Antialiased 2D Grid, Marker, and Arrow Shaders

Nicolas P. Rougier

Antialiased 2D Grid, Marker, and Arrow Shaders

Nicolas P. Rougier
INRIA, CNRS, Université de Bordeaux

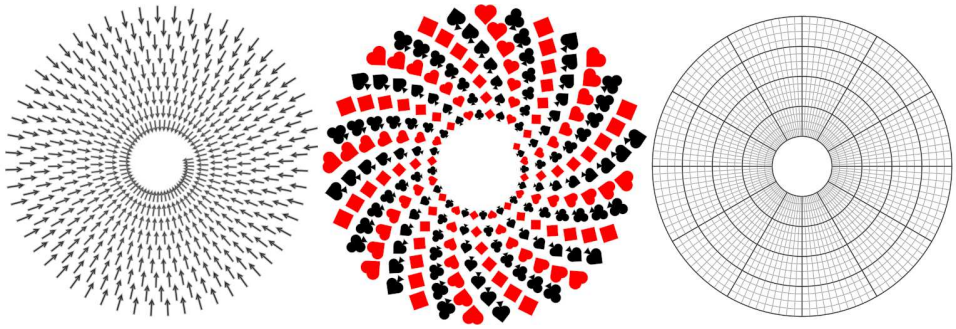

Figure 1. Antialiased arrows, markers, and grid as drawn by the GPU.

Abstract

Grids, markers, and arrows are important components in scientific visualisation. Grids are widely used in scientific plots and help visually locate data. Markers visualize individual points and aggregated data. Quiver plots show vector fields, such as a velocity buffer, through regularly-placed arrows. Being able to draw these components quickly is critical if one wants to offer interactive visualisation. This article provides algorithms with GLSL implementations for drawing grids, markers, and arrows using implicit surfaces that make it possible quickly render pixel-perfect antialiased shapes.

1. Introduction

Grids, markers and arrows are important components in scientific visualization. Grids are widely used in scientific plots and help visually locate data. In the most simple case (orthographic mode, cartesian axis, no rotation), such grids can easily be drawn using a simple set of straight lines, without the need of anti-aliasing techniques. Complexity arises when one want to use other projection systems like polar projection or those found in cartography (e.g., Hammer, Mercator, etc.). In such a case, rendering a grid might require highly tessellated lines to cope with curvature as well as precise anti-aliasing techniques [Chan and Durand 2005; Rougier 2013]. Furthermore, any

 ISSN 2331-7418

change in the projection (zoom, translate, scale) requires a new tessellation stage. In order to achieve resolution independence (e.g., for curves [Loop and Blinn 2005]), I describe an alternative and versatile approach where an anti-aliased grid is drawn directly by the GPU, provided the forward and inverse projection are known (either analytically or approximated). Using the same implicit surface technique, we can advance by drawing different markers (e.g., cross, circle, square, etc) as well as arrows and generate beautiful and efficient scatter or quiver plots.

In this paper, I introduce the different antialiasing techniques that are relatively easy to understand before introducing an atlas of markers and arrows built from them. Then, using the same implicit surface approach, I will explain how this method can be extended to drawing a grid using any kind of projection as long as the topological relationships are preserved.

As most of this paper is structured as a cookbook or reference manual for 2D visualization elements, I provide a hyperlinked table of contents for quickly jumping to the section of interest when reading the PDF file on a computer instead of a printout.

Contents

2. Antialiasing using implicit surfaces

Chan and Durand [2005] introduced an antialiasing technique for lines (based on work originally developed by McNamara [2000]) where the fragment distance to the mathematical line is used to compute fragment coverage. This idea can be further extended using a signed distance that distinguishes between exterior and interior. Then, as it has been proposed by Green [2007], we can easily fill, outline, and stroke shapes as illustrated on figure 2. In the code listings that follow, all units are in pixels.

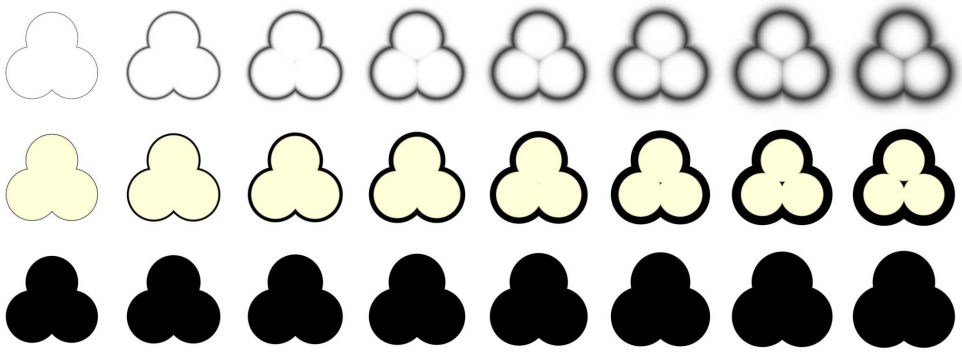

Figure 2. Antialias functions. From bottom to top: filled, outlined, stroked. From left to right: increasing linewidth. Top row from left to right: increading antialiased area.

2.1 Stroked shape

```
vec4 stroke(float distance,  // Signed distance to line
            float linewidth, // Stroke line width
            float antialias, // Stroke antialiased area
            vec4 stroke)     // Stroke color
{
    float t = linewidth / 2.0 - antialias;
    float signed_distance = distance;
    float border_distance = abs(signed_distance) - t;
    float alpha = border_distance / antialias;
    alpha = exp(-alpha * alpha);

    if( border_distance < 0.0 )
        return stroke;
    else
        return vec4(stroke.rgb, stroke.a * alpha);
}
```

Listing 1. Antialiased stroked shape.

2.2 Filled shape

```
vec4 filled(float distance,  // Signed distance to line
            float linewidth, // Stroke line width
            float antialias, // Stroke antialiased area
            vec4 fill)       // Fill color
{
    float t = linewidth / 2.0 - antialias;
    float signed_distance = distance;
    float border_distance = abs(signed_distance) - t;
    float alpha = border_distance / antialias;
    alpha = exp(-alpha * alpha);

    if( border_distance < 0.0 )
        return fill;
    else if( signed_distance < 0.0 )
        return fill;
    else
        return vec4(fill.rgb, alpha * fill.a);
}
```

Listing 2. Antialiased filled shape.

2.3 Outlined shape

```
vec4 outline(float distance, // Signed distance to line
             float linewidth, // Stroke line width
             float antialias, // Stroke antialiased area
             vec4 stroke,     // Stroke color
             vec4 fill)       // Fill color
{
    float t = linewidth / 2.0 - antialias;
    float signed_distance = distance;
    float border_distance = abs(signed_distance) - t;
    float alpha = border_distance / antialias;
    alpha = exp(-alpha * alpha);

    if( border_distance < 0.0 )
        return stroke;
    else if( signed_distance < 0.0 )
        return mix(fill, stroke, sqrt(alpha));
    else
        return vec4(stroke.rgb, stroke.a * alpha);
}
```

Listing 3. Antialiased outlined shape.

3. Markers

An elementary shape is defined by an implicit function giving the signed distance to the shape frontiers. We use an arbitrary convention such that negative values are inside the shape and positive values are outside the shape. Considering two implicit surfaces S_1 and S_2, we can define the union, difference and intersection operators as follow:

- $\text{Union}(S_1,S_2) : \forall x,y, U(x,y) = \min(S1(x,y), S2(x,y))$
- $\text{Difference}(S_1,S_2) : \forall x,y, D(x,y) = \max(S1(x,y), -S2(x,y))$
- $\text{Intersection}(S_1,S_2) : \forall x,y, I(x,y) = \max(S1(x,y), S2(x,y))$

This provides a simple two-dimensional constructive geometry, a simple application of the larger ideas presented by Schmidt [2011]. All of the subsequent markers are combinations of half-planes and circles, except for the ellipse, which is is a special case. These use the base vertex and fragment shaders so that only the `marker` needs to be replaced with the marker actual code. I described some common markers; any traditional constructive solid geometry operations can be applied as well.

Finally, I note that markers may be rotated. We must to take this into account when computing the size of the `Point`.

3.1. Generic vertex and fragment marker shaders

Here are the vertex and fragment shaders that are used to display markers using points.

```
#version 120

const float SQRT_2 = 1.4142135623730951;

uniform mat4 ortho;
uniform float size, orientation, linewidth, antialias;
attribute vec3 position;
varying vec2 rotation;
varying vec2 v_size;
void main (void)
{
    rotation = vec2(cos(orientation), sin(orientation));
    gl_Position = ortho * vec4(position, 1.0);
    v_size = M_SQRT_2 * size + 2.0*(linewidth + 1.5*antialias);
    gl_PointSize = v_size;
}
```

Listing 4. Marker vertex

```
#version 120

const float PI = 3.14159265358979323846264;
const float SQRT_2 = 1.4142135623730951;

uniform float size, linewidth, antialias;
uniform vec4 fg_color, bg_color;
varying vec2 rotation;
varying vec2 v_size;
void main()
{
    vec2 P = gl_PointCoord.xy - vec2(0.5,0.5);
    P = vec2(rotation.x*P.x - rotation.y*P.y,
             rotation.y*P.x + rotation.x*P.y);
    float distance = marker(P*v_size, size);
    gl_FragColor = outline(distance,
                           linewidth, antialias, fg_color, bg_color);
}
```

Listing 5. Marker fragment.

3.2. Disc marker

A simple disc.

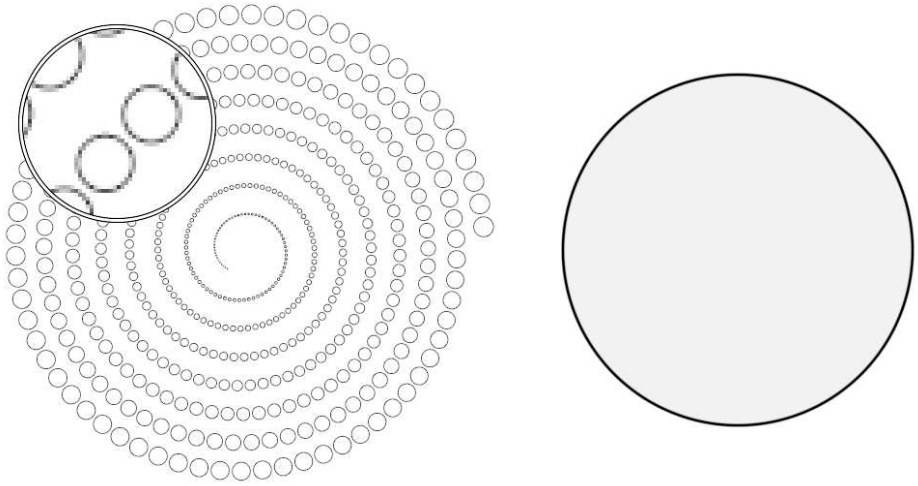

Figure 3. Disc marker.

```
float disc(vec2 P, float size)
{
    return length(P) - size/2;
}
```

Listing 6. Disc marker.

3.3. Square marker

A square is the intersection of four half-planes but we can use the symmetry of the object (`abs`) to shorten the code.

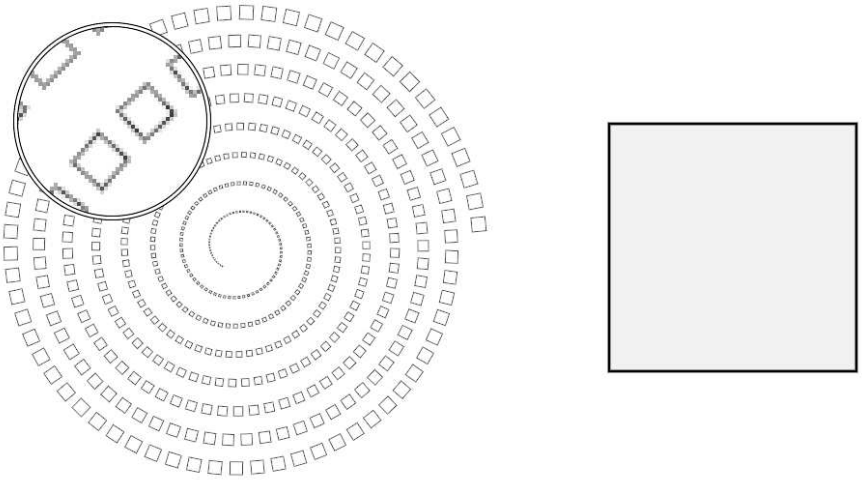

Figure 4. Square marker.

```
float square(vec2 P, float size)
{
    return max(abs(P.x), abs(P.y)) - size/(2.0*M_SQRT_2);
}
```

Listing 7. Square marker.

