visually and intuituively grasp how the curves need to be adjusted.

Author’s note: I am only speculating, but I believe that if the length and angle of the second leg of the control polygons were the same, then the curvature of the Bézier splines would be the same there as well. Caveat emptor.

In order to calculate the curvature of a spline, one needs to make use of the first and second derivatives of the curve to calculate the signed curvature κ (kappa).

$$\kappa = \frac{F'(x)F''(y) - F'(y)F''(x)}{[F'(x)^2 + F'(y)^2]^{(3/2)}}$$

Formulas for the first and second derivatives of Bernstein polynomials are available, and so assembling all of the necessary terms to calculate the curvature is possible.

Curvature combs

Once this is accomplished, a very useful CAD/CAGD-style tool called a curvature comb is available to compare the curvatures of adjacent Bézier splines. By plotting cylinders or surfaces along the spline that stand away from it by the amount of curvature at each point, the curvature of two splines can be visually (and numerically) compared. Color-coding by curvature is possible, and which side of the spline the comb lies on indicates whether the spline has a positive (convex) or negative (concave) curvature in that region.

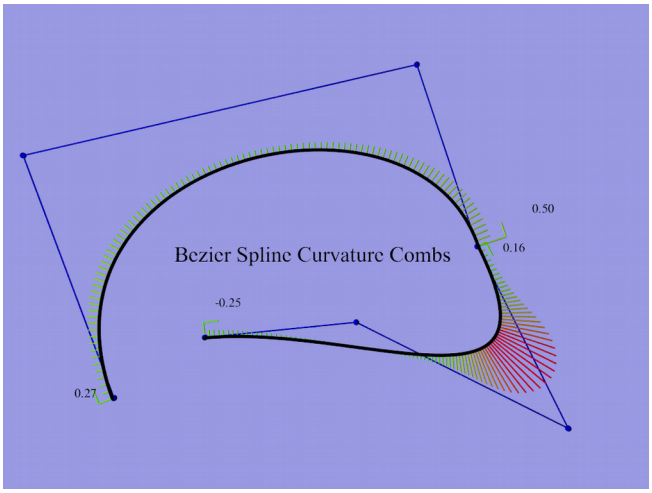


Illustration 12: Curvature comb showing mismatched spline curvatures

It took about a week or two of evenings to research all of the aspects of splines, smoothness, and curvature presented here, and another week to filter out false leads and actually code the equations for the derivatives and instantaneous spline curvature. A series of renders verified that all of the calculations were right for the tangent, concavity, curvature, and finally the spline-length combs.

[visualize curvature of surface with cylinders or color map: WIP]

Zebra lines

A simpler and less arduous method that relies on the visual appearance of a surface is called

an isophote. These are lines of constant brightness that can be generated on a surface by illuminating it with an array of parallel light sources, such as fluorescent tubes. The result is a black and white striped pattern distributed over the reflective surface. Digitally, this can be readily simulated using a single patterned light source that is rendered invisible in the scene. The “light source” can be as simple as a *sky_sphere* with a light-emissive texture:

```
texture {pigment {gradient x}
finish {emission 1}}
```

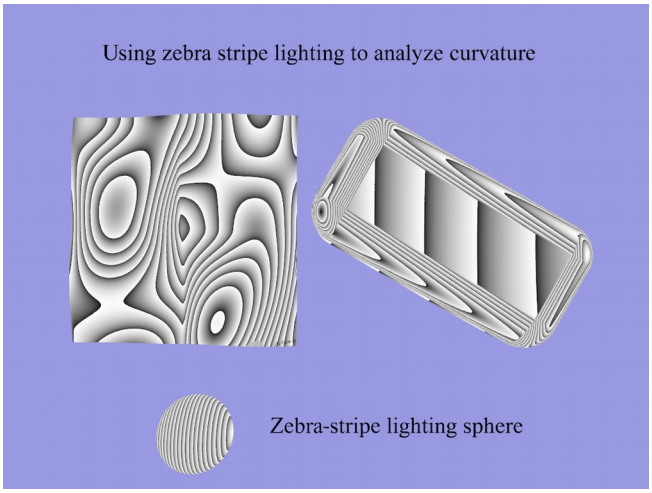


Illustration 13: Using "zebra-stripe" reflections to visualize changes in curvature

The heightfield on the left shows a surface generated by a smooth algebraic function, and the lines are smoothly curved. The rounded box on the right shows the abrupt angular transitions where the curvatures of the adjacent surfaces suddenly change.

These zebra stripe lines, being reflections, are dependent upon a static viewing position, and so moving real-world objects or changing the arrangement of digital models through lighting, rotation, translation, or camera position change the patterns of lines.

