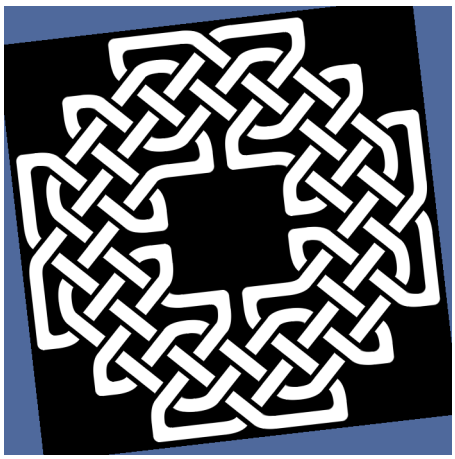


Beyond the pixel: towards infinite resolution textures

(or “Hardware rendering of implicit functions for curvilinear contours on surfaces in arbitrary resolution”, if you prefer a formal but boring title)

Stefan Gustavson, Linköping University, ITN (stegu@itn.liu.se) Internal report, February 16, 2006

Abstract

We propose a simple yet flexible method to encode and render curved contours in an arbitrary resolution on a 3D surface by using data from a low resolution texture map. Conic sections in implicit form, evaluated by second degree polynomials over the texture coordinates, are used to describe a straight or curved contour through each texel. Rendering of such contour texels can be performed at interactive rates with current mainstream graphics hardware.

Introduction and previous work

Textures are used everywhere in computer graphics, for realism and detail. Storing texture data as sampled pixel-based images limits their resolution and requires large amounts of data if the surfaces are to be viewed close up. Layered multitexturing with repeating detail textures for close-up views can sometimes mask the problem, but not fix it. Procedural texturing with shader programs can help, but traditional procedural patterns have focused on fairly simple and regular patterns and on random, noise-like functions.

In 2D graphics, artwork designs like symbols, text and other patterns with complex, irregular and sharp contours are now almost exclusively created and represented as scalable object contours (“vector graphics”), which may be rendered to an on-screen pixel representation in an arbitrary resolution, even on the fly for 2D animations. In 3D graphics, however, such patterns are still stored as sampled pixel images, even in off-line rendering. It would be helpful to have a procedural, resolution independent description of arbitrary curved contours that could be applied as a texture map to 3D objects, preferably applicable to real time rendering. A method based on pre-rendering and smart region caching of a vector graphics image in Adobe Illustrator format was presented at Siggraph 2001 [Haddon01] and has been developed further into a very useful DSO shadeop for RenderMan [Segal05]. However, that method has a high latency for cache misses, and it requires the CPU to render all the detail, and therefore it is unsuitable for interactive real time rendering.

Recent work by several authors [Tumblin04] [Ramanarayanan04] [Sen04] showed the advantages of storing information on discontinuities in textures by making a silhouette map. [Tumblin04] and [Ramanarayanan04] effectively encode edges as a train of straight line segments between points positioned anywhere inside the area of each texel. [Sen04] encodes curved contours, although in a manner unsuitable for hardware rendering. [Loop05] recently demonstrated that the implicit form of a quadratic Béziér spline contour can be evaluated efficiently in a graphics hardware fragment shader, and [Ray05] presented true vector texture maps, although implemented in a fairly complex way that has performance problems on current mainstream graphics hardware.

We propose a similar but less complicated method to encode and render curved contours on a surface by using data from a low resolution texture map. Our method constitutes a framework where patterns with curved contours of arbitrary shape and high complexity can be represented compactly and accurately, stored as texture data and rendered at an arbitrary resolution with very good real time performance in programmable graphics hardware.

There are significant similarities between the work presented here and [Ray05]. The main part of this work was performed independently, before that paper was published, but this paper does not present any significant and fundamental additional scientific contribution. We still think our approach is different enough to motivate this write-up, though. We also present a runnable demo with full source code, which is unfortunately lacking from [Ray05].

Parametric and implicit curves

Parametric curves are a familiar tool for both 2D and 3D graphics professionals. The most common flavor of parametric curves is the polynomial curve:

$$P(t) = \sum_{i=0}^{N} P_i B_i(t)$$

where P_i are control points (x_i, y_i) (for the 2D case), each of the basis functions $B_i(t)$ are polynomials of degree N or lower, and the parameter t is in the range $0 \le t \le 1$.