3.4. Triangle marker

A triangle is the intersection of three half-planes.

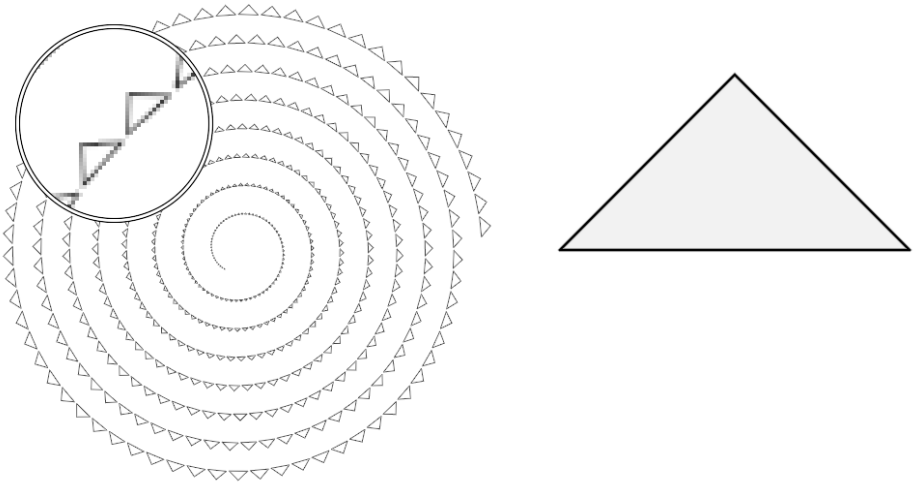

Figure 5. Triangle marker.

```
float triangle(vec2 P, float size)
{
   float x = M_SQRT_2/2.0 * (P.x - P.y);
   float y = M_SQRT_2/2.0 * (P.x + P.y);

   float r1 = max(abs(x), abs(y)) - size/(2*M_SQRT_2);
   float r2 = P.y;
   return max(r1,r2);
}
```

Listing 8. Triangle marker.

3.5. Diamond marker

A diamond is the rotation of a square.

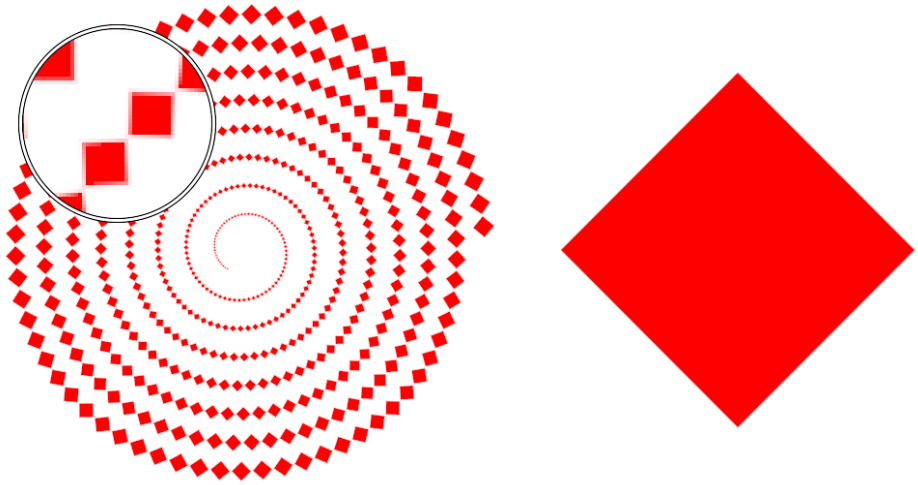

Figure 6. Diamond marker.

```
float diamond(vec2 P, float size)
{
   float x = M_SQRT_2/2.0 * (P.x - P.y);
   float y = M_SQRT_2/2.0 * (P.x + P.y);

   return max(abs(x), abs(y)) - size/(2.0*M_SQRT_2);
}
```

Listing 9. Diamond marker.

3.6. Heart marker

A heart is the union of a diamond and two discs.

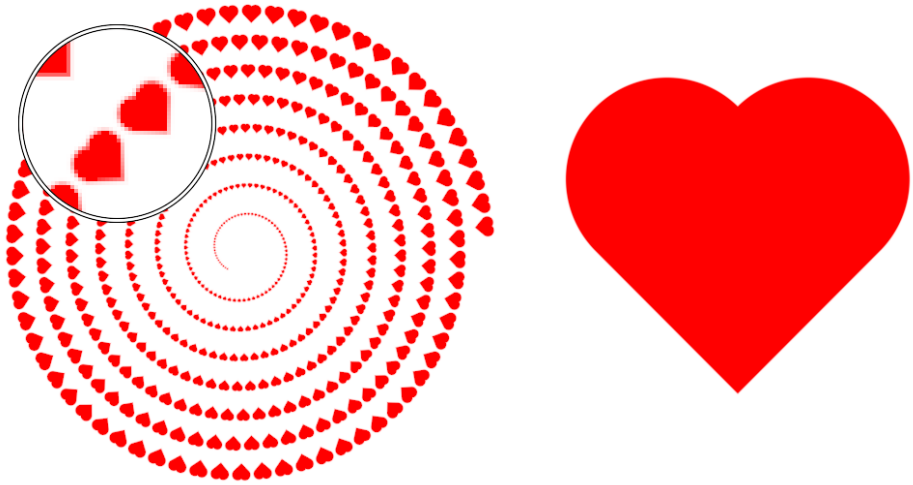

Figure 7. Heart marker.

```
float heart(vec2 P, float size)
{
   float x = M_SQRT_2/2.0 * (P.x - P.y);
   float y = M_SQRT_2/2.0 * (P.x + P.y);
   float r1 = max(abs(x),abs(y))-size/3.5;
   float r2 = length(P - M_SQRT_2/2.0*vec2(+1.0,-1.0)*size/3.5)
              - size/3.5;
   float r3 = length(P - M_SQRT_2/2.0*vec2(-1.0,-1.0)*size/3.5)
              - size/3.5;

   return min(min(r1,r2),r3);
}
```

Listing 10. Heart marker.

3.7. Spade marker

A spade is an inverted heart and a tail is made of two discs and two half-planes.

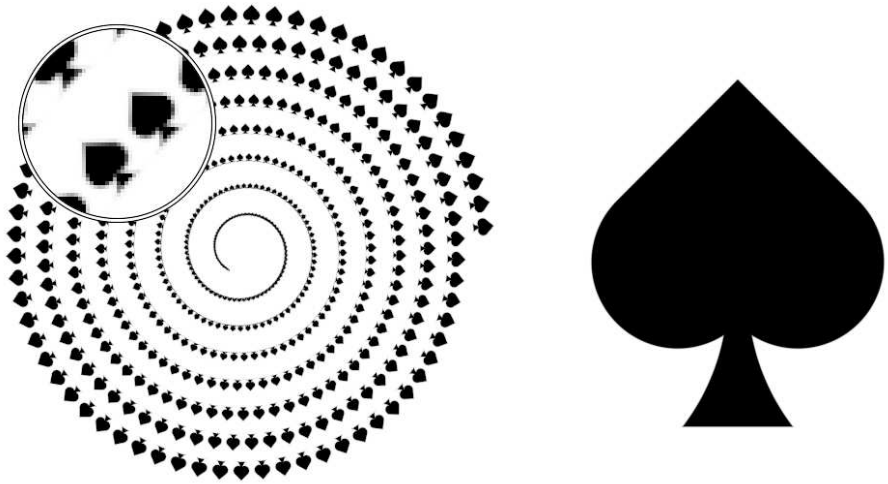

Figure 8. Spade marker.

```
float spade(vec2 P, float size)
{
   // Reversed heart (diamond + 2 circles)
   float s= size * 0.85 / 3.5;
   float x = M_SQRT_2/2.0 * (P.x + P.y) + 0.4*s;
   float y = M_SQRT_2/2.0 * (P.x - P.y) - 0.4*s;
   float r1 = max(abs(x),abs(y)) - s;
   float r2 = length(P - M_SQRT_2/2.0*vec2(+1.0,+0.2)*s) - s;
   float r3 = length(P - M_SQRT_2/2.0*vec2(-1.0,+0.2)*s) - s;
   float r4 =  min(min(r1,r2),r3);

   // Root (2 circles and 2 half-planes)
   const vec2 c1 = vec2(+0.65, 0.125);
   const vec2 c2 = vec2(-0.65, 0.125);
   float r5 = length(P-c1*size) - size/1.6;
   float r6 = length(P-c2*size) - size/1.6;
   float r7 = P.y - 0.5*size;
   float r8 = 0.1*size - P.y;
   float r9 = max(-min(r5,r6), max(r7,r8));

    return min(r4,r9);
}
```

Listing 11. Spade marker.

3.8. Club marker

A club is a clover and a tail.

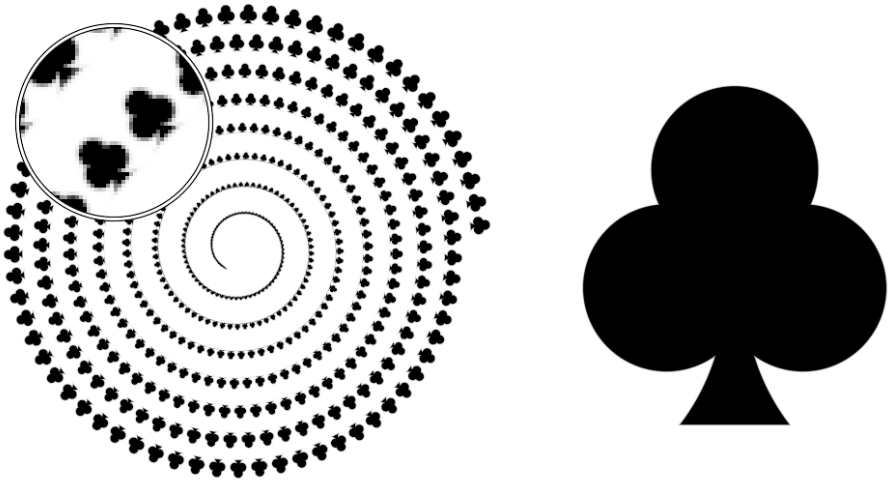

Figure 9. Club marker.

```
float club(vec2 P, float size)
{
    // clover (3 discs)
    const float t1 = -M_PI/2.0;
    const vec2  c1 = 0.225*vec2(cos(t1),sin(t1));
    const float t2 = t1+2*M_PI/3.0;
    const vec2  c2 = 0.225*vec2(cos(t2),sin(t2));
    const float t3 = t2+2*M_PI/3.0;
    const vec2  c3 = 0.225*vec2(cos(t3),sin(t3));
    float r1 = length( P - c1*size) - size/4.25;
    float r2 = length( P - c2*size) - size/4.25;
    float r3 = length( P - c3*size) - size/4.25;
    float r4 =  min(min(r1,r2),r3);

    // Root (2 circles and 2 half-planes)
    const vec2 c4 = vec2(+0.65, 0.125);
    const vec2 c5 = vec2(-0.65, 0.125);
    float r5 = length(P-c4*size) - size/1.6;
    float r6 = length(P-c5*size) - size/1.6;
    float r7 = P.y - 0.5*size;
    float r8 = 0.2*size - P.y;
    float r9 = max(-min(r5,r6), max(r7,r8));

    return min(r4,r9);
}
```

Listing 12. Club marker.