Non-reflective surfaces may appear smooth with G1 continuity, but if the surface is reflective, it may show irregularities due to mismatched curvature. Matching the curvature will produce smooth reflections. An additional level of smoothness, G3 – where the rates of curvature change were the same, would produce smooth and even reflections.

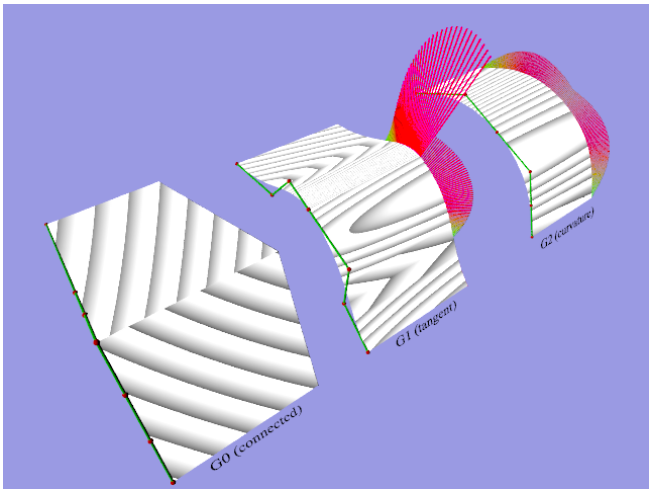


Illustration 14: Pairs of Bézier surfaces showing the changes in reflections generated by different levels of geometric continuity and smoothness

Digital Isophotes

In order to overcome the limitations of the reflective

isophote, a very similar method relying on how a surface is digitally patterned can be used to make isophote lines. The angle of the surface compared to some reference vector is used to generate the color map of the object. This is trivially accomplished in POV-Ray using the *slope* pigment pattern.

```
#declare isophote_vector = x;
slope {isophote_vector}
```

Changing the scale of the pattern produces thinner, more frequent bands, and defining a *color_map* to create sharp lines will produce a pattern that is more like the elevation isolines on a geographic terrain map.

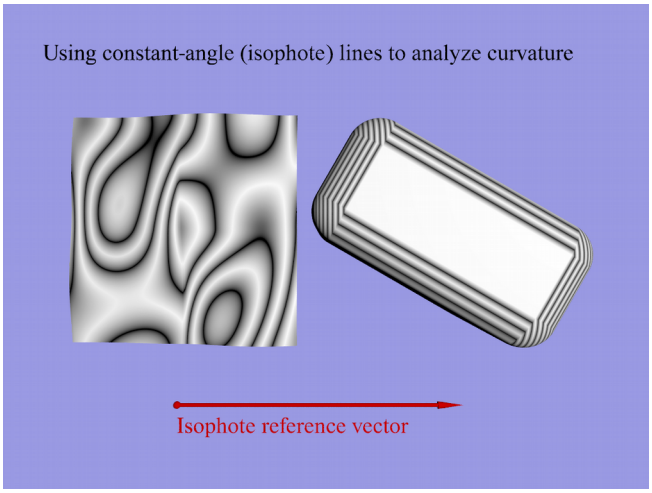


Illustration 15: Using a digital isophote pattern to visualize surface curvature

Since the pattern is dependent upon a vector that can be rotated and translated with the objects, it is invariant to viewing angle and can be animated or matrix transformed in a modeler without changing the surface pattern.

Bernstein Polynomials

The Bernstein polynomial is a linear combination of a set of basis polynomials developed by Sergei Natanovich Bernstein as a means of proving the Weierstrass approximation theorem: *Any continuous function on a closed and bounded interval can be uniformly approximated on that interval by polynomials to any degree of accuracy*.

This means that, given a curve – ANY curve – if we introduce a sufficient number of terms and raise the degree high enough (employ pow (x, 27), etc.) then we can get arbitrarily close to the actual shape of the curve using only polynomial equations.

Polynomial equations are used because they are well defined functions that are easily manipulated, are differentiable, which is important when continuity and curvature must be exact, and there is a fair degree of control over the complexity of the curve's shape. There are trade-offs when employing polynomials for modeling and other practical purposes. Quadratic curves don't have many control points, or degrees of freedom, and have no inflecion points – where the curve changes direction. Having only 3 control points, they are also completely planar. Trying to interpolate an entire curve with higher order

polynomials becomes costly to evaluate, and suffers from a progressively more pronounced oscillation effect that is called Runge’s phenomenon, discovered by Carl David Tolmé Runge in 1901 when investigating the magnitude of polynomial interpolation errors.[4] As you can see in the following graph, the polynomial correctly intersects all of the points, but deviates more from the average of the data the farther from the center it gets. The other problem with the use of polynomial interpolation is numeric instability: slightly changing a single point can drastically change the shape of the entire curve. There is also no unique polynomial which intersects a given set of points, and no well-defined way to determine which of the infinite number of polynomials best fits any given curve.

