Direct rendering of cubic Béziér contours in RSL

Stefan Gustavson, Linköping University, Sweden (stegu@itn.liu.se)

The rendering of "vector graphics", resolution-independent parametric contours, is essential to many 2D graphics applications. The outline representation is very compact, and the pattern can be transformed freely and rendered at an arbitrary resolution. One way of dealing with this in the RenderMan framework is to use a pre-rendering DSO backend that accepts a vector graphics file as input and provides a texture-like access to a vector graphics pattern. This is done by the shadeop VTexture[1], inspired by a Siggraph sketch from 2001[2], and that is indeed a smart and very useful solution.

A stupid trick to try, on the other hand, is to refuse any external dependencies and cram the entire rendering of vector graphics into a pure SL surface shader.[3] That is what is presented here.

Implicit drawing

The basic problem is that in a procedural shader, contours cannot be drawn or painted. Instead, they must be expressed in an indirect and quite awkward manner as implicit functions over the texture coordinate domain, so that a shader evaluation at a single surface point can immediately decide whether that point is inside or outside the contour and color it accordingly. Some simplistic patterns with similarities to vector graphics can be found in the SL literature, mostly based on edges, rectangles and ellipses. They are quite useful, and reasonably simple shapes like the Aqsis logo below are fairly easy to express as implicit functions.

However, only a limited range of patterns can be expressed in terms of such a restricted set of drawing primitives. Stretching the use of ellipses and rectangles to its limits by using quite a lot of them, and also using 2D set operations like intersection and subtraction between basic shapes, we can even design character shapes, albeit not terribly good looking.

Béziér shapes

We want more than this. Ideally, we would like to use cubic Béziér curves like most 2D drawing programs do, and thus be able to render arbitrary contour shapes, like designs stored in EPS or SVG format, or character glyphs from a professional typeface. It is in fact perfectly possible to

1. "VTexture DSO shadeop", Alex Segal, http://www.renderman.ru/vtexture/indexe.html
2. "Implementing Vector-Based Texturing In RenderMan", John Haddon and Ian Stephenson, http://dct-systems.co.uk/Text/haddon.pdf
3. In the narrowly focused context of SL and RenderMan, this is indeed a stupid and (as we shall see) rather painful exercise, but in a larger perspective, it can be applied to hardware shaders written in GLSL or HLSL, and then it is no longer quite as stupid, because it removes the delay and the CPU overhead with having a pre-rendering backend. Compared to off-line rendering, the range of viewing scales for real time applications is generally wider, and often unknown at the time of content creation.

express such a primitive in a shader. The normal way of drawing a Béziér curve is by using its parametric form:

$$\begin{aligned} x(t) &= x_0(1-t)^3 + 3x_1t(1-t)^2 + 3x_2t^2(1-t) + x_3t^3 \\ y(t) &= y_0(1-t)^3 + 3y_1t(1-t)^2 + 3y_2t^2(1-t) + y_3t^3 \end{aligned} \quad , 0 \le t \le 1$$

To use such a contour in a shader, we need to express it as an implicit function instead. The conversion is a complex but mathematically well investigated process known as *implicitization*. A famous theorem from analytic geometry states that *any* parametric polynomial curve of degree N can alternatively be defined as the zeroes of a 2D polynomial function $F(x, y)$ also of degree N, so that the points on the curve $(x, y) = (x(t), y(t))$ are those that satisfy the equality $F(x, y) = 0$. The interior points of the contour satisfy the inequality $F(x, y) > 0$, and exterior points satisfy $F(x, y) < 0$ (or vice versa).

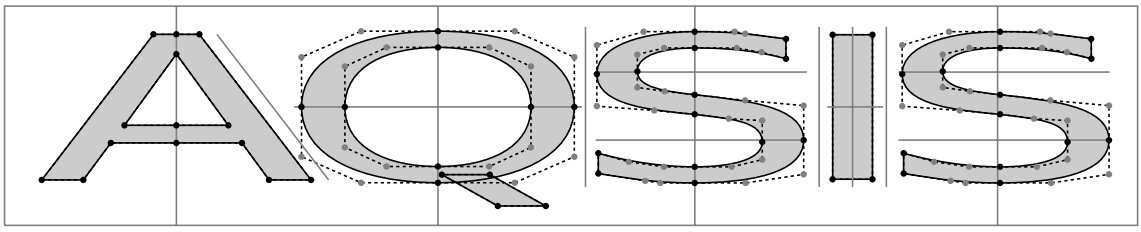
The implicit form is well defined but really hairy to derive, so a symbolic math package like Maple or Mathematica is useful to do all the tedious equation solving. Relying on the theorem above, we know that the implicit form of a cubic Béziér segment, in fact of any rational polynomial of degree 3, is a general third degree polynomial:

$$F(x, y) = Ax^3 + Bx^2y + Cxy^2 + Dy^3 + Ex^2 + Fxy + Gy^2 + Hx + Ky + L$$

There are closed-form but complicated expressions that relate each of the polynomial coefficients A to L to the Béziér control points x_i, y_i. The full expressions are given in the appendix.