3.9. Chevron marker

A chevron is the difference of two diamonds.

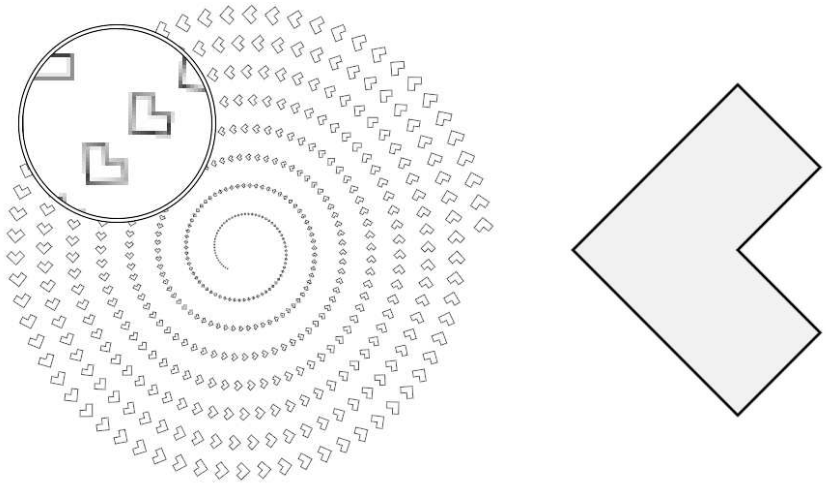

Figure 10. Chevron marker.

```
float chevron(vec2 P, float size)
{
    float x = 1.0/M_SQRT_2 * (P.x - P.y);
    float y = 1.0/M_SQRT_2 * (P.x + P.y);
    float r1 = max(abs(x), abs(y)) - size/3.0;
    float r2 = max(abs(x-size/3.0), abs(y-size/3.0)) - size/3.0;
    return max(r1,-r2);
}
```

Listing 13. Chevron marker.

3.10. Clover marker

A clover is the union of three discs.

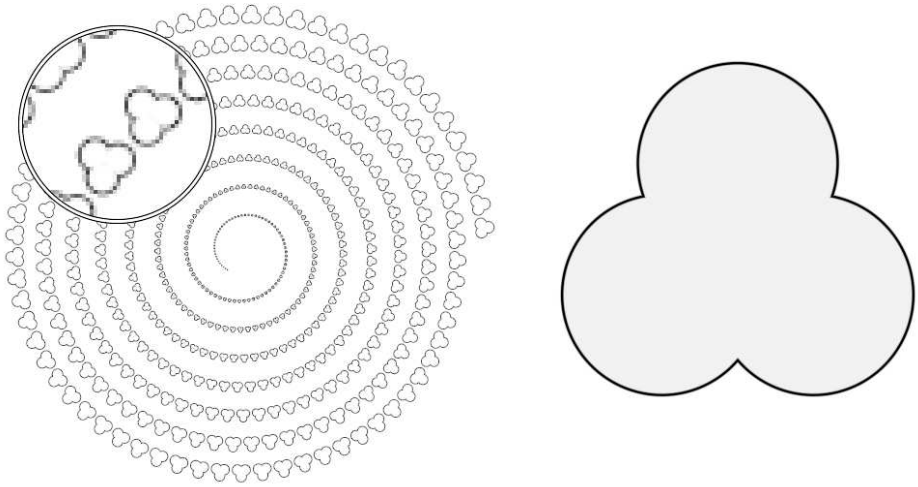

Figure 11. Clover marker.

```
float clover(vec2 P, float size)
{
    const float PI = 3.14159265358979323846264;
    const float t1 = -PI/2;
    const vec2  c1 = 0.25*vec2(cos(t1),sin(t1));
    const float t2 = t1+2*PI/3;
    const vec2  c2 = 0.25*vec2(cos(t2),sin(t2));
    const float t3 = t2+2*PI/3;
    const vec2  c3 = 0.25*vec2(cos(t3),sin(t3));

    float r1 = length( P - c1*size) - size/3.5;
    float r2 = length( P - c2*size) - size/3.5;
    float r3 = length( P - c3*size) - size/3.5;
    return min(min(r1,r2),r3);
}
```

Listing 14. Clover marker.

3.11. Ring marker

A ring is the difference of two discs.

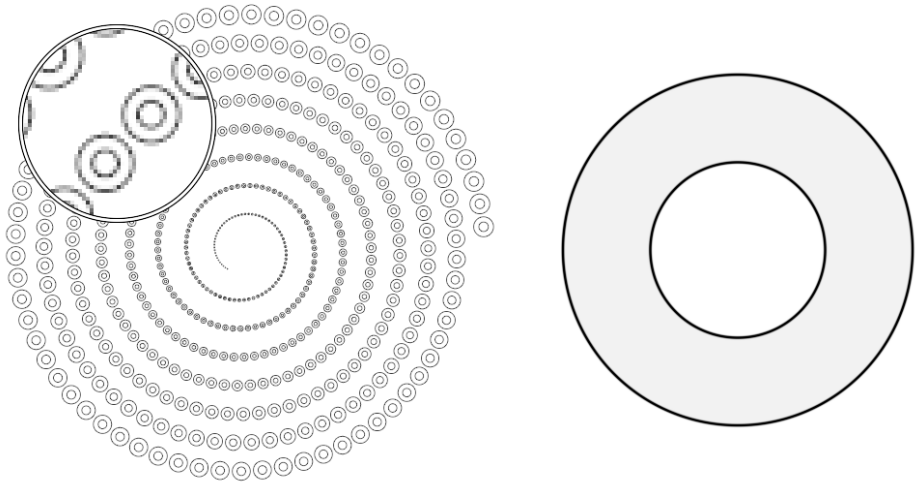

Figure 12. Ring marker.

```
float ring(vec2 P, float size)
{
    float r1 = length(P) - size/2.0;
    float r2 = length(P) - size/4.0;
    return max(r1,-r2);
}
```

Listing 15. Ring marker.

3.12. Tag marker

A tag is the interesection of five half-planes.

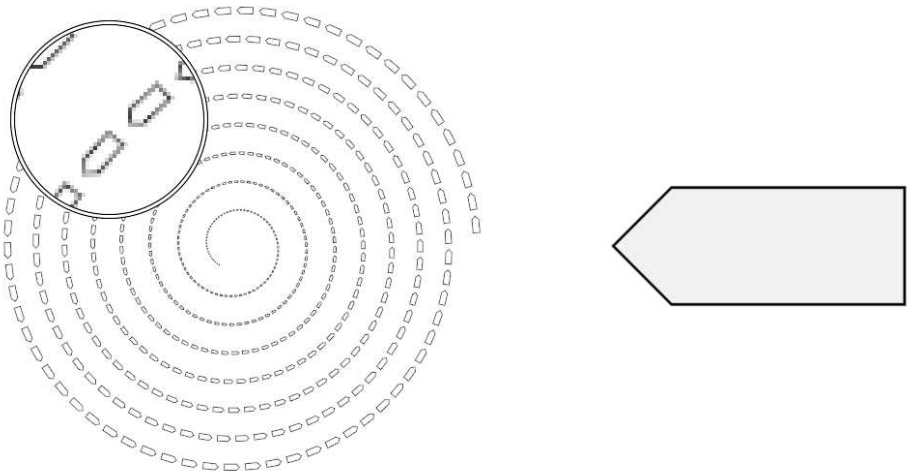

Figure 13. Tag marker.

```
float tag(vec2 P, float size)
{
    float r1 = max(abs(P.x)- size/2.0, abs(P.y)- size/6.0);
    float r2 = abs(P.x-size/1.5)+abs(P.y)-size;
    return max(r1,0.75*r2);
}
```

Listing 16. Tag marker.

3.13. Cross marker

A cross is the intersection of eight half-planes that can be reduced to four using symmetries.

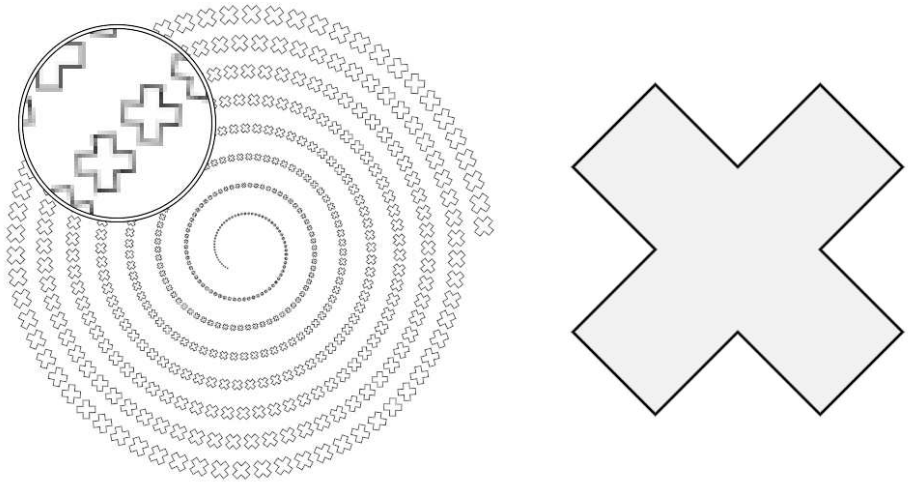

Figure 14. Cross marker.

```
float cross(vec2 P, float size)
{
    float x = M_SQRT_2/2.0 * (P.x - P.y);
    float y = M_SQRT_2/2.0 * (P.x + P.y);
    float r1 = max(abs(x - size/3.0), abs(x + size/3.0));
    float r2 = max(abs(y - size/3.0), abs(y + size/3.0));
    float r3 = max(abs(x), abs(y));
    return max(min(r1,r2),r3) - size/2.0;
}
```

Listing 17. Cross marker.

3.14. Asterisk marker

An asterisk is the union of two crosses.

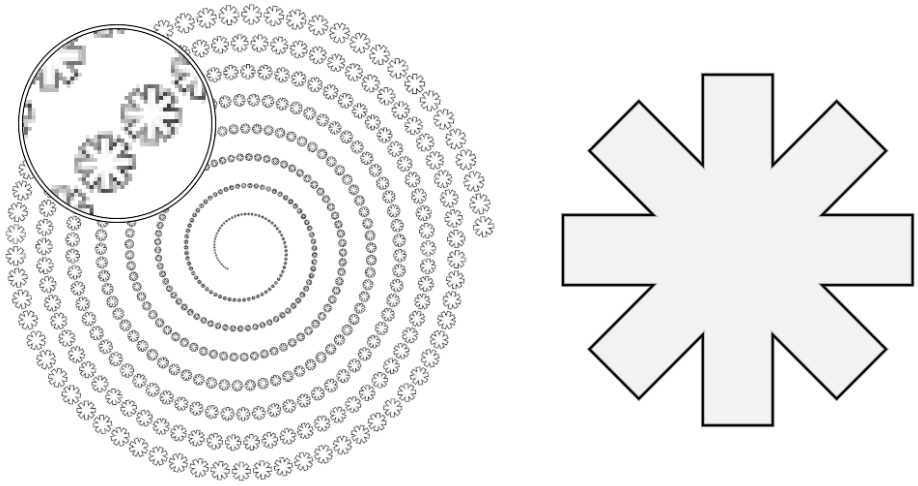

Figure 15. Asterisk marker.

```
float asterisk(vec2 P, float size)
{
   float x = M_SQRT_2/2.0 * (P.x - P.y);
   float y = M_SQRT_2/2.0 * (P.x + P.y);

   float r1 = max(abs(x)- size/2.0, abs(y)- size/10.0);
   float r2 = max(abs(y)- size/2.0, abs(x)- size/10.0);
   float r3 = max(abs(P.x)- size/2.0, abs(P.y)- size/10.0);
   float r4 = max(abs(P.y)- size/2.0, abs(P.x)- size/10.0);
   return min( min(r1,r2), min(r3,r4));
}
```

Listing 18. Asterisk marker.