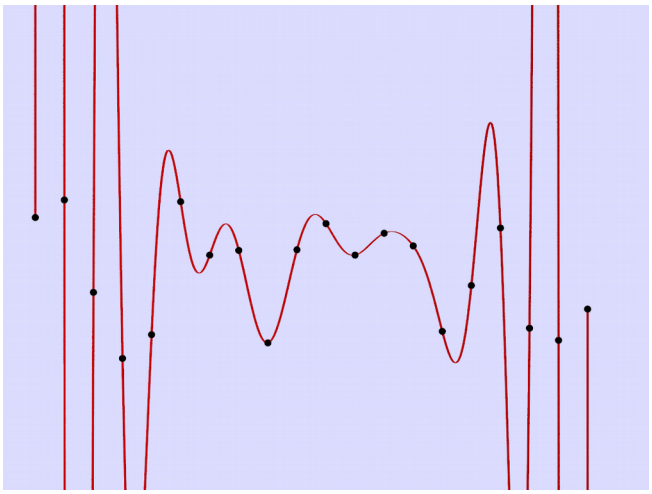


Illustration 16: A Lagrange polynomial interpolation showing the serious “ringing” problem of Runge's phonomenon

With quadratic splines, changing one of the control points will result in a large change to the whole curve. With cubic splines, perturbations decay exponentially, with each point further away from the change, the magnitude of the deviation is multiplied by -0.268.[5,6]

A practical compromise is a cubic Bézier spline which has an inflection point, and is the minimum degree to be able to twist the curve in 3-dimensional space. It’s also the lowest degree that allows separate control of the two endpoints and their derivatives. Long curves are typically composite splines composed of contiguous, smoothly-joined cubic Bézier splines. Curve segments requiring detailed shaping, or very high degrees of continuity and smoothing can be easily be degree elevated or are initially modeled as splines of degree 6 or 7.

Although all of the detailed mathematics may at first sound complicated, the basic principle is the same as our initial linear interpolation example between 2 points. In the case of the linear interpolation, as soon as t increases, the effect of the starting point begins to diminish, while the effect of the end point progressively grows. The influence of one point gets “handed off” to the other. With the summed basis polynomials

describing a Bézier spline, as soon as *t* increases, the effect of the starting point begins to diminish, while the effect all of the other control points begins to grow. Once the second polynomial peaks, the influence of the second control point is at a maximum, and then the third control point begins to dominate the position of the spline. Finally, the last polynomial progressively increases while the influence of all the other control points fades completely out. This is why the spline actually passes through the endpoints but is only influenced by the “internal” control points. The sum of all the basis polynomials at any given value of *t* is always one.

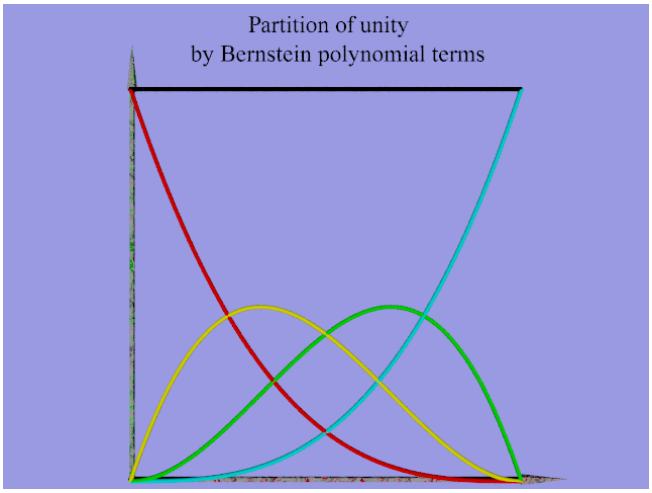


Illustration 17: The Bernstein polynomial terms for a cubic Bézier curve. The black line is the sum of the colored curves.

For a bicubic Bézier surface, the same thing happens, only in two dimensions. This is probably difficult to visualize, but it works much like the rows and columns of a spreadsheet. In either dimension, there is a complete cubic basis spline that is multiplied by the value of an orthogonal Bernstein function as it progresses in that direction. Again, the sum of all the basis polynomials at any given value of *u* or *v* is always one. This gives us a unit square. If the points are the result of control point coordinates multiplied by the product of all the basis polynomials at that point, then the unit square is curved and stretched to give the final surface of the bicubic Bézier patch.

Illustration 18: The sixteen Bernstein basis polynomials over the domain of [u, v] rendered as individual isosurface functions, with the sum of all 16 functions illustrating how they all add up to 1 over the entire domain of the patch

Indeed, since all of the functions are simply used to compute the vector components of

every point on the surface of the patch, a bicubic Bézier patch can be created using functions composed of the x, y, and z values of the control points, the Bernstein basis polynomials, and then rendered as a parametric isosurface {} object in POV-Ray. This is much slower than rendering an actual bicubic_patch primitive, but one of the nice advantages of this method is that there is much greater control over the detailed appearance of the surface.