Joining several Bézier segments

A contour made from Béziér curves is not a single function for the entire shape. Several segments are used, and joined together at their endpoints. To make this work in a shader, we must express the shape as a set of several different implicit functions, each with their own particular area of influence. The partitioning can be made in a fully general way if needed, e.g. by partitioning the plane into cells with linear boundaries, each containing only one segment or part of one segment, but here we will resort to some manual hand-tuning of our contours to make the programming easier. The pattern chosen for this demo was the five characters in the Aqsis logo, this time in the proper typeface:



Note that the anchor points of these shapes were hand tuned to line up nicely both vertically and horizontally. This makes it possible to express all the characters as a number of rectangular cells, each containing one or two curves each. The A and the I are special in that they only contain linear boundaries, so we can design them using simple linear polynomials only. The tail for the Q is treated similarly. The round part of the Q is split into four quadrants, and each S is split into six rectangles, as outlined in the figure.

With some care, the painted part of each rectangular sub-region can now be expressed as the min of two step functions. The result is a Boolean AND of the interior of the two contours:

$$F(x, y) = min(step(0, F_1(x, y)), step(0, F_2(x, y)))$$

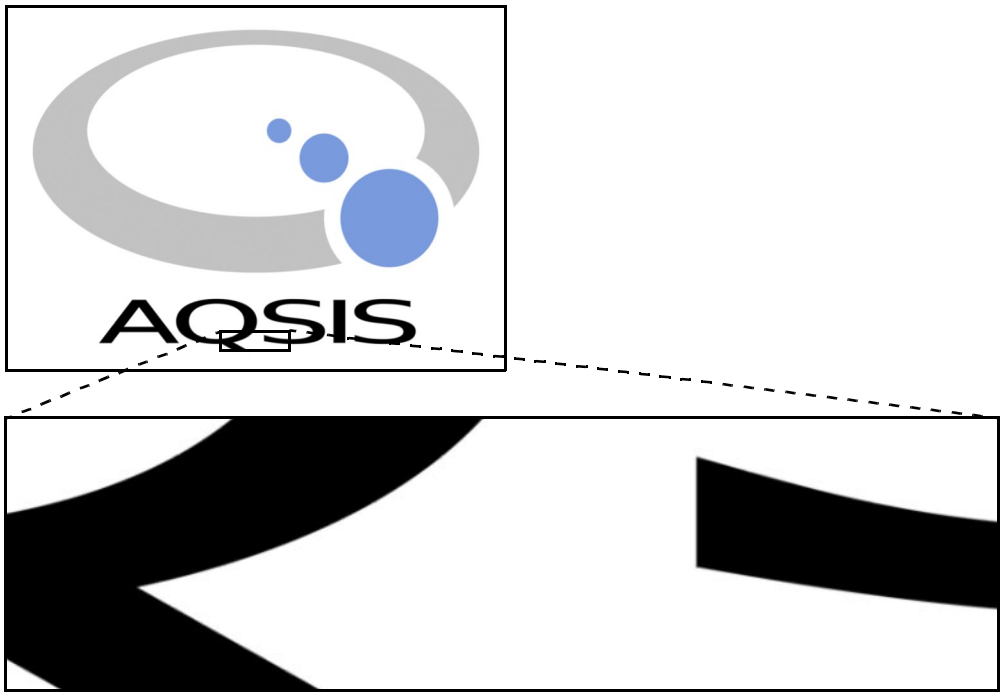
Antialiasing

A key issue for any shader is antialiasing, and it is even more important for a shader like ours that encodes sharp boundaries. We could do it with `filterstep()`, and that would work if we put our mind to it. However, because we have so many different regions, we want simple early-out tests to decide which region we are in, in the form of nested `if-else` statements or some other multiple choice based on array indexing, or else the shader would spend most of its time evaluating polynomials that do not influence the part of the surface that is actually being shaded. Conditional statements in SL break the automatic derivatives that are required for `filterstep()` to work, so we will see ugly artefacts at the decision boundaries if we try to use `filterstep()` inside conditionals. Instead, we choose to do our own analytic antialiasing. The implicit functions are third degree polynomials, so their gradients in the (x, y) plane consist of second degree polynomials:

$$\nabla F(x, y) = \begin{bmatrix} 3Ax^2 + 2Bxy + Cy^2 + 2Ex + Fy + H \\ Bx^2 + 2Cxy + 3Dy^2 + Fx + 2Gy + K \end{bmatrix}$$