3.15. Infinity marker

Infinity is the union of two rings.

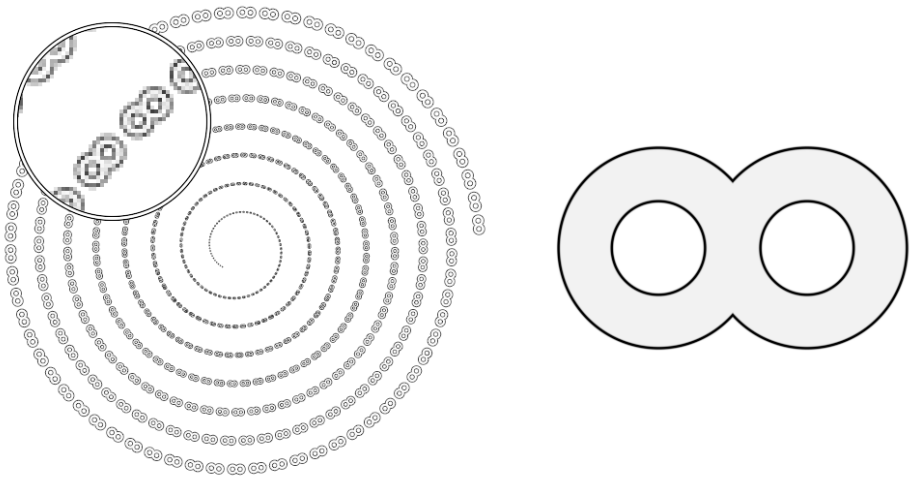

Figure 16. Infinity marker.

```
float infinity(vec2 P, float size)
{
    const vec2 c1 = vec2(+0.2125, 0.00);
    const vec2 c2 = vec2(-0.2125, 0.00);
    float r1 = length(P-c1*size) - size/3.5;
    float r2 = length(P-c1*size) - size/7.5;
    float r3 = length(P-c2*size) - size/3.5;
    float r4 = length(P-c2*size) - size/7.5;
    return min( max(r1,-r2), max(r3,-r4));
}
```

Listing 19. Infinity marker.

3.16. Pin marker

A pin is the difference between the union of three discs and a disc.

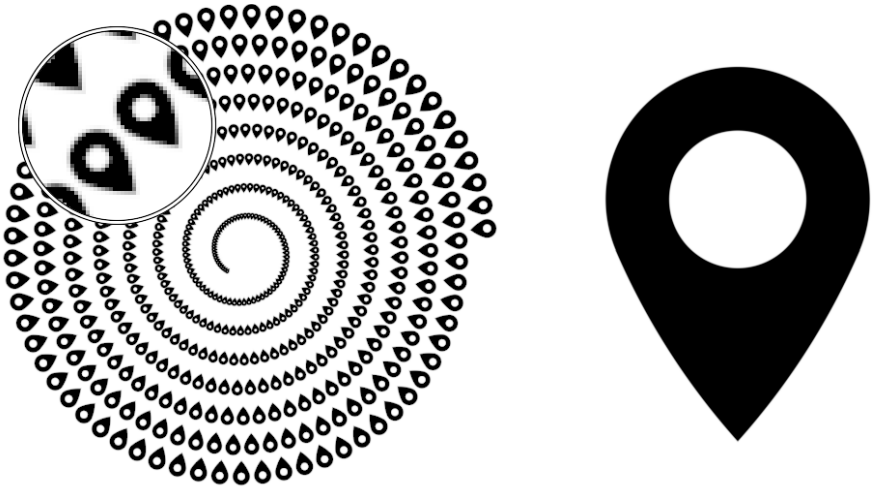

Figure 17. Pin marker.

```
float pin(vec2 P, float size) {
    vec2 c1 = vec2(0.0,-0.15)*size;
    float r1 = length(P-c1)-size/2.675;
    vec2 c2 = vec2(+1.49,-0.80)*size;
    float r2 = length(P-c2) - 2.*size;
    vec2 c3 = vec2(-1.49,-0.80)*size;
    float r3 = length(P-c3) - 2.*size;
    float r4 = length(P-c1)-size/5;
    return max( min(r1,max(max(r2,r3),-P.y)), -r4);
}
```

Listing 20. Pin marker.

3.17. Block arrow marker

Block arrow is the intersection of several half-planes using symmetries.

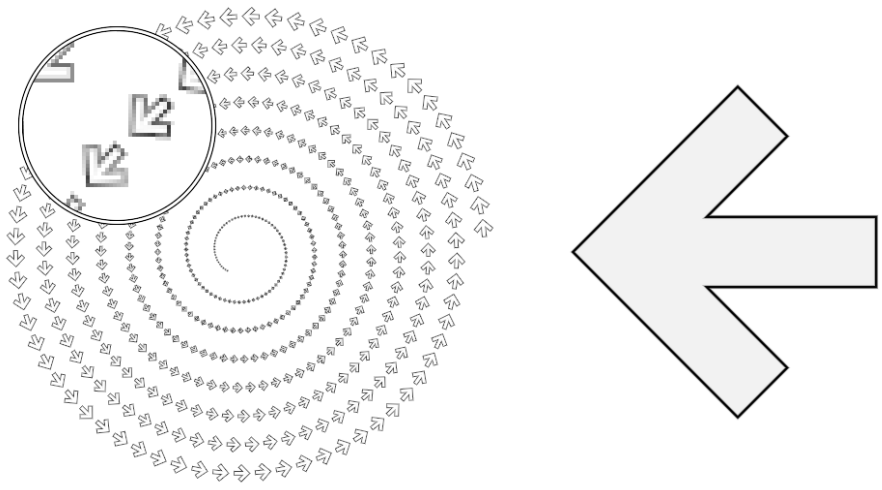

Figure 18. Block arrow marker.

```
float arrow(vec2 P, float size)
{
    float x = P.x;
    float y = P.y;
    float r1 = abs(x) + abs(y) - size/2;
    float r2 = max(abs(x+size/2), abs(y)) - size/2;
    float r3 = max(abs(x-size/6)-size/4, abs(y)- size/4);
    return min(r3,max(.75*r1,r2));
}
```

Listing 21. Block arrow marker.

3.18. Ellipse marker

Ellipse markers are quite different from other markers because there is no easy solution for computing the distance to an ellipse. Fortunately, [Quílez 2009] provided a complete analysis and implementation.

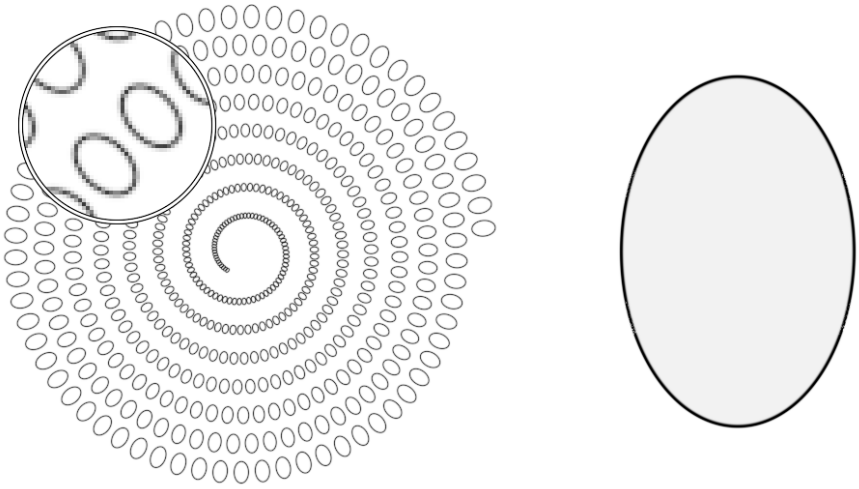

Figure 19. Ellipse marker by Iñigo Quilez.

```
float ellipse(vec2 P, float size) {
// Created by inigo quilez - iq/2013
// License Creative Commons
// Attribution-NonCommercial-ShareAlike 3.0 Unported License.
    vec2 ab = vec2(size/3.0, size/2.0);
    vec2 p = abs( P );
    if( p.x > p.y ){
        p = p.yx;
        ab = ab.yx;
    }
    float l = ab.y*ab.y - ab.x*ab.x;
    float m = ab.x*p.x/l;
    float n = ab.y*p.y/l;
    float m2 = m*m;
    float n2 = n*n;
    float c = (m2 + n2 - 1.0)/3.0;
    float c3 = c*c*c;
    float q = c3 + m2*n2*2.0;
    float d = c3 + m2*n2;
    float g = m + m*n2;
    float co;

    if(d < 0.0)
    {
        float p = acos(q/c3)/3.0;
        float s = cos(p);
        float t = sin(p)*sqrt(3.0);
        float rx = sqrt( -c*(s + t + 2.0) + m2 );
        float ry = sqrt( -c*(s - t + 2.0) + m2 );
        co = ( ry + sign(l)*rx + abs(g)/(rx*ry) - m)/2.0;
    }
    else
    {
        float h = 2.0*m*n*sqrt( d );
        float s = sign(q+h)*pow( abs(q+h), 1.0/3.0 );
        float u = sign(q-h)*pow( abs(q-h), 1.0/3.0 );
        float rx = -s - u - c*4.0 + 2.0*m2;
        float ry = (s - u)*sqrt(3.0);
        float rm = sqrt( rx*rx + ry*ry );
        float p = ry/sqrt(rm-rx);
        co = (p + 2.0*g/rm - m)/2.0;
    }

    float si = sqrt(1.0 - co*co);
    vec2 closestPoint = vec2(ab.x*co, ab.y*si);
    return length(closestPoint - p ) * sign(p.y-closestPoint.y);
}
```

Listing 22. Ellipse marker by Iñigo Quilez.

4. Arrows

Styled arrows require different approaches because only the heads of the arrows are plain, while the bodies are thick line. Furthermore, arrows are parameterized on length and the thickness of the head relatively to the length of the body. I took the same approach as the one introduced by Morgan McGuire [2014] in his quiver plot demonstration, and implemented a set of helper functions to measure the signed distance to an infinite line and line segment, and to compute the center of a circle passing through two points with a specified radius needed for the curved styled arrow.

```
// Computes the signed distance from a line
float line_distance(vec2 p, vec2 p1, vec2 p2) {
    vec2 center = (p1 + p2) * 0.5;
    float len = length(p2 - p1);
    vec2 dir = (p2 - p1) / len;
    vec2 rel_p = p - center;
    return dot(rel_p, vec2(dir.y, -dir.x));
}
```

Listing 23. Signed line distance.

```
// Computes the signed distance from a line segment
float segment_distance(vec2 p, vec2 p1, vec2 p2) {
    vec2 center = (p1 + p2) * 0.5;
    float len = length(p2 - p1);
    vec2 dir = (p2 - p1) / len;
    vec2 rel_p = p - center;
    float dist1 = abs(dot(rel_p, vec2(dir.y, -dir.x)));
    float dist2 = abs(dot(rel_p, dir)) - 0.5*len;
    return max(dist1, dist2);
}
```

Listing 24. Signed segment distance.

```
// Computes the centers of a circle with
// given radius passing through p1 & p2
vec4 inscribed_circle(vec2 p1, vec2 p2, float radius)
{
    float q = length(p2-p1);
    vec2 m = (p1+p2)/2.0;
    vec2 d = vec2( sqrt(radius*radius - (q*q/4.0)) * (p1.y-p2.y)/q,
                   sqrt(radius*radius - (q*q/4.0)) * (p2.x-p1.x)/q);
    return  vec4(m+d, m-d);
}
```

Listing 25. Inscribed circle.

4.1. Curved arrows

The head of a curved arrow is made of the intersection of three circles.