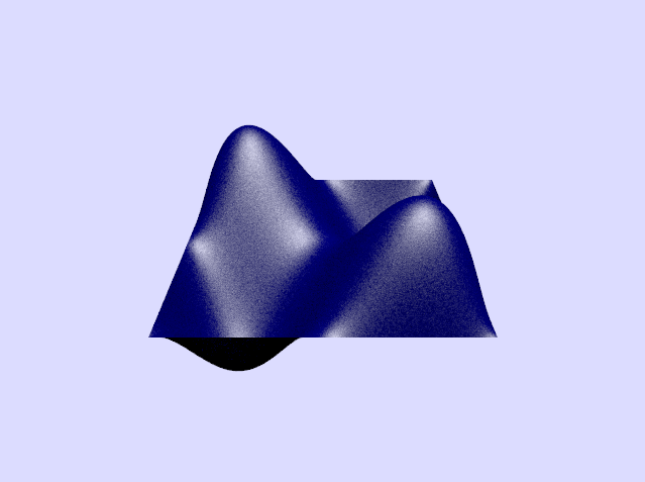


Illustration 19: The product of all 16 Bernstein polynomials and control points describing a bicubic Bézier patch, rendered as a POV-Ray isosurface {} object

To illustrate an analogy, we can think of a flat plane as an infinite series of adjacent individual lines, all placed side by side. In the same way, we can visualize a Bézier surface as an infinite number of side by side Bézier splines – in either direction. This is just like considering something either as a collection of columns, or a collection of rows. And of course, they are both. This can be visualized as the orthogonal threads in a piece of fabric: the *v* direction forming the warp, with the *u* direction forming the weft. And with complete control of the surface in function form, we can use the Bernstein polynomials and the control point values to modulate POV-Ray’s internal f_mesh1 function to provide a more concrete visual example.

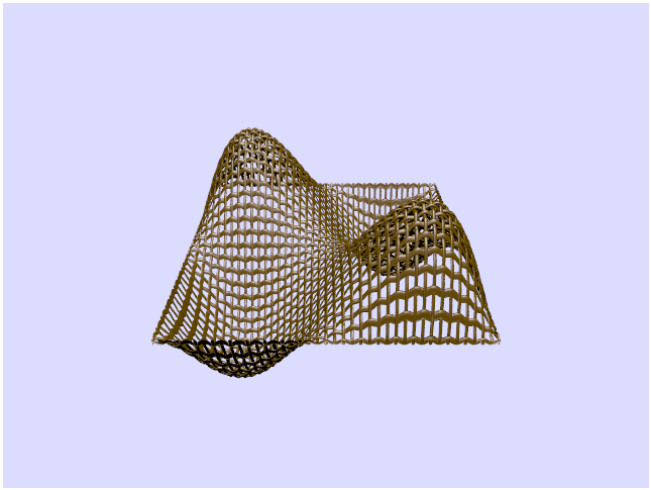


Illustration 20: The same functions as in Illustration 17, used to modify POV-Ray's internal f_mesh1 function

So in the same way that a piece of cloth deforms according to the position of an object it is draped over, the warp and the weft both moving in response to the object, so the interdependent splines of a Bézier patch change in response to the positions of its control points. A plethora of interesting modifications can

be made to the basic Bézier surface in this way, using functions to effectively create a displacement map of the surface that is independent of the degree of the Bernstein polynomials and number of control points.

Degree Elevation

Without going into too much detail, it is useful to know that Bézier splines of one degree can be represented by a Bézier spline of a higher degree. Therefore, a Bézier spline with only 4 control points can be converted to one with 5 or 6 control points, still have exactly the same shape, and the exact coordinates of the new control points can be easily calculated.

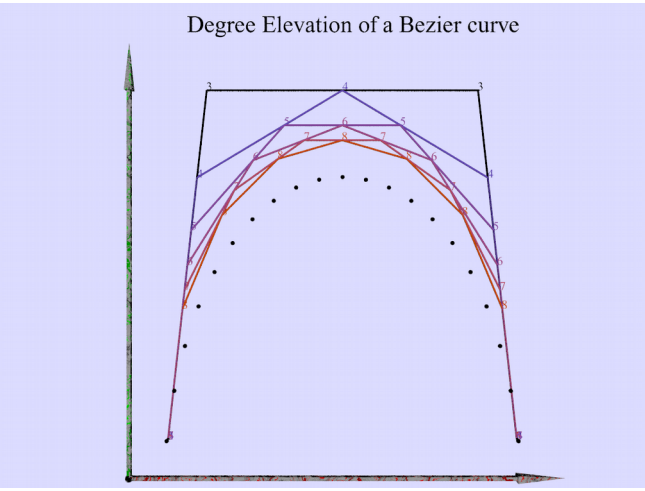


Illustration 21: The control grids of a Bézier spline showing the progressive degree elevation of a cubic spline with 4 control points, to one of degree 8, with 9 control points. The spline shape and position of any point on the curve at t stays exactly the same.