These gradients present only a reasonable amount of extra computations and no extra coefficients, and they let us do the `if-else` conditionals for the partitioning without messing up the antialiasing. The only thing we need the automatic derivatives for is to find a good filter width for the shader in local coordinates, and that can be performed first of all, before we do any conditionals. The rest can be performed by explicit `smoothstep()` functions and a properly chosen step width.

The conditionals will in fact make the shader have reasonable, even good performance. Even though the shader for the example text ends up having a lot of code, only a small part of that code will be executed for each surface point. For each shader invocation, we get away with evaluating a small number of simple linear boundary conditions, and at most two cubic polynomials and their gradients.



The code for the final shader is presented in the appendix. It copes with extreme magnifications, limited only by the numerical floating-point precision of the SL implementation. Minification is not quite as graceful, though. When the curved shapes become too thin, around a couple of pixels across, the antialiased edges will be too blurry and the gradient filtering will be messed up, so the characters will break. If minification is important it should be handled by a regular mipmapped texture image. That texture image can be rendered by the same shader and saved to a file in a manner similar to a shadow pass. When the shader is used with extreme minification, a texture lookup should be chosen instead of the implicit description.

Appendix A: the source for the Aqsis logo with good looking text

SL surface shader

```
/*
 * aqsislogo.sl : A shader to draw the Aqsis logo in 2D,
 * with text in the proper typeface.
 *
 * Author: Stefan Gustavson (stegu@itn.liu.se) 2006-02-05
 * As far as I know, this is the world's first SL-only
 * rendering of general Bezier contours.
 *
 * This shader does analytic antialiasing by explicit gradient
 * calculations, to avoid artefacts from the conditionals.
 * Despite its complexity, it executes reasonably fast.
 * Most of the heavy math in is inside conditionals, and for
 * each evaluation, only a small part of the code is executed.
 *
 * This shader requires the following RIB option to look good:
 * ShadingInterpolation "smooth"
 * The remedy for renderers which do not implement that is to use
 * "PixelSamples 4 4" instead, that will hide the problems.
 * It takes longer to render, but looks good.
 * You should still use "ShadingRate 1", though, this shader was
 * designed for that, so a lower value will not do much good.
 *
 * Lighting has absolutely no influence on this shader.
 * The colors are always fully bright.
 */

// Evaluate a general cubic polynomial with coefficients from an array
float cubic(float Pc[]; float x; float y) {
    return Pc[0]*x*x*x + Pc[1]*x*x*y + Pc[2]*x*y*y + Pc[3]*y*y*y
    + Pc[4]*x*x + Pc[5]*x*y + Pc[6]*y*y + Pc[7]*x + Pc[8]*y + Pc[9];
}

// Calculate the gradient magnitude of the polynomial
float gradmag(float Pc[]; float x; float y) {
    float gx = 3*Pc[0]*x*x + 2*Pc[1]*x*y + Pc[2]*y*y + 2*Pc[4]*x + Pc[5]*y + Pc[7];
    float gy = Pc[1]*x*x + 2*Pc[2]*x*y + 3*Pc[3]*y*y + Pc[5]*x + 2*Pc[6]*y + Pc[8];
    return sqrt(gx*gx+gy*gy);
}

#define min_x 66
#define min_y 230
#define max_x 774
#define max_y 367
#define mid_y 304

#define Amid_x 173.38
#define Ay0 -45.6
#define Ay1 45.35
#define Ay2 -22.5
#define Ay3 -11.47
#define Ax0 -43
#define Ax1 43
#define Alegw 2364.7
```