A useful variant is the rational polynomial curve:

$$P(t) = \left(\sum_{i=0}^{N} P_i B_i(t) \right) / \left(\sum_{i=0}^{N} B_i(t) \right)$$

Higher degrees N for the polynomials allow for more complicated shapes of the curve, but such curves are more easily approximated by a sequence of shorter curve segments of lower degree. For computer graphics applications, the degree of the polynomials are seldom higher than 3. Cubic curves (polynomials of degree 3) are most common, but quadratic curves (polynomials of degree 2) are sufficient if more segments are used. The PostScript page description language uses cubic curves, but TrueType font glyph descriptions use quadratic curves. Modern 2D graphics frameworks like Java2D and SVG provide both variants for flexibility.

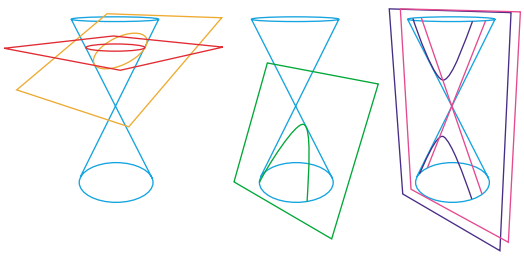
An implicit curve, on the other hand, is described in terms of the location of the zeroes of a function:

$F(x, y) = 0$ for points on the curve.

A theorem from classic algebraic geometry states that for every rational parametric curve description with basis polynomials $B_i(t)$ of degree no higher than N, there is a corresponding implicit form with a polynomial function $F(x, y)$ of degree no higher than N. For lower degree curves, finding the implicit form from the parametric form is quite simple, and for higher degree curves, there are formal methods to always find a closed form expression for $F(x, y)$. (The converse is not always true, though: for an arbitrary implicit curve with a polynomial function of degree N, there is no guarantee that there is a rational parameterisation using polynomials, or that those polynomials are of degree N or lower.)

Conic sections

Conic sections, the various possible curves of intersection between a cone and a plane, is a classic subject in algebraic geometry. Three classes of curves can be generated, depending on the angle between the plane and the cone: ellipses (including circles), parabolas and hyperbolas.



The general implicit equation for a conic section is:

$$F(x, y) = Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$$

This is actually a fully general second degree polynomial in x and y, and any rational quadratic spline can be implicitized to that form. Quadratic Béziér curves like the ones used for TrueType are non-rational quadratic splines, a subset of conic sections which are particularly easy to map to their corresponding implicit functions [Loop05]. All quadratic non-rational Béziér splines are inherently parabolas. General conic sections also encompass circles, ellipses and hyperbolas, and useful degenerate cases of the general form of the polynomial include straight lines, parallel lines and crossing lines of arbitrary position and angle. Thus, implicit curves described by second degree polynomials are a larger class of shapes than quadratic Béziér splines, and make a good candidate for a drawing primitive.

An extension to third degree polynomials and cubic contours is perfectly possible, but it requires more data and more processing without offering significant extra functionality. Furthermore, cubic parametric curves present some additional problems for implicitization which are a bit hard to overcome in the general case: cubic splines can have unwanted nearby strands and self-intersections outside the parameter range $0 \le t \le 1$, and as the implicit form has no notion of a parameter or any sequential trace along the path, those unwanted portions of the curve will be hard to get rid of in the implicit form. Similar problems arise for second degree curves as well, but they are more easily overcome. Most notably, second degree curves can not self-intersect, which makes them a lot more suitable for implicitization.

We use conic sections in their implicit form, evaluated by general second degree polynomials over the 2D texture coordinates, to describe a straight or curved contour through each texel. By encoding the polynomial coefficients directly as texture data, we can use piecewise quadratic segments with a different implicit curve for each texel to make up a closed contour of arbitrary shape and high complexity from only a small amount of texture data.

Rendering

The rendering of implicit contour texels is not complicated, and it can be performed at interactive rates with modern graphics hardware. The polynomial coefficients are stored in one or two textures as RGB data, the texture data is sampled without any interpolation, and the polynomials are evaluated over the relative texture coordinates within each texel using a fragment shader program. In simplified pseudo-code, the shader looks simply like this:

```
[A,B,C,D,E,F] = texture(s,t);
[x,y] = frac([s*texw, t*texh]);
Fxy = A*x*x+B*x*y+C*y*y+D*x+E*y+F;
color = step(0.0, Fxy);
```

An actual working fragment shader program in GLSL is found in the appendix, and a link to a runnable demo is found at the end of this presentation.