This is useful for situations where more detail needs to be represented, or a finer degree of control over the shape of the curve is needed. In smoothing continuous curves, the layers of control points for each level of smoothing move farther and farther "into" the array from the endpoints, and for high levels of smoothness, there may not be a sufficient number of control points to accommodate the changes dictated by the adjacent curves on either end of the spline being modified. Elevating the degree of a cubic Bézier spline to a quartic or quintic gives a spline of the same shape but with more control points, so the influences from adjacent segments on either end of the curve don't overlap.

Bézier Triangles

There are other ways in which Bézier splines can be combined to form surfaces. On 14 Aug 2018, user JimT created, *"A Bezier triangular patch, based on (u+v+w)^3 = 1, where u, v and w (w=1-u-v) are triangular co-ordinates. Using a singular quadrilateral patch could cause problems / wrinkles. However, trying to smoothly join Bezier triangular patches is even more of a mug's game than trying to smoothly join Bezier quadrilateral patches since there is only one internal point."*

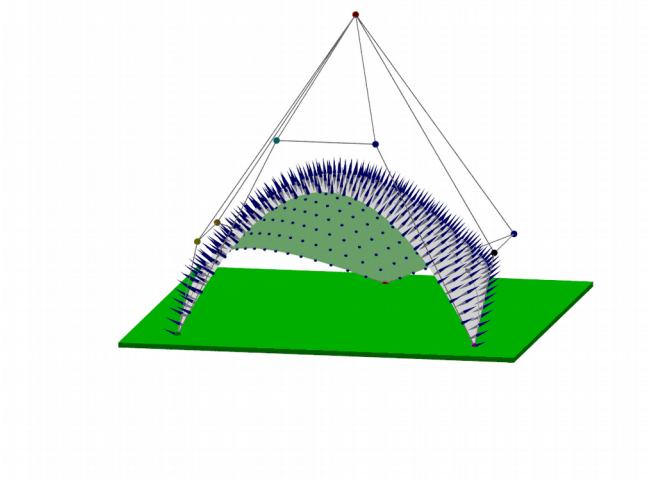


Illustration 22: JimT's Bézier triangular patch

Not realizing it at the time, the author now notes that if degree elevated splines were used, there would be sufficient control points to do so. There are indeed known methods to do so[7,8]

Rational Bézier splines

Bézier bicubic patch syntax and implementation in POV-Ray

Nearly all of the source code for implementing Bézier patches in POV-Ray was written by Alexander Enzmann. The syntax for rendering a single Bézier bicubic surface is:

```
bicubic_patch {
   type 1
   flatness 0.01
   u_steps 4
   v_steps 4
   uv_vectors
   <0,0>,<1,0>,<1,1>,<0,1>
   CP1,  CP2,  CP3,  CP4,
   CP5,  CP6,  CP7,  CP8,
   CP9,  CP10, CP11, CP12,
   CP13, CP14, CP15, CP16
   uv_mapping
   texture {myTexture}
   interior_texture {T_interior}
}
```

The type keyword is required, and may be 0 or 1. It describes how POV-Ray stores the patch data in memory. If it is 0, then only the corner points are stored in memory, and all of the parts of the patch are calculated at render time. If it is 1, then the patch is preprocessed into subpatches that are stored in memory, thus reducing the render time. (*Author’s note: values of -1 and 2 have both been tested and produce no parse errors. -1 appears to yield results equivalent to 0, and 2 appears to be equivalent to 1*)

The u_steps, v_steps and uv_vectors keywords are optional, and are only necessary if uv-mapped textures or a flatness < 1 are employed. It is worth noting here that the uv-vectors listed above the array of control points are applied starting at CP1, and then if the control points are arranged in a 4x4 matrix as they are in the example given, they then proceed in a clockwise fashion as the control points actually appear in the code. That is: CP1, CP4, CP16, CP13.

The flatness keyword is optional, and defaults to 0 if not supplied. Nonzero values are used to determine to what degree

the adaptive subdivision of the patch is suppressed. A flatness of 1 causes POV-Ray to restrict the patch to being completely flat. Intermediate values need to be tested to ensure that adjacent areas of the patch are adequately subdivided such that there are no “cracking” artefacts in the surface.