```
#define AQw 1000

#define QS 429
#define Qmid_x 336.8
#define Qy0 -42.3
#define Qy1 -61.9
#define Qtailw 588
#define Qxtemp1 2.42
#define Qxtemp2 32.38

#define SI 574
#define S1mx 497.34
#define Sy1 -20.72
#define Sy2 21.78
#define Sx0 -60.34
#define Sx1 57.13
#define S2mx 688.52

#define IS 617
#define Ix0 583.46
#define Ix1 608.34
#define Iy0 258.83
#define Iy1 349.13

// For any point (s,t) in the unit square, return 1 if it
// is inside any part of the text, 0 otherwise.
float aqsistext_mask(float s; float t) {

// These are the polynomials, obtained by implicitization
// of a number of hand-tuned Bezier segments.
uniform float Qc1[10] = { // Outer contour for Q
    -2.515455999999997e+000,   -1.468212479999998e+002,   -2.856536927999995e+003,
    -1.852548213599996e+004,   -3.138538013868060e+006,    4.184088320992688e+006,
    -9.080414450785264e+006,   -1.916338893062538e+008,   -3.555433156037660e+008,
    3.913333377932780e+010
};
uniform float Qc2[10] = { // Inner contour for Q
    3.176522999999993e+000,   -1.303022699999999e+002,    1.781684100000000e+003,
    -8.120601000000011e+003,   -8.709029702512207e+005,    8.446794868238396e+005,
    -1.940215343163121e+006,   -3.388764263479087e+007,   -4.866920986486877e+007,
    4.913481392003620e+009
};

uniform float Sc1a[10] = { // Outer contour for upper left part of S
    -1.560895999999972e+000,   -1.076614559999988e+002,   -2.475285371999988e+003,
    -1.897007496300003e+004,   -4.203757925723708e+005,   -1.008869191772405e+005,
    -7.395343386686108e+005,    1.021959972815947e+007,    4.231012854331020e+007,
    1.619176102444687e+009
};
uniform float Sc1b[10] = { // Inner contour for upper left part of S
    1.281290399999982e+001,    2.375315279999981e+002,    1.467823031999996e+003,
    3.023464536000010e+003,    1.205445041712036e+004,   -6.328299537503940e+004,
    -7.180880134607939e+004,    3.392372672377217e+005,   -1.514004847128142e+006,
    2.064460489533454e+006
};
uniform float Sc2a[10] = { // Outer contour for upper right part of S
    -3.732479999999983e-001,   -5.782233599999982e+001,   -2.985881183999995e+003,
    -5.139586223200000e+004,   -2.631670560009098e+000,    3.699543764663996e+005,
    7.819825232895721e+006,   -1.079471991752684e+007,   -3.985480538763968e+008,
    6.793959228746701e+009
};
uniform float Sc2b[10] = { // Inner contour for upper right part of S
    2.352637000000104e+000,   -5.720622600000166e+001,    4.636715160000063e+002,
    -1.252726551999998e+003,   -1.846763602109960e+003,   -6.592559436540401e+003,
    1.285963272360889e+005,   -3.864184150032301e+005,   -7.438671069995483e+006,
    1.621462796641338e+008
};
uniform float Sc3a[10] = { // Outer contour for middle left part of S
    5.053029696000000e+003,   -4.118395881599998e+004,    1.118878881119999e+005,
    -1.013250455279999e+005,   -3.572216779433399e+005,   -9.000108087994801e+005,
```

```
    5.756484950751777e+006,    1.383060896290501e+007,    2.711575485516367e+008,
    1.126225316211760e+009
};
uniform float Sc3b[10] = { // Inner contour for middle left part of S
    3.842405830000005e+002,   -2.440218393000003e+003,    5.165744301000007e+003,
   -3.645153819000006e+003,    1.739903439690002e+004,   -3.924885612033014e+004,
    4.897977973740215e+003,    1.641019444597991e+006,    1.507638564822112e+007,
   -1.138439458260446e+008
};
uniform float Sc4a[10] = { // Outer contour for middle right part of S
   -6.612913132999999e+003,    4.398978869400002e+004,   -9.754155596400009e+004,