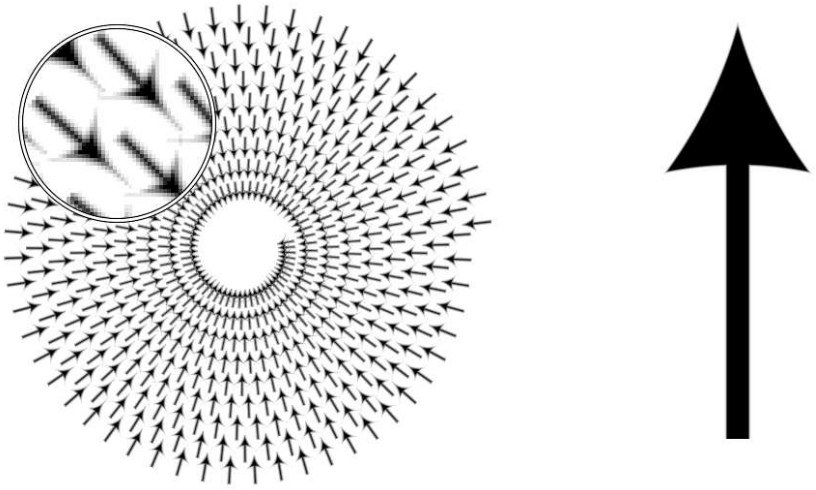

Figure 20. Curved arrow using using $length(head) = 0.25 * length(body)$.

```
float arrow_curved(vec2 texcoord, float body, float head,
                   float linewidth, float antialias)
{
    float w = linewidth/2.0 + antialias;
    vec2 start = -vec2(body/2.0, 0.0);
    vec2 end   = +vec2(body/2.0, 0.0);
    float height = 0.5;

    vec2 p1 = end - head*vec2(+1.0,+height);
    vec2 p2 = end - head*vec2(+1.0,-height);
    vec2 p3 = end;

    // Head :  3 circles
    vec2 c1  = inscribed_circle(p1, p3, 1.25*body).zw;
    float d1 = length(texcoord - c1) - 1.25*body;

    vec2 c2  = inscribed_circle(p2, p3, 1.25*body).xy;
    float d2 = length(texcoord - c2) - 1.25*body;

    vec2 c3  = inscribed_circle(p1, p2, max(body-head, 1.0*body)).xy;
    float d3 = length(texcoord - c3) - max(body-head, 1.0*body);

    // Body :  1 segment
    float d4 = segment_distance(texcoord,
                                start, end - vec2(linewidth,0.0));

    // Outside rejection (because of circles)
```

```
    if( texcoord.y > +(2.0*head + antialias) )
         return 1000.0;
    if( texcoord.y < -(2.0*head + antialias) )
         return 1000.0;
    if( texcoord.x < -(body/2.0 + antialias) )
         return 1000.0;
    if( texcoord.x > c1.x )
         return 1000.0;

    return min( d4, -min(d3,min(d1,d2)));
}
```

Listing 26. Curved arrow.

4.2. Stealth arrows

The head of a stealth arrow is made of the intersection of four half-planes.

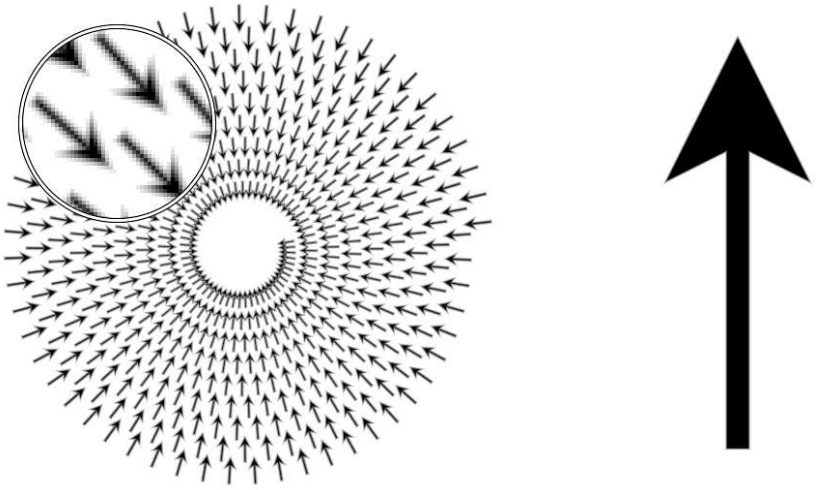

Figure 21. Stealth arrow using using $length(head) = 0.25 * length(body)$.

```
float arrow_stealth(vec2 texcoord, float body, float head,
                    float linewidth, float antialias)
{
    float w = linewidth/2.0 + antialias;
    vec2 start = -vec2(body/2.0, 0.0);
    vec2 end   = +vec2(body/2.0, 0.0);
    float height = 0.5;

    // Head :  4 lines
    float d1 = line_distance(texcoord, end-head*vec2(+1.0,-height), end);
    float d2 = line_distance(texcoord, end-head*vec2(+1.0,-height),
                                       end-vec2(3.0*head/4.0,0.0));
    float d3 = line_distance(texcoord, end-head*vec2(+1.0,+height), end);
    float d4 = line_distance(texcoord, end-head*vec2(+1.0,+0.5),
                                       end-vec2(3.0*head/4.0,0.0));

    // Body :  1 segment
    float d5 = segment_distance(texcoord,
                                start,
                                end - vec2(linewidth,0.0));

    return min(d5, max( max(-d1, d3), - max(-d2,d4)));
}
```

Listing 27. Stealth arrow.

4.3. Triangle arrow

The head of a triangle arrow is made of the intersection of three half-planes.

4.3.1. Generic triangle arrow

```
float arrow_triangle(vec2 texcoord,
                     float body, float head, float height,
                     float linewidth, float antialias)
{
    float w = linewidth/2.0 + antialias;
    vec2 start = -vec2(body/2.0, 0.0);
    vec2 end   = +vec2(body/2.0, 0.0);

    // Head :  3 lines
    float d1 = line_distance(texcoord,
                             end, end - head*vec2(+1.0,-height));
    float d2 = line_distance(texcoord,
                             end - head*vec2(+1.0,+height), end);
    float d3 = texcoord.x - end.x + head;

    // Body :  1 segment
    float d4 = segment_distance(texcoord,
                                start, end - vec2(linewidth,0.0));

    float d = min(max(max(d1, d2), -d3), d4);
    return d;
}
```

Listing 28. Triangle arrow.

4.3.2. *Triangle arrow* ($30°$)

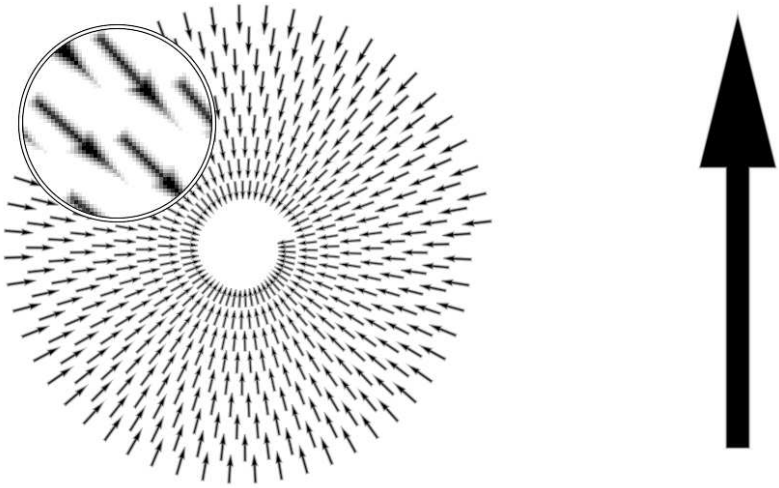

Figure 22. Triangle arrow ($30°$) using $length(head) = 0.25 * length(body)$.

```
float arrow_triangle_30(vec2 texcoord,
                        float body, float head,
                        float linewidth, float antialias)
{
    return arrow_triangle(texcoord, body, head,
                          0.25, linewidth, antialias);
}
```

Listing 29. Triangle arrow ($30°$).

4.3.3. *Triangle arrow (60°)*

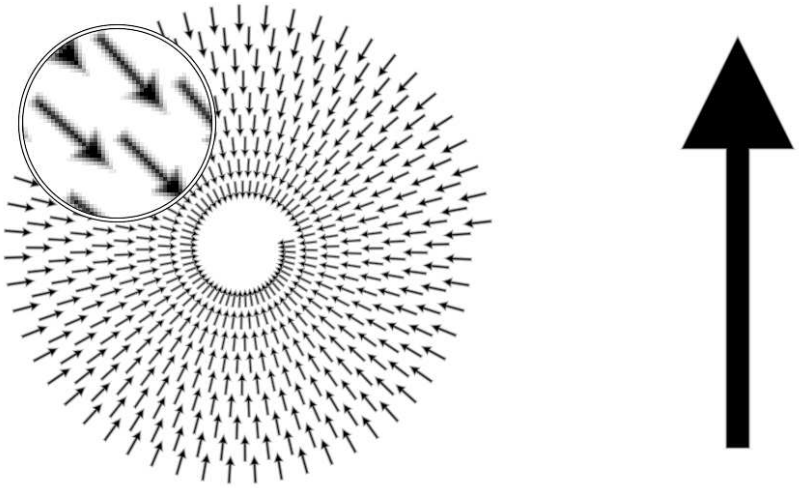

Figure 23. Triangle arrow (60°) using $length(head) = 0.20 * length(body)$.

```
float arrow_triangle_60(vec2 texcoord,
                        float body, float head,
                        float linewidth, float antialias)
{
    return arrow_triangle(texcoord, body, head,
                          0.50, linewidth, antialias);
}
```

Listing 30. Triangle arrow (60°).

4.3.4. *Triangle arrow (90°)*

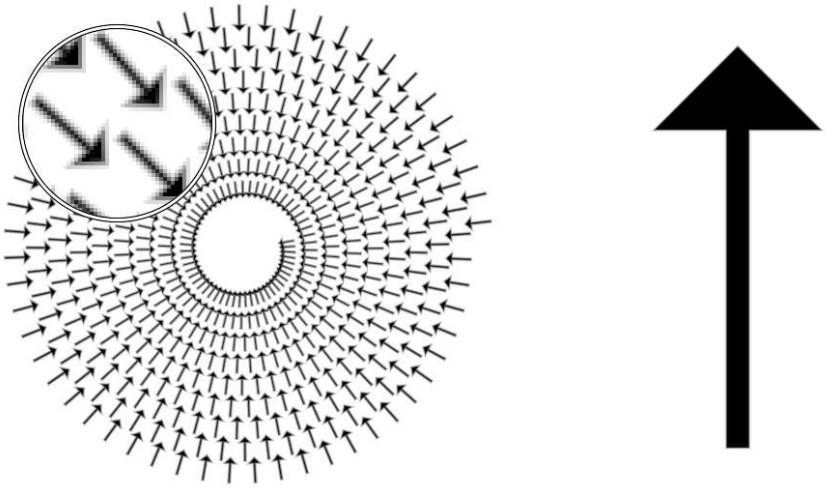

Figure 24. Triangle arrow (90°) using $length(head) = 0.15 * length(body)$.

```
float arrow_triangle_90(vec2 texcoord,
                        float body, float head,
                        float linewidth, float antialias)
{
    return arrow_triangle(texcoord, body, head,
                          1.00, linewidth, antialias);
}
```

Listing 31. Triangle arrow (90°).

4.4. Angle arrow

The head of an angle arrow is made of two lines. Note that we have to consider the tip of the head located beyond the end segment and ensure that head lines are cut and replaced by the corresponding triangle that forms the tip of the head.

4.4.1. Generic angle arrow

```
float arrow_angle(vec2 texcoord,
                  float body, float head, float height,
                  float linewidth, float antialias)
{
    float d;
    float w = linewidth/2.0 + antialias;
    vec2 start = -vec2(body/2.0, 0.0);
    vec2 end   = +vec2(body/2.0, 0.0);

    // Arrow tip (beyond segment end)
    if( texcoord.x > body/2.0) {
        // Head :  2 segments
        float d1 = line_distance(texcoord,
                                 end, end - head*vec2(+1.0,-height));
        float d2 = line_distance(texcoord,
                                 end - head*vec2(+1.0,+height), end);
        // Body :  1 segment
        float d3 = end.x - texcoord.x;
        d = max(max(d1,d2), d3);
    } else {
        // Head :  2 segments
        float d1 = segment_distance(texcoord,
                                 end - head*vec2(+1.0,-height), end);
        float d2 = segment_distance(texcoord,
                                 end - head*vec2(+1.0,+height), end);
        // Body :  1 segment
        float d3 = segment_distance(texcoord,
                                 start, end - vec2(linewidth,0.0));
        d = min(min(d1,d2), d3);
    }
    return d;
}
```

Listing 32. Angle arrow.