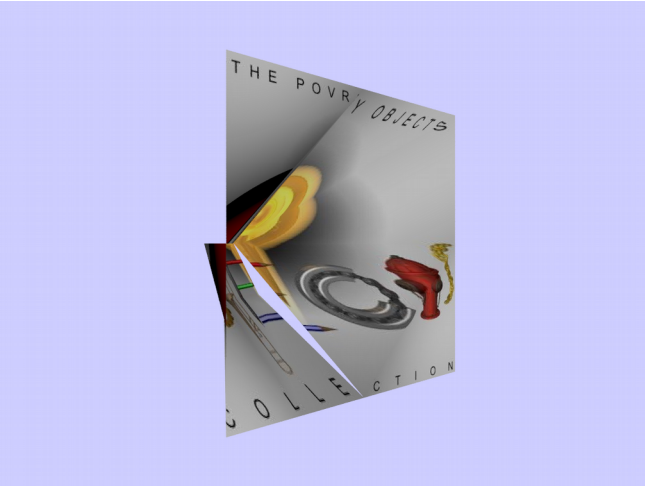


Illustration 23: A severely cracked bicubic patch. u_steps 3, v_steps 3, flatness 0.7

If the value for flatness is 0 POV-Ray will always subdivide the patch to the extent specified by u_steps and v_steps. POV-Ray divides the patch into a maximum of 2^{u_steps} x 2^{v_steps}. If u_steps and v_steps are not specified, they default to 0, and the patch is rendered as a flat square made of two triangles.

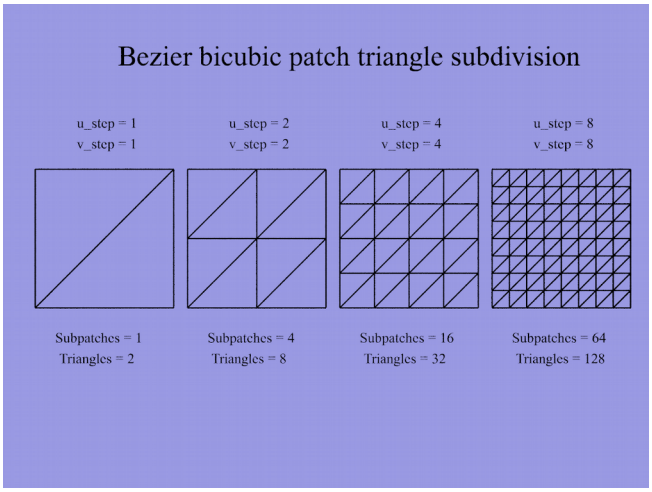


Illustration 24: How POV-Ray subdivides patches into triangles based on step values

Curious as to whether it were possible to assign fractional values, a series of numbers that are base2 logs of integers was calculated. These were tested in order to try to increase the number of divisions per side incrementally, rather than simply assigning integer values to *u_steps* and *v_steps* that result in large jumps in the amount of surface subdivision.

N per side	N log 2	Subpatches	Triangles
1	0.0000	1	
2	1.0000	4	
3	1.5850	9	1
4	2.0000	16	3
5	2.3219	25	5
6	2.5850	36	7
7	2.8074	49	9
8	3.0000	64	12
9	3.1699	81	16
10	3.3219	100	20
11	3.4594	121	24
12	3.5850	144	28
13	3.7004	169	33
14	3.8074	196	39
15	3.9069	225	45
16	4.0000	256	51

Table 1: Divisions per side, value (N log 2) to use for n_step, and stats

This does not work in the current version, as apparently the floor value of the exponent is used to calculate the number of subdivisions.

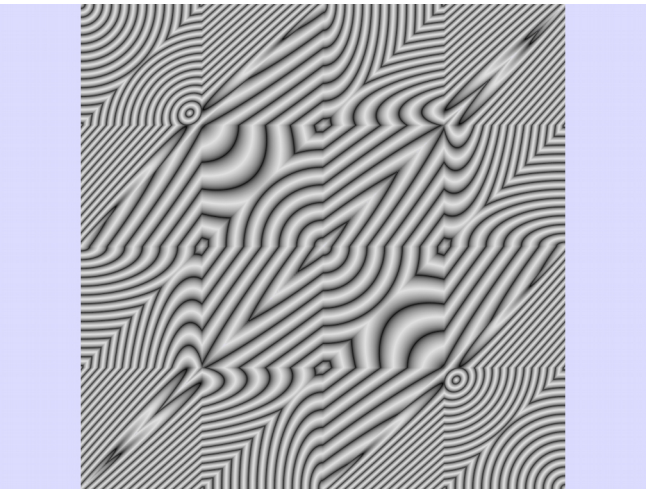


Illustration 25: A bicubic_patch patch specially curved and using digital isophotes to highlight actual triangles and levels of subdivision by POV-Ray. u_steps = 2 v_steps = 2

Uses for Bézier splines and bicubic patches

Bézier splines are frequently used to encode the 2-dimensional shape of typefaces and drawings, because rather than concretely defining the absolute position of points, as in a digital image, they describe how the shape is constructed between certain key points. This allows them to be smoothly rendered at any resolution sufficient to display the shape. Enlarging or scaling a digital image results in a coarse, blocky appearance due to its static nature, whereas rendering a typeface or SVG drawing on a large, high-resolution screen or printout maintains the smooth appearance of the whole shape since the regions between the control points are interpolated at the given resolution of the device.