    7.209517952800010e+004,   -3.900824717933405e+005,    1.178804479406984e+005,
    1.696167582980258e+006,   -4.879382691530628e+007,   -5.122662213495559e+008,
    3.782558913476014e+009
};
uniform float Sc4b[10] = { // Inner contour for middle right part of S
   -2.509911278999998e+003,    1.145250908099999e+004,   -1.741893765299999e+004,
    8.831234762999993e+003,    1.364069703129298e+005,   -9.938912235713974e+004,
   -1.704812004398699e+005,   -8.481654673763756e+006,   -6.854711509977910e+007,
   -3.141699222699289e+008
};
uniform float Sc5a[10] = { // Outer contour for lower left part of S
   -1.108956700000006e+001,    1.405341540000006e+002,   -5.936465160000021e+002,
    8.358968880000023e+002,    2.227940722530032e+003,   -3.338964917802015e+004,
    9.913940441721011e+004,   -2.470532821691856e+005,    6.859080825118593e+006,
    1.916457264647791e+008
};
uniform float Sc5b[10] = { // Inner contour for lower left part of S
   -1.000000000000011e+000,    8.271000000000059e+001,   -2.280314700000008e+003,
    2.095609209300000e+004,   -1.640258734248001e+004,    4.888192197095951e+004,
    2.539159491029130e+006,    5.028775045349196e+006,    1.189107392873075e+008,
    1.992721536814904e+009
};
uniform float Sc6a[10] = { // Outer contour for lower right part of S
    1.194389981000002e+003,    5.504775690000010e+003,    8.456912700000017e+003,
    4.330747000000009e+003,    5.079518640874504e+005,    2.042561801719503e+006,
    1.952816523506253e+006,    7.791598981763187e+007,   -6.508484938309293e+007,
   -7.029013111518749e+009
};
uniform float Sc6b[10] = { // Inner contour for lower right part of S
    8.615124999999830e+000,    1.894907249999976e+002,    1.389290534999992e+003,
    3.395290527000006e+003,   -2.161032396471031e+004,    2.697558897572927e+004,
    8.647800264089205e+003,   -9.382476414227656e+005,   -3.404335509997699e+006,
    3.871235795097654e+007
};

#define AASTEP(a,w,x) smoothstep(a-0.5*w,a+0.5*w,x)
#define AACUBIC(Pc,x,y) AASTEP(0, stepwidth, cubic(Pc,x,y)/gradmag(Pc,x,y))

float x_global, y_global; // global coordinates
float x, y; // local coordinates
float mask1, mask2, mask3, mask4, pattern; // temporary variables

x_global = s * 708 + 66;  // Scale unit square to cover our somewhat arbitrary bounding box
y_global = t * 708 + 230; // (The units were originally "points" in a full-page PostScript file.)

// Calculate a filter width in local coordinates for the step antialiasing
// We are shading over (s,t), not (u,v), so we need ds/du etc.
float dxdu = 708*Du(s)*du; // dx/ds = 708 for all the letters
float dxdv = 708*Dv(s)*dv;
float dydu = 708*Du(t)*du; // dy/dt = 708 for all the letters
float dydv = 708*Dv(t)*dv;
// Calculate an approximate stepwidth in (x,y) space.
// (This could be done better in an anisotropic, pattern-specific manner
// by first transforming the specific gradient vector to screen space
// and checking its length there, but this is a lot simpler and works OK.)
float stepwidth = sqrt(dxdu*dxdu + dxdv*dxdv + dydu*dydu + dydv*dydv);

if((x_global<min_x) || (x_global>max_x) || (y_global<min_y) || (y_global>max_y)) // If outside bbox
    pattern = 0.0; // The area outside the bounding box contains no shapes
```

```
else { // We are inside the bounding box, and need to test for intersections with the letters
    y = y_global - mid_y; // transform to local y coordinate system
    float Ax = x_global - Amid_x; // transform to local x coordinate for A
    float A2 = -90.95*Ax - 69.82*y + 4468.2875; // Sloping decision boundary between A and Q
    if(A2 > -AQw) { // We are to the left of the Q, so draw the A