4.4.2. *Angle arrow (30°)*

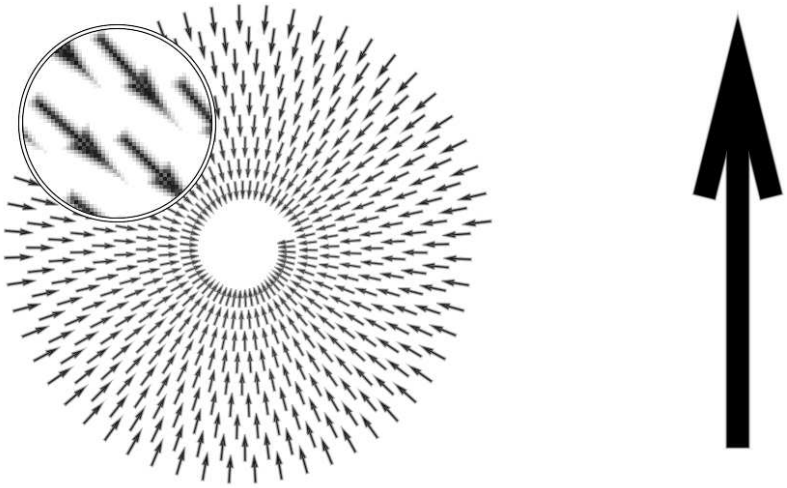

Figure 25. Angle arrow (30°) using $length(head) = 0.33 * length(body)$

```
float arrow_angle_30(vec2 texcoord,
                     float body, float head,
                     float linewidth, float antialias)
{
    return arrow_angle(texcoord, body, head,
                       0.25, linewidth, antialias);
}
```

Listing 33. Angle arrow (30°).

4.4.3. *Angle arrow (60°)*

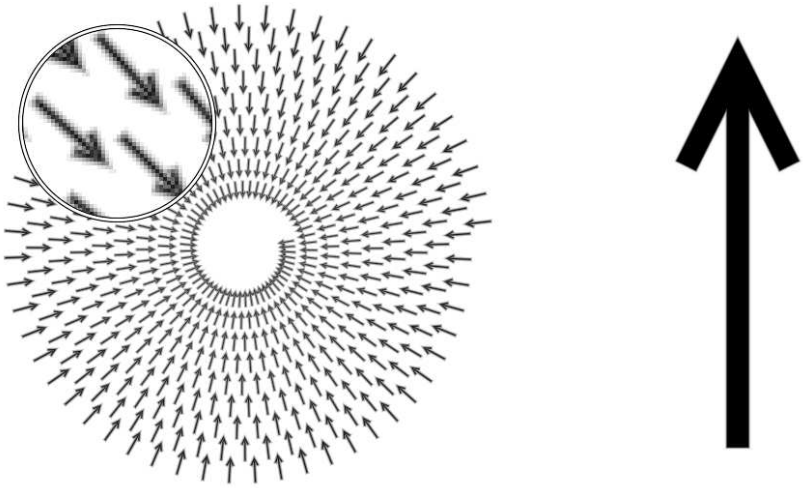

Figure 26. Angle arrow (60°) using $length(head) = 0.25 * length(body)$.

```
float arrow_angle_60(vec2 texcoord,
                     float body, float head,
                     float linewidth, float antialias)
{
    return arrow_angle(texcoord, body, head,
                       0.50, linewidth, antialias);
}
```

Listing 34. Angle arrow (60°).

4.4.4. *Angle arrow (90°)*

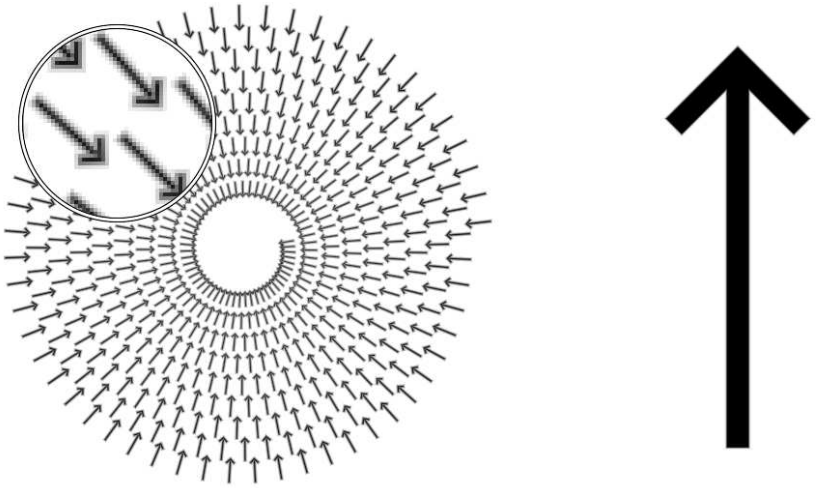

Figure 27. Angle arrow (90°) using $length(head) = 0.15 * length(body)$.

```
float arrow_angle_90(vec2 texcoord,
                     float body, float head,
                     float linewidth, float antialias)
{
    return arrow_angle(texcoord, body, head,
                       1.00, linewidth, antialias);
}
```

Listing 35. Angle arrow (90°).

5. Grids

Grids are the most complex shapes considered in this paper. They require dynamic generation of the implicit surface function, which itself depends on the parameters of the grid:

- Projection function (both forward and inverse).
- Spacing between major and minor grid lines respectively.
- Limits of the cartesian domain (axis 1) containing the projected grid.
- Limits of the projected domain (axis 2).
- Thickness of major and minor grid lines.
- Color of major and minor grid lines.

Fortunately, the implementation is straightforward. Considering a quad whose texture coordinates span the $[-0.5, +0.5]^2$ domain, for any quad fragment with texture coordinates P, we need to:

1. Compute the corresponding coordinates $P1$ in cartesian reference using the `scale_forward` function).
2. Compute the corresponding coordinates $P2$ in projected reference using the `transform_inverse` function).
3. Compute the nearest ticks for all grids (X major, Y major, X minor, Y minor) in projected reference using the `get_tick` function.
4. Compute the cartesian projection of the tick using the `transform_forward` function.
5. Compute the screen distance (in pixels) between the current fragment and all computed ticks location.
6. Compute the dominant color according to the distance of the fragment to the ticks location.

We also must to take into account the border of the grid. If we directly enforced the limits of the projected domain, then the external borders would be cut in half due to their thickness. We instead must ensure those lines are properly drawn by tracking when we are outside the domain, and then drawing those lines later (see listing 39).

The fragment shader is long because of the parameterization. The caller can control the location of ticks, the transform functions, and the projected domain. It works

in both orthographic and perspective mode using a fixed apparent-line thickness and assumes that the quad is lying on the *xz* plane. To make the code easier to read, I present it in four parts:

- Listing 36 Grid Fragment: Parameters
- Listing 37 Grid Fragment: Projection functions
- Listing 38 Grid Fragment: Helper functions
- Listing 39 Grid Fragment: Main function

```
// Line antialias area (usually 1 pixel)
uniform float u_antialias;

// Cartesian and projected limits as xmin,xmax,ymin,ymax
uniform vec4 u_limits1, u_limits2;

// Major and minor grid steps
uniform vec2 u_major_grid_step, u_minor_grid_step;

// Major and minor grid line widths (1.50 pixel, 0.75 pixel)
uniform float u_major_grid_width, u_minor_grid_width;

// Major grid line color
uniform vec4 u_major_grid_color;

// Minor grid line color
uniform vec4 u_minor_grid_color;

// Texture coordinates (from (-0.5,-0.5) to (+0.5,+0.5)
varying vec2 v_texcoord;

// View matrix
uniform mat4 view;

// Model matrix
uniform mat4 model;

// Projection matrix
uniform mat4 projection;

// Viewport resolution (in pixels)
uniform vec2 iResolution;
```

Listing 36. Grid fragment: parameters.

```
// Forward transform (polar)
vec2 transform_forward(vec2 P)
{
    float x = P.x * cos(P.y);
    float y = P.x * sin(P.y);
    return vec2(x,y);
}

// Inverse transform (polar)
vec2 transform_inverse(vec2 P)
{
    float rho = length(P);
    float theta = atan(P.y,P.x);
    if( theta < 0.0 )
        theta = 2.0*M_PI+theta;
    return vec2(rho,theta);
}

// [-0.5,-0.5]x[0.5,0.5] -> [xmin,xmax]x[ymin,ymax]
vec2 scale_forward(vec2 P, vec4 limits)
{
    // limits = xmin,xmax,ymin,ymax
    P += vec2(.5,.5);
    P *= vec2(limits[1] - limits[0], limits[3]-limits[2]);
    P += vec2(limits[0], limits[2]);
    return P;
}

// [xmin,xmax]x[ymin,ymax] -> [-0.5,-0.5]x[0.5,0.5]
vec2 scale_inverse(vec2 P, vec4 limits)
{
    // limits = xmin,xmax,ymin,ymax
    P -= vec2(limits[0], limits[2]);
    P /= vec2(limits[1]-limits[0], limits[3]-limits[2]);
    return P - vec2(.5,.5);
}
```

Listing 37. Grid fragment: projection and scaling.

```
// Antialias stroke alpha coeff
float stroke_alpha(float distance, float linewidth, float antialias)
{
    float t = linewidth/2.0 - antialias;
    float signed_distance = distance;
    float border_distance = abs(signed_distance) - t;
    float alpha = border_distance/antialias;
    alpha = exp(-alpha*alpha);
    if( border_distance > (linewidth/2.0 + antialias) )
        return 0.0;
    else if( border_distance < 0.0 )
        return 1.0;
    else
        return alpha;
}

// Compute the nearest tick from a (normalized) t value
float get_tick(float t, float vmin, float vmax, float step)
{
    float first_tick = floor((vmin + step/2.0)/step) * step;
    float last_tick = floor((vmax + step/2.0)/step) * step;
    float tick = vmin + t*(vmax-vmin);
    if (tick < (vmin + (first_tick-vmin)/2.0))
        return vmin;
    if (tick > (last_tick + (vmax-last_tick)/2.0))
        return vmax;
    tick += step/2.0;
    tick = floor(tick/step)*step;
    return min(max(vmin,tick),vmax);
}

// Compute the distance (in screen coordinates) between A and B
float screen_distance(vec4 A, vec4 B)
{
    vec4 pA = projection*view*model*A;
    pA /= pA.w;
    pA.xy = pA.xy * iResolution/2.0;

    vec4 pB = projection*view*model*B;
    pB /= pB.w;
    pB.xy = pB.xy * iResolution/2.0;

    return length(pA.xy - pB.xy);
}
```

