Intro

After having posted about the basics of distance functions in several places (pouet, my blog, shadertoy, private emails, etc), I thought it might make sense to put these together in centralized place. Here you will find the distance functions for basic primitives, plus the formulas for combining them together for building more complex shapes, as well as some distortion functions that you can use to shape your objects. Hopefully this will be usefull for those rendering scenes with raymarching. You can see some of the results you can get by using these techniques in the raymarching distance fields article. Lastly, this article doesn't include lighting tricks, nor marching acceleartion tricks or more advanced techniques as recursive primitives or fractals.

primitives

All primitives are centered at the origin. You will have to transform the point to get arbitrarily rotated, translated and scaled objects (see below).

Sphere - signed - exact

```
float sdSphere( vec3 p, float s )
{
  return length(p)-s;
}
```

Box - unsigned - exact

```
float udBox( vec3 p, vec3 b )
{
  return length(max(abs(p)-b,0.0));
}
```

Round Box - unsigned - exact

```
float udRoundBox( vec3 p, vec3 b, float r )
{
  return length(max(abs(p)-b,0.0))-r;
}
```

Box - signed - exact

```
float sdBox( vec3 p, vec3 b )
{
  vec3 d = abs(p) - b;
  return min(max(d.x,max(d.y,d.z)),0.0) + length(max(d,0.0));
}
```

Torus - signed - exact

```
float sdTorus( vec3 p, vec2 t )
{
  vec2 q = vec2(length(p.xz)-t.x,p.y);
  return length(q)-t.y;
}
```

Cylinder - signed - exact

```
float sdCylinder( vec3 p, vec3 c )
{
  return length(p.xz-c.xy)-c.z;
}
```

Cone - signed - exact

```
float sdCone( vec3 p, vec2 c )
{
    // c must be normalized
    float q = length(p.xy);
    return dot(c,vec2(q,p.z));
}
```

Plane - signed - exact

```
float sdPlane( vec3 p, vec4 n )
{
  // n must be normalized
  return dot(p,n.xyz) + n.w;
}
```

Hexagonal Prism - signed - exact

```
float sdHexPrism( vec3 p, vec2 h )
{
    vec3 q = abs(p);
    return max(q.z-h.y,max((q.x*0.866025+q.y*0.5),q.y)-h.x);
}
```

Triangular Prism - signed - exact

```
float sdTriPrism( vec3 p, vec2 h )
{
    vec3 q = abs(p);
    return max(q.z-h.y,max(q.x*0.866025+p.y*0.5,-p.y)-h.x*0.5);
}
```

Capsule / Line - signed - exact

```
float sdCapsule( vec3 p, vec3 a, vec3 b, float r )
{
    vec3 pa = p - a, ba = b - a;
    float h = clamp( dot(pa,ba)/dot(ba,ba), 0.0, 1.0 );
    return length( pa - ba*h ) - r;
}
```

Capped cylinder - signed - exact

```
float sdCappedCylinder( vec3 p, vec2 h )
{
  vec2 d = abs(vec2(length(p.xz),p.y)) - h;
  return min(max(d.x,d.y),0.0) + length(max(d,0.0));
}
```

Capped Cone - signed - bound

```
float sdCappedCone( in vec3 p, in vec3 c )
{
    vec2 q = vec2( length(p.xz), p.y );
    vec2 v = vec2( c.z*c.y/c.x, -c.z );
    vec2 w = v - q;
    vec2 vv = vec2( dot(v,v), v.x*v.x );
    vec2 qv = vec2( dot(v,w), v.x*w.x );
    vec2 d = max(qv,0.0)*qv/vv;
    return sqrt( dot(w,w) - max(d.x,d.y) ) * sign(max(q.y*v.x-q.x*v.y,w.y));
}
```

Ellipsoid - signed - bound

```
float sdEllipsoid( in vec3 p, in vec3 r )
{
    return (length( p/r ) - 1.0) * min(min(r.x,r.y),r.z);
}
```