        x = Ax; // local x coordinate for A
        float A1 = 90.95*x - 69.82*y + 4467.8315;
        mask1 = min(AASTEP(Ay0, stepwidth, y), 1-AASTEP(Ay1, stepwidth, y)); // top and bottom
        mask2 = min(AASTEP(Ay2, stepwidth, y), 1-AASTEP(Ay3, stepwidth, y)); // horizontal bar
        float gradmagA = sqrt(90.95*90.95+69.82*69.82); // Constant gradient, same for A1 and A2
        if(x<0) { // left half
            mask3 = min(mask1, min(AASTEP(0, stepwidth, A1/gradmagA),
            1-AASTEP(Alegw/gradmagA, stepwidth, A1/gradmagA))); // left leg
            mask4 = min(mask2, step(Ax0, x)); // left limit of horizontal bar
            pattern = max(mask3, mask4);
        }
        else { // right half
            mask3 = min(mask1, min(AASTEP(0, stepwidth, A2/gradmagA),
            1-AASTEP(Alegw/gradmagA, stepwidth, A2/gradmagA))); // right leg
            mask4 = min(mask2, 1-step(Ax1, x)); // right limit of horizontal bar
            pattern = max(mask3, mask4);
        }
    }

    else if(x_global<QS) { // We are to the left of the first S, so draw Q
        x = x_global - Qmid_x; // transform to local x coordinate for Q
        if(x<0) { // left half
            if(y<0) { // bottom left quarter
                mask1 = min(1-AACUBIC(Qc2,-x,-y), AACUBIC(Qc1,-x,-y));
            }
            else { // top left quarter
                mask1 = min(1-AACUBIC(Qc2,-x,y), AACUBIC(Qc1,-x,y));
            }
        }
        else { // right half
            if(y<0) { // bottom right quarter
                mask1 = min(1-AACUBIC(Qc2,x,-y), AACUBIC(Qc1,x,-y));
            }
            else { // top right quarter
                mask1 = min(1-AACUBIC(Qc2,x,y), AACUBIC(Qc1,x,y));
            }
        }
        float Q1 = 19.6*x + 35.02*y + 1433.914;
        float gradQ1 = sqrt(19.6*19.6+35.02*35.02);
        mask2 = min(min(step(Qy1,y), 1-AASTEP(Qy0,stepwidth,y)),
        min(AASTEP(0,stepwidth,Q1/gradQ1), 1-AASTEP(Qtailw/gradQ1,stepwidth,Q1/gradQ1)));
        pattern = max(mask1, mask2); // Add the little "tail" for Q
    }

    else if(x_global<SI) { // We are to the left of the I, so draw the first S
        x = x_global - S1mx; // transform to local x coordinate for first S
        if(x<0) { // left half
            if(y<Sy1) { // bottom left part
                mask1 = min(AACUBIC(Sc5a,x,y), 1-AACUBIC(Sc5b,x,y));
                pattern = min(mask1, AASTEP(Sx0,stepwidth,x));
            }
            else if(y<Sy2) { // middle left part
                pattern = min(AACUBIC(Sc3a,x,y), 1-AACUBIC(Sc3b,x,y));
            }
            else { // upper left part
                pattern = min(AACUBIC(Sc1a,x,y), AACUBIC(Sc1b,x,y));
            }
        }
        else { // right half
            if(y<Sy1) { // bottom right part
                pattern = min(1-AACUBIC(Sc6a,x,y), 1-AACUBIC(Sc6b,x,y));
            }
            else if(y<Sy2) { // middle right part
                pattern = min(AACUBIC(Sc4a,x,y), 1-AACUBIC(Sc4b,x,y));
            }
```

```
            else { // upper right part
                mask1 = min(AACUBIC(Sc2a,x,y), 1-AACUBIC(Sc2b,x,y));
                pattern = min(mask1, 1-AASTEP(Sx1,stepwidth,x));
            }
        }
    }

    else if(x_global<IS) { // We are to the left of the second S, do draw the I (a simple box)
        pattern = min(min(AASTEP(Ix0,stepwidth,x_global), 1-AASTEP(Ix1,stepwidth,x_global)),
            min(AASTEP(Iy0,stepwidth,y_global), 1-AASTEP(Iy1,stepwidth,y_global)));
    }

    else { // We are far out to the right, so draw the second S