Listing 38. Grid fragment: helper functions.

```
void main()
{
    vec2 NP1 = v_texcoord;
    vec2 P1 = scale_forward(NP1, u_limits1);
    vec2 P2 = transform_inverse(P1);

    // Test if we are within limits but we do not discard the
    // fragment yet because we want to draw border.  Discarding
    // would mean that the exterior would not be drawn.
    bvec2 outside = bvec2(false);
    if( P2.x < u_limits2[0] ) outside.x = true;
    if( P2.x > u_limits2[1] ) outside.x = true;
    if( P2.y < u_limits2[2] ) outside.y = true;
    if( P2.y > u_limits2[3] ) outside.y = true;

    vec2 NP2 = scale_inverse(P2,u_limits2);
    vec2 P;
    float tick;

    // Major tick, X axis
    tick = get_tick(NP2.x+.5, u_limits2[0], u_limits2[1], u_major_grid_step[0]);
    P = transform_forward(vec2(tick,P2.y));
    P = scale_inverse(P, u_limits1);
    // float Mx = length(v_size * (NP1 - P));
    // Here we assume the quad is contained in the XZ plane
    float Mx = screen_distance(vec4(NP1,0,1), vec4(P,0,1));

    // Minor tick, X axis
    tick = get_tick(NP2.x+.5, u_limits2[0], u_limits2[1], u_minor_grid_step[0]);
    P = transform_forward(vec2(tick,P2.y));
    P = scale_inverse(P, u_limits1);
    // float mx = length(v_size * (NP1 - P));
    // Here we assume the quad is contained in the XZ plane
    float mx = screen_distance(vec4(NP1,0,1), vec4(P,0,1));

    // Major tick, Y axis
    tick = get_tick(NP2.y+.5, u_limits2[2], u_limits2[3], u_major_grid_step[1]);
    P = transform_forward(vec2(P2.x,tick));
    P = scale_inverse(P, u_limits1);
    // float My = length(v_size * (NP1 - P));
    // Here we assume the quad is contained in the XZ plane
    float My = screen_distance(vec4(NP1,0,1), vec4(P,0,1));

    // Minor tick, Y axis
    tick = get_tick(NP2.y+.5, u_limits2[2], u_limits2[3], u_minor_grid_step[1]);
    P = transform_forward(vec2(P2.x,tick));
    P = scale_inverse(P, u_limits1);
    // float my = length(v_size * (NP1 - P));
    // Here we assume the quad is contained in the XZ plane
```

```
    float my = screen_distance(vec4(NP1,0,1), vec4(P,0,1));

    float M = min(Mx,My);
    float m = min(mx,my);

    // Here we take care of "finishing" the border lines
    if( outside.x && outside.y ) {
        if (Mx > 0.5*(u_major_grid_width + u_antialias)) {
            discard;
        } else if (My > 0.5*(u_major_grid_width + u_antialias)) {
            discard;
        } else {
            M = max(Mx,My);
        }
    } else if( outside.x ) {
        if (Mx > 0.5*(u_major_grid_width + u_antialias)) {
            discard;
        } else {
            M = m = Mx;
        }
    } else if( outside.y ) {
        if (My > 0.5*(u_major_grid_width + u_antialias)) {
            discard;
        } else {
            M = m = My;
        }
    }

    // Mix major/minor colors to get dominant color
    vec4 color = u_major_grid_color;
    float alpha1 = stroke_alpha( M, u_major_grid_width, u_antialias);
    float alpha2 = stroke_alpha( m, u_minor_grid_width, u_antialias);
    float alpha  = alpha1;
    if( alpha2 > alpha1*1.5 )
    {
        alpha = alpha2;
        color = u_minor_grid_color;
    }

    // Without extra cost, we can also project a texture
    // vec4 texcolor = texture2D(u_texture, vec2(NP2.x, 1.0-NP2.y));
    // gl_FragColor = mix(texcolor, color, alpha);
    gl_FragColor = vec4(color.rgb, color.a*alpha);
}
```

Listing 39. Grid fragment: main.

5.1. Cartesian projection

This is the most simple projection because transformations are actually identity functions. However, to take the border into account, the respective domains are not exactly the same (the cartesian domain needs to be slightly bigger to make room for borders).

- `u_limits1 = [-5.1,+5.1,-5.1,+5.1]`
- `u_limits2 = [-5.0,+5.0,-5.0,+5.0]`
- `u_major_grid_step = [1.0,1.0]`
- `u_minor_grid_step = [0.1,0.1]`
- `u_major_grid_width = 1.50`
- `u_minor_grid_width = 0.75`

```
vec2 transform_forward(vec2 P)
{
    return P;
}

vec2 transform_inverse(vec2 P)
{
    return P;
}
```

Listing 40. Cartesian transformation functions.

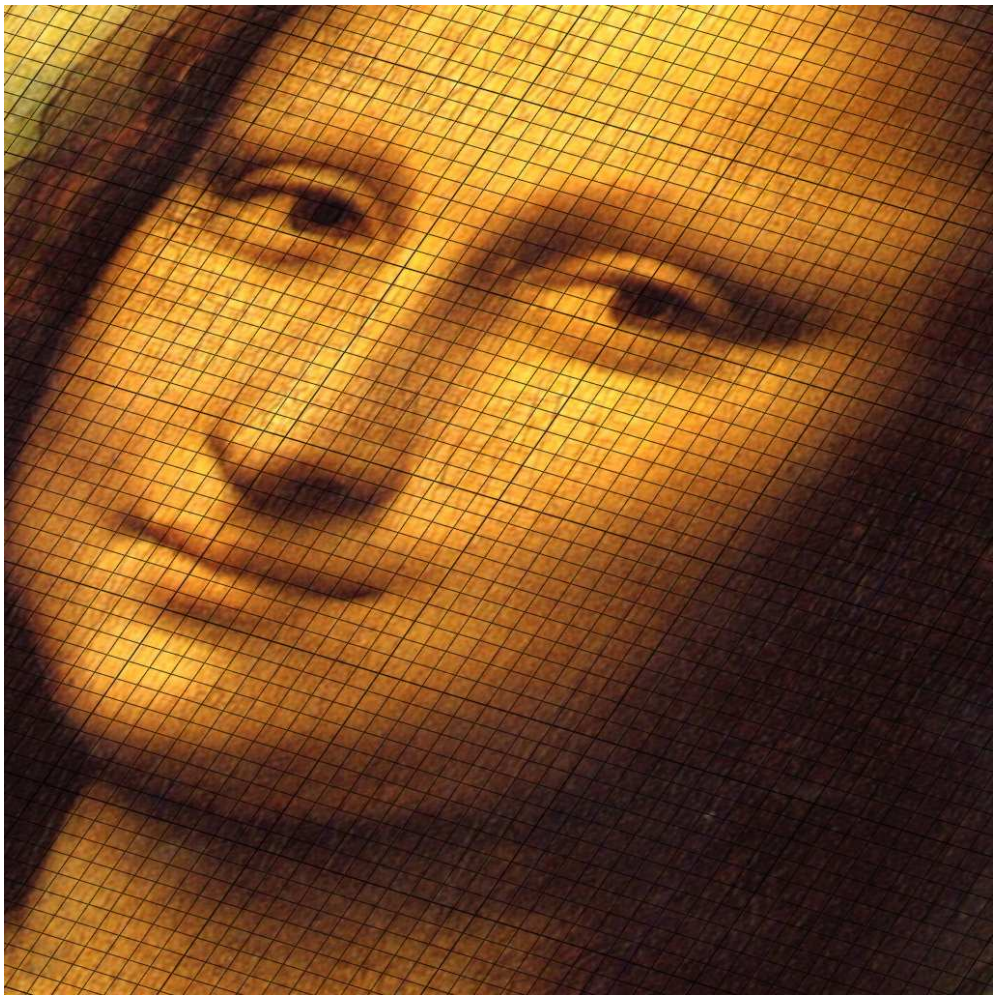

Figure 28. Cartesian projection in perspective mode using grid and texture.

5.2. Polar projection

In a polar projection, the region around $\theta = 0$ is tricky, because a fragment can be considered to belong to either the start region ($\theta > 0$) or the end region ($\theta < 0$). This is mostly visible when the lower limit of the projected domain is 0 and upper limit of the projected domain is strictly less than 2π. So, I handle this case specially.

- `u_limits1 = [-5.1,+5.1,-5.1,+5.1]`
- `u_limits2 = [-5.0,+5.0, M_PI/6.0, 11.0*M_PI/6.0]`
- `u_major_grid_step = [1.00,M_PI/ 6.0]`
- `u_minor_grid_step = [0.25,M_PI/60.0]`
- `u_major_grid_width = 1.50`
- `u_minor_grid_width = 0.75`

```
const float M_PI = 3.14159265358979323846;

vec2 transform_forward(vec2 P)
{
    float x = P.x * cos(P.y);
    float y = P.x * sin(P.y);
    return vec2(x,y);
}

vec2 transform_inverse(vec2 P)
{
    float rho = length(P);
    float theta = atan(P.y,P.x);
    if( theta < 0.0 )
        theta = 2.0*M_PI+theta;
    return vec2(rho,theta);
}
```

Listing 41. Polar transformation functions.

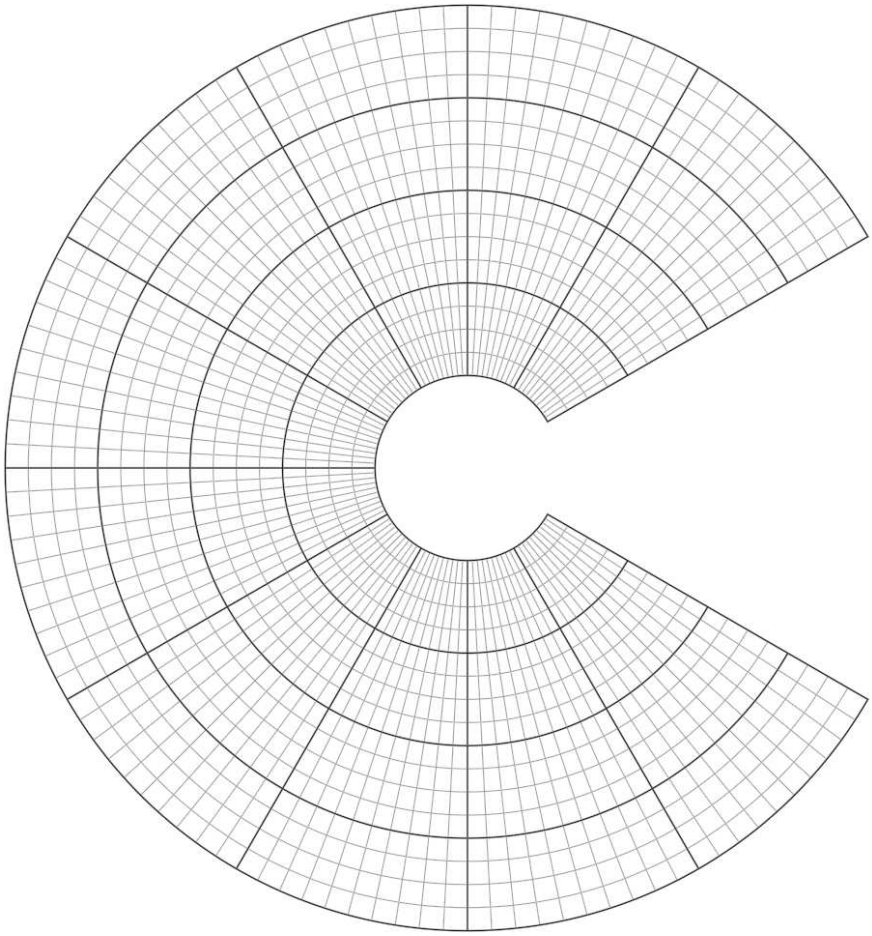

Figure 29. Polar projection using grid only.

5.3. Hammer projection

Hammer projection is one of the standard projection used in cartography to project the Earth's surface onto a plane. For this projection, I use:

- `u_limits1 = [-3.0, +3.0, -1.5, +1.5]`
- `u_limits2 = [-M_PI, +M_PI, -M_PI/3, +M_PI/3]`
- `u_major_grid_step = [M_PI/ 6.0,M_PI/ 6.0]`
- `u_minor_grid_step = [M_PI/30.0,M_PI/30.0]`
- `u_major_grid_width = 1.50`
- `u_minor_grid_width = 0.75`

```
vec2 transform_forward(vec2 P)
{
    const float B = 2.0;
    float longitude = P.x;
    float latitude  = P.y;
    float cos_lat = cos(latitude);
    float sin_lat = sin(latitude);
    float cos_lon = cos(longitude/B);
    float sin_lon = sin(longitude/B);
    float d = sqrt(1.0 + cos_lat * cos_lon);
    float x = (B * M_SQRT2 * cos_lat * sin_lon) / d;
    float y =     (M_SQRT2 * sin_lat) / d;
    return vec2(x,y);
}

vec2 transform_inverse(vec2 P)
{
    const float B = 2.0;
    float x = P.x;
    float y = P.y;
    float z = 1.0 - (x*x/16.0) - (y*y/4.0);
    if (z < 0.0)
        discard;
    z = sqrt(z);
    float lon = 2.0*atan( (z*x),(2.0*(2.0*z*z - 1.0)));
    float lat = asin(z*y);
    return vec2(lon,lat);
}
```

Listing 42. Hammer transformation functions.