Anti-aliasing

For points close to the implicit curve, the value of the function $F(x, y)$ is a reasonable approximation to the orthogonal distance to the curve, so a smooth, anti-aliased edge can be rendered by applying a smooth step function to the implicit equation, i.e. by replacing `step()` with `smoothstep()`.

However, the polynomials need to be scaled to show a reasonably consistent gradient magnitude over a texel, and particularly across the edge between two texels. One method, proposed also by [Ray05], is to normalise the function value with the magnitude of the gradient at the point in question, approximating the distance to the curve as $d \approx F(x, y) / |\nabla F(x, y)|$. This is a first order approximation which is valid near the contour, and reasonably well-behaved also for points further away from the contour, with the obvious exception of points where $\nabla F(x, y) = 0$.

The gradient of a polynomial of degree N is a polynomial of degree $N - 1$, related in a trivial manner to the original polynomial and not difficult to evaluate. By calculating the magnitude of the gradient $|\nabla F(x, y)|$ in addition to the value of the implicit function $F(x, y)$, we can perform anti-aliasing of edges under an arbitrary magnification.

Expressed in pseudo-code, the antialiased contour would be rendered according to the following:

```
dstdx = length([dFdx(s),dFdx(t)]);
dstdx = length([dFdy(s),dFdy(t)]);
stepw = 0.5*length([dstdx,dstdy]);
[A,B,C,D,E,F] = texture(s,t);
[x,y] = frac([s*texw, t*texh]);
Fxy = A*x*x+B*x*y+C*y*y+D*x+E*y+F;
gradFxy = [2*A*x+B*y+D, B*x+2*C*y+E];
d = Fxy/length(gradFxy);
color = smoothstep(-stepw, stepw, d);
```

The appendix contains a full, working GLSL shader program to do this.

Minification

As was pointed out also in [Ray05], anti-aliasing on minification is difficult using the implicit form, but it can be handled by using a regular mipmapped texture instead of the implicit contour map when the area of a rendered pixel approaches the same scale as one texel. When the texels are so small that they are only about the size of a few rendered pixels, a regular pre-rendered texture map of reasonable size will do nicely, and it will behave well even under further extreme minification.

Performance

The fragment shader program for this method is simple enough to be handled well by modern graphics hardware, and its performance is good. Frame rates of thousands of frames per second can be acheived with current high-end consumer level hardware, and even low-cost mainstream graphics chipsets yield fully interactive frame rates.

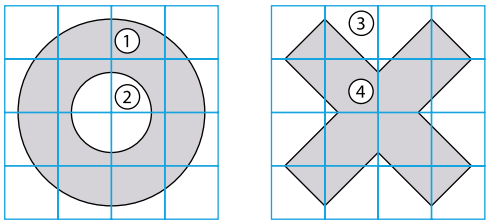
Pattern generation

The generation of the polynomial coefficients required for the contour texture involves a process known as implicitization. It is not a trivial procedure, but it is performed off-line, so interactive rates are not required.

If the contour is given as a sequence of lines or quadratic splines, the problem is fairly simple. Because our framework requires the same implicit equation to be used in an entire texel, there will be a need to move control points around and approximate the contour around the joints between segments with slightly different curve shapes, but for the most part, the conversion can be exact. If the contours are specified using cubic splines, the conversion will be approximate, but if the texels are chosen reasonably small, each cubic curve segment will be approximated with several quadratic curves, and the match will be very close.

Examples

We have not yet designed a general interactive drawing or conversion program for the pattern generation. As examples for demonstration, we first calculated by hand two simple patterns, a ring and a cross:



1) $x^2 + (y + 1)^2 < \left(\frac{7}{4}\right)^2 \Leftrightarrow 16x^2 + 16y^2 + 32y - 33 < 0$

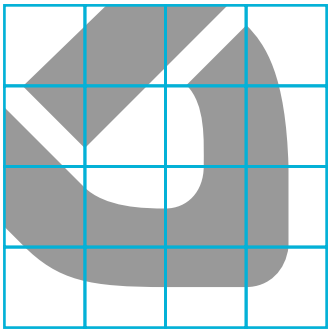
2) $x^2 + y^2 > \left(\frac{3}{4}\right)^2 \Leftrightarrow -16x^2 - 16y^2 + 9 < 0$