        x = x_global - S2mx; // transform to local x coordinate for second S
        // From here on, the code is identical to the first S.
        if(x<0) { // left half
            if(y<Sy1) { // bottom left part
                mask1 = min(AACUBIC(Sc5a,x,y), 1-AACUBIC(Sc5b,x,y));
                pattern = min(mask1, AASTEP(Sx0,stepwidth,x));
            }
            else if(y<Sy2) { // middle left part
                pattern = min(AACUBIC(Sc3a,x,y), 1-AACUBIC(Sc3b,x,y));
            }
            else { // upper left part
                pattern = min(AACUBIC(Sc1a,x,y), AACUBIC(Sc1b,x,y));
            }
        }
        else { // right half
            if(y<Sy1) { // bottom right part
                pattern = min(1-AACUBIC(Sc6a,x,y), 1-AACUBIC(Sc6b,x,y));
            }
            else if(y<Sy2) { // middle right part
                pattern = min(AACUBIC(Sc4a,x,y), 1-AACUBIC(Sc4b,x,y));
            }
            else { // upper right part
                mask1 = min(AACUBIC(Sc2a,x,y), 1-AACUBIC(Sc2b,x,y));
                pattern = min(mask1, 1-AASTEP(Sx1,stepwidth,x));
            }
        }
    }
 }
 return pattern;
}

// filterstep() works fine with Aqsis and PRMan, but fails with Pixie (a Pixie bug)
#define STEP(a,x) filterstep(a,x)

surface aqsislogo() {

    // Define some colors we will use for this surface.
    // These are the correct, official colors for the Aqsis logo.
    // Only "bgcolor" may be changed, it can be set to black, white
    // or any gray tone far enough from (0.6, 0.6, 0.6).
    color bgcolor  = color(1,1,1); //color(0.282, 0.282, 0.282);
    color ringcolor = color(0.6, 0.6, 0.6);
    float ringalpha = 0.6;
    color ballcolor  = color(0.2, 0.4, 0.8);
    float ballalpha = 0.66;
    // The text should be black on a light background, white on dark.
    color textcolor = color(0.0, 0.0, 0.0); // color(1.0,1.0,1.0);

    // The logo part of the shader draws a pattern of size 370x370 units.
    // Rescale the texture coordinates to match that.
    float x = s * 370;
    float y = t * 370;

    // Calculate normalised distances to the ring contours
    float outerdist = (x-185)*(x-185)/164/164 + (y-185)*(y-185)/89/89;
    float innerdist = (x-185)*(x-185)/124/124 + (y-200)*(y-200)/63/63;
    float gapdist   = (x-283)*(x-283)/48/48 + (y-136)*(y-136)/48/48;
```

```
    outerdist = (sqrt(outerdist) - 1) * 164/370;
    innerdist = (sqrt(innerdist) - 1) * 124/370;
    gapdist   = (sqrt(gapdist) - 1) * 48/370;

    float ring = 1-STEP(0, outerdist); // Big ring outline
    ring *= STEP(0, innerdist);        // Hole in big ring
    ring *= STEP(0, gapdist);          // Gap in big ring

    // Calculate normalised distances to the three circles
    float bigballdist = (x-283)*(x-283)/36/36 + (y-136)*(y-136)/36/36;
    float mediumballdist = (x-235)*(x-235)/18/18 + (y-180)*(y-180)/18/18;
    float smallballdist = (x-202)*(x-202)/9/9 + (y-200)*(y-200)/9/9;
    bigballdist = (sqrt(bigballdist) - 1) * 36/370;
    mediumballdist = (sqrt(mediumballdist) - 1) * 18/370;
    smallballdist = (sqrt(smallballdist) - 1) * 9/370;

    float balls = (1-STEP(0, bigballdist)); // Big ball
    balls += (1-STEP(0, mediumballdist));   // Medium ball
    balls += (1-STEP(0, smallballdist));    // Small ball

    color ringmixcolor = bgcolor * (1-ringalpha) + ringcolor * ringalpha;
    color ballmixcolor = bgcolor * (1-ballalpha) + ballcolor * ballalpha;
    Ci = mix(mix(mix(bgcolor, ringmixcolor, ring), ballmixcolor, balls),
        textcolor, aqsistext_mask(s*1.5-0.25, t*1.5-0.14));

    Oi = Os;
}
```