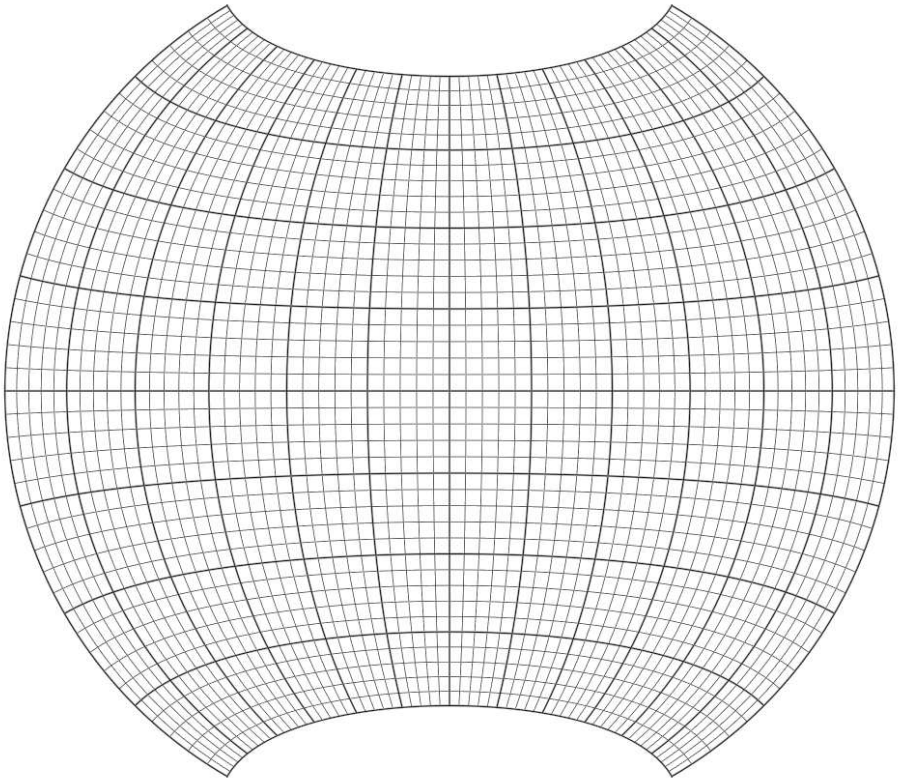

Figure 30. Hammer projection (without pole regions) using a grid only.

5.4. Transverse Mercator

Transverse Mercator is another one of the standard projection used in cartography to project the earth surface onto a plane.

- `u_limits1 = [-3.0, +3.0, -1.5, +1.5]`
- `u_limits2 = [-M_PI, +M_PI, -M_PI/2, +M_PI/2]`
- `u_major_grid_step = [M_PI/ 6.0,M_PI/ 6.0]`
- `u_minor_grid_step = [M_PI/30.0,M_PI/30.0]`
- `u_major_grid_width = 1.50`
- `u_minor_grid_width = 0.75`

```
// Constants
// ------------------------
const float k0 = 0.75;
const float a  = 1.00;

// Helper functions
// ------------------------
float cosh(float x) { return 0.5 * (exp(x)+exp(-x)); }
float sinh(float x) { return 0.5 * (exp(x)-exp(-x)); }

// Forward transform
// ------------------------
vec2 transform_forward(vec2 P)
{
    float lambda = P.x;
    float phi = P.y;
    float x = 0.5*k0*log((1.0+sin(lambda)*cos(phi))
            / (1.0 - sin(lambda)*cos(phi)));
    float y = k0*a*atan(tan(phi), cos(lambda));
    return vec2(x,y);
}

// Inverse transform
// ------------------------
vec2 transform_inverse(vec2 P)
{
    float x = P.x;
    float y = P.y;
    float lambda = atan(sinh(x/(k0*a)),cos(y/(k0*a)));
    float phi    = asin(sin(y/(k0*a))/cosh(x/(k0*a)));
    return vec2(lambda,phi);
}
```

Listing 43. Transverse Mercator transformation functions.

Acknowledgements

This work was inspired by a particular set of online demos:

- Morgan McGuire:
  - https://www.shadertoy.com/view/4s23DG
  - https://www.shadertoy.com/view/4slGDB
- Iñigo Quilez:
  - https://www.shadertoy.com/view/4sS3zz
  - https://www.shadertoy.com/view/MdfGWn

Thanks to Amanda Turner of Williams College for additional text editing.

References

CHAN, E., AND DURAND, F. 2005. *GPU Gems II: Programming Techniques for High-Performance Graphics and General-Purpose Computation* . Addison-Wesley Professional, ch. 22. Fast Prefiltered Lines, 345–369. URL: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter22.html. 1, 3

GREEN, C. 2007. Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 courses*, ACM, New York, NY, USA, SIGGRAPH '07, 9–18. URL: http://www.valvesoftware.com/publications/2007/SIGGRAPH2007_AlphaTestedMagnification.pdf. 3

LOOP, C., AND BLINN, J. 2005. Resolution Independent Curve Rendering Using Programmable Graphics Hardware. In *ACM Trans. Graph.*, ACM, vol. 24, 1000–1009. URL: http://research.microsoft.com/en-us/um/people/cloop/LoopBlinn05.pdf. 2

MCGUIRE, M., 2014. 2d vector field flow. https://www.shadertoy.com/view/4s23DG. 25

MCNAMARA, R., MCCORMACK, J., AND JOUPPI, N. 2000. Prefiltered Antialiased Lines Using Half-Plane Distance Functions. In *ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, ACM, 77–85. URL: http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-98-2.pdf. 3

QUÍLEZ, I., 2009. Distance to an ellipse. http://www.iquilezles.org/www/articles/ellipsedist/ellipsedist.htm. 23

ROUGIER, N. P. 2013. Shader-based antialiased dashed stroked polylines. *Journal of Computer Graphics Techniques 2*, 2, 105–121. URL: http://jcgt.org/published/0002/02/08/. 1

SCHMIDT, R., AND WYVILL, B. 2011. *Sketch-based Interfaces and Modeling*. J. Jorge and F. Samavati, ch. ShapeShop: Free-Form 3D Design with Implicit Solid Modeling. URL: http://www.springer.com/computer/image+processing/book/978-1-84882-811-7. 5

Index of Supplemental Materials

The shaders are available from the *JCGT* website and I have also released them on the `shadertoy.com` website for easy browsing:

- Transverse Mercator grid: `https://www.shadertoy.com/view/lsSXzm`
- Cartesian grid: `https://www.shadertoy.com/view/MdSXRm`
- Polar grid: `https://www.shadertoy.com/view/MsBSRm`
- Hammer grid: `https://www.shadertoy.com/view/ldSXRm`
- Arrows: `https://www.shadertoy.com/view/ldlSWj`
- Markers: `https://www.shadertoy.com/view/XsXXDX`

Author Contact Information

Nicolas P. Rougier
Mnemosyne, INRIA Bordeaux - Sud Ouest
LaBRI, UMR 5800 CNRS, Bordeaux University
Institute of Neurodegenerative Diseases, UMR 5293
351, Cours de la Libération
33405 Talence Cedex, France
Nicolas.Rougier@inria.fr
http://www.loria.fr/~rougier

Nicolas P. Rougier, Antialiased 2D Grid, Marker, and Arrow Shaders, *Journal of Computer Graphics Techniques (JCGT)*, vol. 3, no. 4, 1–52, 2014
http://jcgt.org/published/0003/04/01/

Received:	2014-09-09		
Recommended:	2014-09-10	Corresponding Editor:	Morgan McGuire
Published:	2014-11-02	Editor-in-Chief:	Morgan McGuire

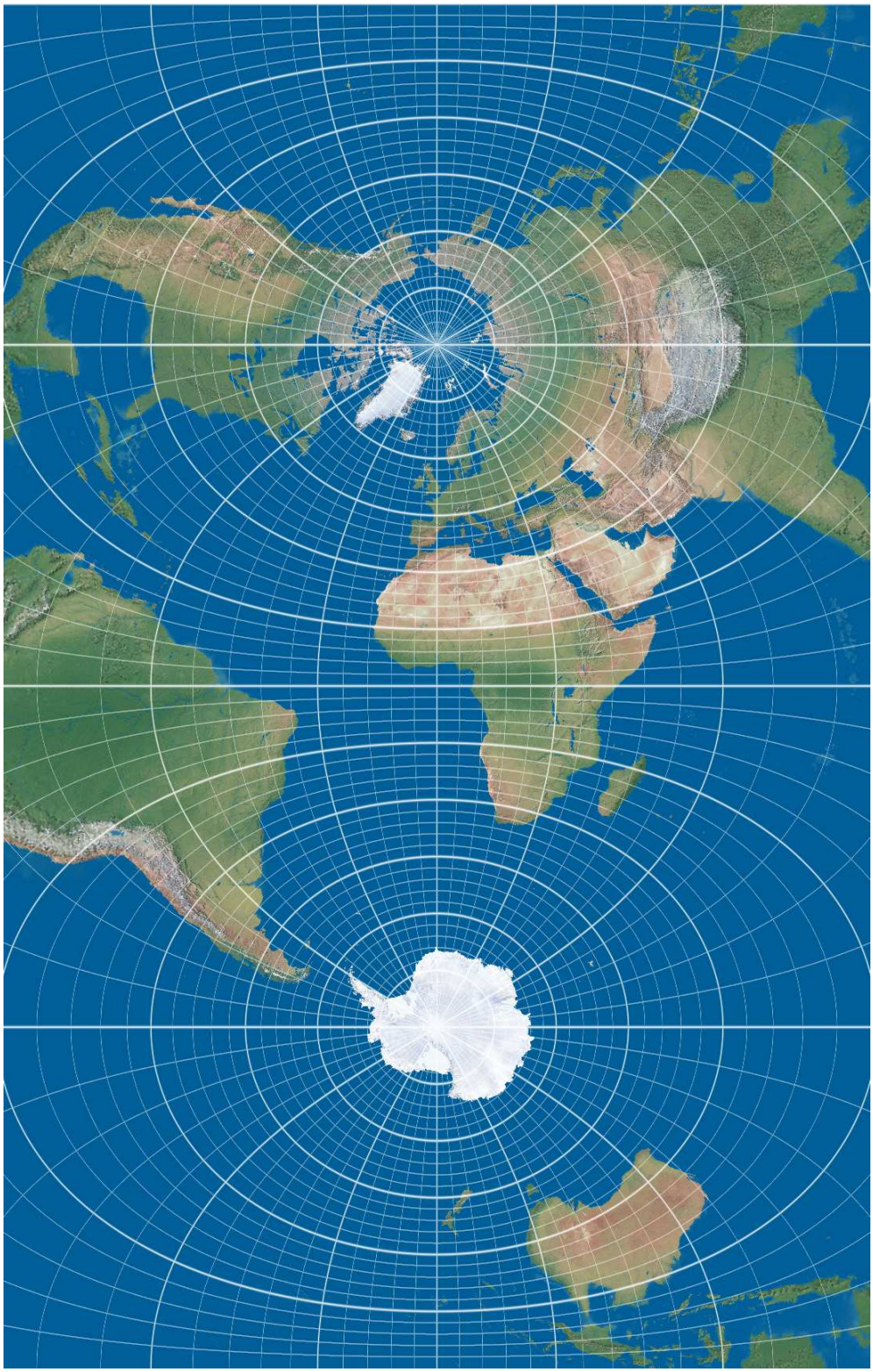

Figure 31. Transverse Mercator projection using grid and texture.