Since Bézier surfaces are simply defined and easily manipulated, flexible objects such as flags, ribbons, cloth, and skin are often modeled using bicubic patch objects.

Bézier splines are also extensively used in animation, for controlling the position of objects, the speed of their movements, and the paths of the camera around a scene. This can give a much smoother and physically realistic effect than other methods which can appear jerky and unnatural.

Instead of 3-dimensional spatial vectors, if the control points are rgb or other values in a given color space, splines and surfaces can be used to control color blending.

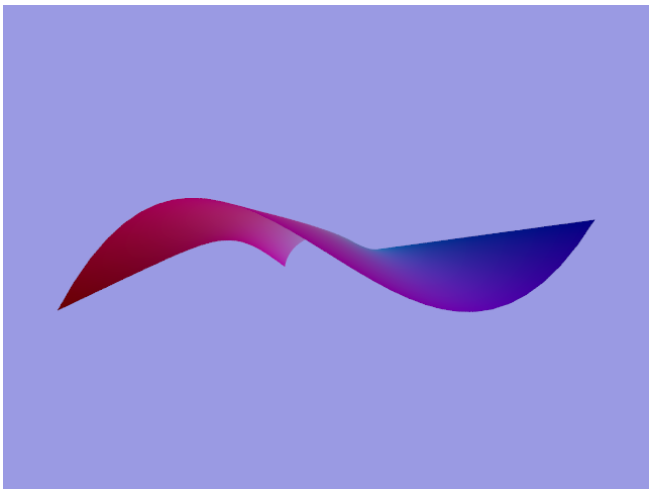


Illustration 26: A 4-color Bézier patch with bilinearly interpolated surface coloring. Ribbons and backgrounds such as this are typically seen in presentation slides.

Unlike a usual implicit mathematical function which yields a unique value for any given values of *x*, *y*, and *z*, a parametric function can double-back on itself and even self-intersect. This makes using Bézier splines and surfaces useful for modeling such things as knots, crumpled paper, rocks and complex geologic formations, etc.

Implicitization, Parametrization and Intersection

For any parametric curve, an implicit polynomial equation exists that describes exactly the same curve. Likewise, for any parametric surface, there exists an implicit equation that describes exactly the same surface. The process of finding the implicit equation of a parametric curve or surface is called implicitization.

An inversion formula for a parametric curve can be derived using polynomials. If the parametrization of a curve is a generally one-to-one map between parameter values and points on the curve, the inversion formula returns the parameter value t corresponding to a point (x, y) that lies on the curve.

Algebraic methods also can facilitate the design of algorithms for computing intersections between curves and surfaces.

CONCLUSION

Bézier splines, surfaces, the de Casteljau algorithm, and related approximation techniques are readily and comparatively easily implemented in code and graphics software such as POV-Ray to model complex and interesting organic and fluid forms that are difficult to create and control by other methods. By using tools for analyzing continuity and curvature, surfaces and animation transitions can be designed to give high-quality, pleasing results that can be otherwise hard to achieve. By better understanding the fundamental ideas of basis functions,

compound splines, and Bernstein polynomials that are commonly used to create curves and surfaces, it is hoped that users can now begin to experiment with these basics and expand on the underlying techniques to create new and interesting forms and effects in 3-dimensional computer graphics.

ACKNOWLEDGEMENTS

I would like to thank Tor Olav Kristensen for his invaluable guidance and patient instruction regarding stitching multiple Bézier patches and navigating the surface via coordinates generated with functions and macros that calculate the results of Bernstein polynomials. His beautiful renders and meticulous work serves as a source of continual inspiration.
http://dataduppedings.no/subcube/
https://github.com/t-o-k

Without question, Christoph Lipka always took the time and effort to look through code, stand his ground when making me see the error of my ways, and went above and beyond the call of an open-source developer to debug secenes and isolate and identify the source of user error.

M. Jérôme Grimbert richly deserves much appreciation and respect for his helpful and insightful comments, timely corrections, snippets of code, and impeccable examples of tackling thorny problems in his extensive Hgpovray38 fork.
http://wiki.povray.org/content/User:Le_Forgeron
https://github.com/LeForgeron

Richard Callwood III’s sphere sweep approximation greatly facilitated the work of preparing the many figures in this paper, and his overnight provision of supplementary code for properly rendering quadratic Bézier curves was a welcome surprise.