3) $x + y - \frac{3}{4} < 0 \Leftrightarrow 4x + 4y - 3 < 0$

4) $\left(x + y - \frac{1}{2}\right)^2 < \left(\frac{3}{4}\right)^2 \Leftrightarrow$

$16x^2 + 32xy + 16y^2 - 16x - 16y - 5 < 0$

To make a more interesting and less symmetric pattern, we also worked out polynomials for the shapes occuring in a more complicated but still fairly modular pattern: a Celtic knot on a square grid. The full pattern is in the title image, and a detail of a few texels is shown below. Rotations reflections of 10 different patterns make up a total of 46 texel polynomials, which were reasonably convenient to calculate by a simple Matlab script and re-use appropriately over a larger texture image for the entire pattern.



The Celtic knot design in the example was drawn using a Java program written for this purpose, and exported directly as a polynomial texture map.

Floating point texture data was not required for these two examples. Because we designed the patterns from scratch, the coefficient values could be deliberately chosen to be representable either exactly or with only minor loss of precision using 8-bit signed integer representation.

As you zoom in on these patterns, the curved edges will stay sharp until you hit the limit of the machine precision for the texture coordinates on your graphics card. Modern graphics cards have many bits of precision for the texture coordinate interpolation, so the pattern can be magnified a lot before any artefacts start showing. Situations where a single texel covers the entire display present no problems.

Restrictions and extensions

We would like to stress the point that, although the patterns we demonstrate are fairly regular and highly modular, there is no such inherent restriction on the types of patterns that are possible to represent within this framework. If a pattern is too complex to be represented accurately with a certain number of texels, it can either be redesigned to better fit the restrictions, it can be approximated, or a more dense grid of texels could be used to encode it better. However, there are some limitations that can be addressed to be able to encode a general pattern more accurately using fewer texels.

The restriction to 8 bits signed integer representation for the polynomial coefficients is troublesome. For a more general implementation, 16 bits integer or, better, floating point texture data should be used.

Our simpe demo implementation uses one single contour per texel, which places quite severe constraints on the possible positions of corners in the pattern. The method employed by [Ray05] could be used to remove that constraint: each texel can be defined not by a single polynomial, but by the minimum of two polynomials, and an extra sign to be able to render both outer corners and inner corners:

$$F(x, y) = \pm min(F_1(x, y), F_2(x, y))$$

This would require twice the amount of texture data for each texel, and the shader would take twice as long to execute, but the restrictions on the pattern design would be much alleviated.

The more general but also more complex nature of the solution presented in [Ray05] makes it unsuitable for most current graphics architectures. It requires conditional execution and nested dependent texture lookups, and the strongly non-coherent accesses to the texture has a tendency to trash the texture cache. All of these can be real performance killers, and all are avoided by our proposed solution. The performance of our demo on a GeForce 6800 is around 1000 fps. This method is fast enough for use today.

Demo and source code

A runnable demo of our implemenation, with full source code, is here:

http://staffwww.itn.liu.se/~stegu/GLSL-conics/

References

[Segal05] *VTexture DSO shadeop*, Alex Segal, software available for download at http://www.renderman.ru/vtexture/indexe.html

[Haddon01] *Implementing Vector-Based Texturing In RenderMan*, John Haddon and Ian Stephenson, technical sketch presented at Siggraph 2001, http://dctsystems.co.uk/Text/haddon.pdf

[Loop05] Charles Loop and Jim Blinn, *Resolution Independent Curve Rendering*, paper presented at Siggraph 2005

[Ray05] Nicholas Ray, Xavier Cavin and Bruno Lévy, *Vector Texture Maps on the GPU*, Technical Report 2005, http://www.loria.fr/~levy/publications/papers/2005/VTM/vtm.pdf

[Tumblin04] Jack Tumblin and Prasun Choudhury, *Bixels: Picture Samples with Sharp Embedded Boundaries*, Presented at Eurographics Symposium on Rendering 2004

[Ramanarayanan04] G. Ramanarayanan, K. Bala and B. Walter, *Feature-Based Textures*, presented at Eurographics Symposium on Rendering 2004

[Sen04] Pradeeep Sen, *Silhouette Maps for Improved Texture Magnification*, presented at Eurographics Symposium on Graphics Hardware 2004