A test RIB file

```
Display "aqsislogo.tif" "framebuffer" "rgb"
Format 800 800 1
# "PixelSamples 4 4" is needed in Aqsis to hide interpolation artefacts
PixelSamples 1 1
Projection "perspective" "fov" 30
ShadingRate 1

# Set this option for Pixie
#Hider "raytrace" "jitter" 0

# Uncomment the lines below to get acceptable results with PRMan
Hider "hidden" "jitter" 0 # Required for PRMan to avoid a noisy look
ShadingInterpolation "smooth" # Makes things a lot better in PRMan
Attribute "derivatives" "centered" [0] # Gives less artefacts in PRMan

Translate -0.5 -0.5 1.9

WorldBegin

   Surface "aqsislogo"

   AttributeBegin
      Polygon "P" [0 0 0 0 1 0 1 1 0 1 0 0]
      "st" [0 0 0 1 1 1 1 0]
   AttributeEnd

WorldEnd
```

Appendix B: the implicit form of a general Béziér segment

```
% Polynomial coefficients for a cubic Bezier curve, from
% x = x0*B0(t)+x1*B1(t)+x2*B2(t)+x3*B3(t),
% y = y0*B0(t)+y1*B1(t)+y2*B2(t)+y3*B3(t),
% where (x0,y0), (x1,y1), (x2,y2) and (x3,y3) are the control points
% and B0(t)=(1-t)^3, B1(t)=3*t*(1-t)^2, B2(t)=3*t^2*(1-t), B3(t)=t^3
% to x = A(t), y = B(t) where
% A(t) = a0+a1*t+a2*t^2+a3*t^3, B(t) = b0+b1*t+b2*t^2+b3*t^3

a0 = x0;
a1 = -3*x0 + 3*x1;
a2 = 3*x0 - 6*x1 + 3*x2;
a3 = -x0 + 3*x1 - 3*x2 + x3;
b0 = y0;
b1 = -3*y0 + 3*y1;
b2 = 3*y0 - 6*y1 + 3*y2;
b3 = -y0 + 3*y1 - 3*y2 + y3;

% Implicitization of a general parametric cubic curve
% from x=A(t), y=B(t) to C(x,y)=0
% where A(t) = a0+a1*t+a2*t^2+a3*t^3, B(t)=b0+b1*t+b2*t^2+b3*t^3 and
% C(x,y)=C1+Cx*x+Cy*y+Cx2*x^2+Cxy*x*y+Cy2*y^2+Cx3*x^3+Cx2y*x^2*y+Cxy2*x*y^2+Cy3*y^3

C1 =
a0*b3*a1*b2*a3*b0-3*a0^2*b3*b2*a3*b1-a0^3*b3^3+3*a3*b0*a0^2*b3^2+2*a0^2*b3^2*a2*b1
+a3^3*b0^3-a3*b0*a2*b1*a0*b3+3*a3^2*b0*a0*b2*b1+3*a3*b0^2*a2*a1*b3+a1*b3^2*a0^2*b2
-2*a3*b1*a1^2*b3*b0+2*a3*b1^2*a1*b3*a0-a3^2*b1^3*a0+a1^3*b3^2*b0+a3^2*b1^2*a1*b0
-3*a0*b3^2*a2*b0*a1+a3*b2*a2^2*b0^2+a3*b2^2*a1^2*b0-a3^2*b1*a2*b0^2
-a1^2*b3^2*b1*a0+a3*b2^3*a0^2-2*a3*b2^2*a2*b0*a0-3*a3^2*b0^2*a0*b3
-2*a3^2*b0^2*a1*b2-a2*b3*a1^2*b2*b0+a2^2*b3*b1*a1*b0+a2*b3*a1*b2*b1*a0
+2*a2^2*b3*b0*a0*b2-a2*b3*a0^2*b2^2-a2^2*b3*b1^2*a0+a3*b2*a2*b1^2*a0
-a3*b2^2*a1*b1*a0-a3*b2*a2*b1*a1*b0-a2^3*b3*b0^2;

Cx =
-2*a1*b3^2*b2*a0-3*a3^2*b0*b2*b1-b3*a1*b2*a3*b0+3*b3^3*a0^2-2*a3*b1^2*a1*b3
+6*b3*a0*b2*a3*b1+a1^2*b3^2*b1+a3^2*b1^3+3*a3^2*b0^2*b3+2*a3*b2^2*a2*b0
-a3*b2*a2*b1^2+a2^2*b3*b1^2+a3*b2^2*a1*b1-2*a2^2*b3*b0*b2-4*b3^2*a2*b1*a0
-a2*b3*a1*b2*b1+2*a2*b3*b2^2*a0-2*a3*b2^3*a0-6*a3*b0*b3^2*a0+a3*b0*a2*b1*b3
+3*b3^2*a2*b0*a1;

Cy =
6*a3^2*b0*a0*b3-3*a3^2*a0*b2*b1+2*a3*b2^2*a2*a0-3*a3^3*b0^2-2*a3*b2*a2^2*b0
+2*a3*b1*a1^2*b3+a3*a2*b1*a0*b3+3*a0*b3^2*a2*a1+2*a2^3*b3*b0-a3^2*b1^2*a1
-a0*b3*a1*b2*a3-3*a3*a0^2*b3^2-a2^2*b3*b1*a1+a2*b3*a1^2*b2-2*a2^2*b3*a0*b2
-6*a3*a2*b0*a1*b3+4*a3^2*a1*b2*b0-a1^3*b3^2-a3*b2^2*a1^2+2*a3^2*b1*a2*b0
+a3*b2*a2*b1*a1;

Cx2 =
-3*b3^3*a0-a2*b3*b2^2+2*b3^2*a2*b1+3*a3*b0*b3^2-3*b3*b2*a3*b1+a3*b2^3+a1*b3^2*b2;

Cxy =
-2*a3*b2^2*a2+2*a2^2*b3*b2-6*a3^2*b0*b3+6*a3*b3^2*a0+3*a3^2*b2*b1-a3*a2*b1*b3
+b3*a1*b2*a3-3*b3^2*a2*a1;

Cy2 =
-a3^2*b1*a2+3*a3*a2*a1*b3+a3*b2*a2^2+3*a3^3*b0-2*a3^2*a1*b2-3*a3^2*a0*b3-a2^3*b3;

Cx3 = b3^3;

Cx2y = -3*a3*b3^2;

Cxy2 = 3*a3^2*b3;

Cy3 = -a3^3;
```