William F. Pokorny has provided his expertise and tuteledge in crafting functions, and directing me through some of the trickier syntax subtleties.
http://wiki.povray.org/content/User:Wfpokorny
https://github.com/wfpokorny/povray

Special thanks to Jeffrey Rohl for much encouragement, continued interest, constructive criticism, intellectual stimulation, and proofreading the draft.

I would like to thank the many other members of the POV-Ray community for their friendship, support, interest, constructive criticism, and tireless efforts in keeping the project alive over the past 25+ years.

NOTES ABOUT THE ILLUSTRATIONS

To the greatest degree possible, all figures were rendered by the author using as much Bézier code as possible. For the cover page –

Top image: 3 Sep 2018, uv-mapped torus made entirely of smoothly stitched Bézier bicubic patches.

Center image: 8 Sep 2020, equation for Bernstein polynomial originally as an SVG file, converted to cylinders and quadratic Bézier sphere sweeps. Rendered as a difference {}

Bottom image: 30 Aug 2020, a simple uv-mapped Bézier bicubic patch with control grid

FURTHER READING

Spline types
- natural cubic spline
- B-splines
- Cardinal splines
- Catmull-Clark subdivision surface
- Catmull-Rom splines
- cubic Hermite splines
- Hermite splines
- I-splines
- Kochanek-Bartels splines
- Linear splines
- M-splines
- NURBS
- T-splines

Font and typeface design

Automotive design

Coons patches

NURBS

REFERENCES AND SOURCE MATERIAL

SYMBOLS and GLOSSARY of TERMS

n! Commonly called “n factorial”, the product of all positive integers less than or equal to n.

“n choose k” there are n-choose-k ways to choose an unordered set of k elements from a fixed set of n elements. This can be calculated n!/k!(n-k)! and yields the coeeficients of the x^k terms in the binomial expansion of $(1+x)^n$.

Σ “Sigma” denotes the sum of the following terms.

$\wedge$ “Wedge” operator. Can denote the cross product of two vectors, or the “exterior product”

Binormal vector — a vector perpendicular to the tangent and normal vectors (their cross product)

Frenet-Serret Frame - for a non-degenerate curve parameterized by its arc length f(*s*), the frame defined by the orthogonal tangent, normal, and binormal vectors

Isoline — A line connecting points of equal value, such as elevation, brightness, relative position (*u*, *v*), etc.

Isoparm - Lines of constant *u* or *v* values that run along a NURBS surface

Isophote — a line of constant angle

L’Hôpital’s Rule — $\lim_{x\to c} f(x)/g(x) = \lim_{x\to c} f'(x)/g'(x)$

Product Rule — $(f \times g)' = f' \times g + f \times g'$

Osculating circle – the tangent circle whose center lies on the inner normal of a curve and whose curvature matches the curvature at that point

NURBS – Non-Uninform Rational B-Spline

Normal vector – a vector that is perpendicular to a curve or surface at that point.

Osculating plane – a plane that meets a curve or surface with a second order of contact

Tangent vector – a vector that is tangent (intersects at only that point) to a curve or surface.

Draft notes:
Need to map curvature of a 2D surface
May include an appendix with code, macros, formulas, etc. or provide a link to pbsf.
bezier for circles and spheres
Additional illustrations:
Coons patch for comparison
standard texture vs uv_mapping
bicubic color blending

1 Bernstein, S. Démonstration du théorème de Weierstrass fondée sur le calcul des probabilités, Comm. Kharkov Math. Soc. 13 (1912), 1–2
2 Casteljau, Paul de Faget. (1963). Courbes et Surfaces à Pôles (p. 45, Technical Report). Paris: Citröen.
3 P. Bézier. Définition numérique des courbes et surfaces I. Automatisme, XI:625–632, 1966.
4 Runge, Carl (1901), "Über empirische Funktionen und die Interpolation zwischen äquidistanten Ordinaten", Zeitschrift für Mathematik und Physik, 46: 224–243.
5 (Matt Timmermans) https://math.stackexchange.com/questions/1006207/why-do-we-choose-cubic-polynomials-when-we-make-a-spline
6 https://nbviewer.jupyter.org/github/mtimmerm/IPythonNotebooks/blob/master/NaturalCubicSplines.ipynb
7 Farin, Gerald E. Curves and surfaces for computer aided geometric design : a practical guide. Morgan Kaufmann Publishers Inc., 2001.
8 Xiaoxiao Du, Gang Zhao, Wei Wang, Computer-Aided Design & Applications, 17(2), 2020, 362-383