Appendices

The implicit form of a rational quadratic Béziér contour:

If $x(t) = \dfrac{x_0(1-t)^2 + 2wx_1t(1-t) + x_2t^2}{(1-t)^2 + 2wt(1-t) + t^2}$ and

$y(t) = \dfrac{y_0(1-t)^2 + 2wy_1t(1-t) + y_2t^2}{(1-t)^2 + 2wt(1-t) + t^2}$, then the points

$(x, y) = (x(t), y(t))$ satisfy, for any value of t:

$Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$, where

$A = 4w^2(y_0 - y_1)(y_1 - y_2) - (y_0 - y_2)^2$

$B = 4w^2((x_0 - x_1)(y_2 - y_1) + (x_1 - x_2)(y_1 - y_0))$
$+ 2(x_0 - x_2)(y_0 - y_2)$

$C = 4w^2(x_0 - x_1)(x_1 - x_2) - (x_0 - x_2)^2$

$D = 4w^2((x_0y_1 - x_1y_0)(y_1 - y_2) + (x_1y_2 - x_2y_1)(y_0 - y_1))$
$+ 2(y_2 - y_0)(x_0y_2 - x_2y_0)$

$E = 4w^2((y_0x_1 - y_1x_0)(x_1 - x_2) + (y_1x_2 - y_2x_1)(x_0 - x_1))$
$+ 2(x_2 - x_0)(y_0x_2 - y_2x_0)$

$F = 4w^2(x_1y_2 - x_2y_1)(x_0y_1 - x_1y_0) - (x_2y_0 - x_0y_2)^2$

A GLSL fragment shader to render the implicit contours from polynomial coefficients

```
uniform sampler2D coeffsABC;
uniform sampler2D coeffsDEF;

void main( void ) {
 // Get the texture coordinates
 vec2 st = gl_TexCoord[0].st;
 // Read the polynomial coefficients
 // Fixed point 8 bit, 128/255->0.0
 vec3 ABC = vec3(
  texture2D(coeffsABC, st))-0.50196;
 vec3 DEF = vec3(
  texture2D(coeffsDEF, st))-0.50196;
 // width and height of the textures
 float texwidth = 32.0;
 // Construct powers of u and v
 vec3 uv1 = vec3(
  fract(st*texwidth), 1.0);
 vec3 u2uvv2 = uv1.xxy * uv1.xyy;
 // Evaluate the implicit polynomial
 float f=dot(ABC,u2uvv2)+dot(DEF,uv1);
 // Set the color to be black
 // when P<=0.0, white otherwise
 float a = step(0.0, f);
 gl_FragColor = vec4(a, a, a, 1.0);
}
```

A GLSL fragment shader to compute the same contours, with proper antialiasing. Blue color marks the differences to the version without antialiasing.

```
uniform sampler2D coeffsABC;
uniform sampler2D coeffsDEF;

void main( void ) {
 // Get the texture coordinates
 vec2 st = gl_TexCoord[0].st;
 // Read the polynomial coefficients
 // Fixed point 8 bit, 128/255->0.0
 vec3 ABC = vec3(
  texture2D(coeffsABC, st)-0.50196);
 vec3 DEF = vec3(
  texture2D(coeffsDEF, st)-0.50196);
 // Calculate the pixel size in (u,v)
 // space for antialiasing
 vec4 duvdxy = 32.0*vec4(
  dFdx(st), dFdy(st));
 float stepwidth = 0.5*length(duvdxy);
 // Construct powers of u and v
 vec3 uv1 = vec3(fract(st*32.0), 1.0);
 vec3 u2uvv2 = uv1.xxy * uv1.xyy;
 // Evaluate the implicit polynomial
 float f = dot(ABC,u2uvv2)
  + dot(DEF,uv1);
 // Compute the magnitude of the
 // gradient of the polynomial
 vec2 gradf = vec2(
 dot(uv1,vec3(2.0*ABC.x,ABC.y,DEF.x)),
 dot(uv1,vec3(2.0*ABC.z,ABC.y,DEF.y)));
 float g = rsqrt(dot(gradf, gradf));
 // Set the color to be black
 // when P<=0.0, white otherwise
 float a = smoothstep(
  -stepwidth, stepwidth, f*g);
 gl_FragColor = vec4(a, a, a, 1.0);
}
```