Triangle - unsigned - exact

```
float dot2( in vec3 v ) { return dot(v,v); }
float udTriangle( vec3 p, vec3 a, vec3 b, vec3 c )
{
    vec3 ba = b - a; vec3 pa = p - a;
    vec3 cb = c - b; vec3 pb = p - b;
    vec3 ac = a - c; vec3 pc = p - c;
    vec3 nor = cross( ba, ac );

    return sqrt(
    (sign(dot(cross(ba,nor),pa)) +
     sign(dot(cross(cb,nor),pb)) +
     sign(dot(cross(ac,nor),pc))<2.0)
     ?
     min( min(
     dot2(ba*clamp(dot(ba,pa)/dot2(ba),0.0,1.0)-pa),
     dot2(cb*clamp(dot(cb,pb)/dot2(cb),0.0,1.0)-pb) ),
     dot2(ac*clamp(dot(ac,pc)/dot2(ac),0.0,1.0)-pc) )
     :
     dot(nor,pa)*dot(nor,pa)/dot2(nor) );
}
```

Quad - unsigned - exact

```
float dot2( in vec3 v ) { return dot(v,v); }
float udQuad( vec3 p, vec3 a, vec3 b, vec3 c, vec3 d )
{
    vec3 ba = b - a; vec3 pa = p - a;
    vec3 cb = c - b; vec3 pb = p - b;
    vec3 dc = d - c; vec3 pc = p - c;
    vec3 ad = a - d; vec3 pd = p - d;
    vec3 nor = cross( ba, ad );

    return sqrt(
    (sign(dot(cross(ba,nor),pa)) +
```

```
    sign(dot(cross(cb,nor),pb)) +
    sign(dot(cross(dc,nor),pc)) +
    sign(dot(cross(ad,nor),pd))<3.0)
    ?
    min( min( min(
    dot2(ba*clamp(dot(ba,pa)/dot2(ba),0.0,1.0)-pa),
    dot2(cb*clamp(dot(cb,pb)/dot2(cb),0.0,1.0)-pb) ),
    dot2(dc*clamp(dot(dc,pc)/dot2(dc),0.0,1.0)-pc) ),
    dot2(ad*clamp(dot(ad,pd)/dot2(ad),0.0,1.0)-pd) )
    :
    dot(nor,pa)*dot(nor,pa)/dot2(nor) );
}
```

Most of these functions can be modified to use other norms than the euclidean. By replacing length(p), which computes (x^2+y^2+z^2)^(1/2) by (x^n+y^n+z^n)^(1/n) one can get variations of the basic primitives that have rounded edges rather than sharp ones.

Torus82 - signed

```
float sdTorus82( vec3 p, vec2 t )
{
  vec2 q = vec2(length2(p.xz)-t.x,p.y);
  return length8(q)-t.y;
}
```

Torus88 - signed

```
float sdTorus88( vec3 p, vec2 t )
{
  vec2 q = vec2(length8(p.xz)-t.x,p.y);
  return length8(q)-t.y;
}
```

distance operations

The d1 and d2 parameters in the following functions are the distance to the two distance fields to combine together.

Union

```
float opU( float d1, float d2 )
{
    return min(d1,d2);
}
```

Substraction

```
float opS( float d1, float d2 )
{
    return max(-d1,d2);
```

```
}
```

Intersection

```
float opI( float d1, float d2 )
{
    return max(d1,d2);
}
```

domain operations

Where "primitive", in the examples below, is any distance formula really (one of the basic primitives above, a combination, or a complex distance field).

Repetition

```
float opRep( vec3 p, vec3 c )
{
    vec3 q = mod(p,c)-0.5*c;
    return primitve( q );
}
```

Rotation/Translation

```
vec3 opTx( vec3 p, mat4 m )
{
    vec3 q = invert(m)*p;
    return primitive(q);
}
```

Scale

```
float opScale( vec3 p, float s )
{
    return primitive(p/s)*s;
}
```

distance deformations

You must be carefull when using distance transformation functions, as the field created might not be a real distance function anymore. You will probably need to decrease your step size, if you are using a raymarcher to sample this. The displacement example below is using sin(20*p.x)*sin(20*p.y)*sin(20*p.z) as displacement pattern, but you can of course use anything you might imagine. As for smin() function in opBlend(), please read the smooth minimum article in this same site.

Displacement

```
float opDisplace( vec3 p )
{
    float d1 = primitive(p);
    float d2 = displacement(p);
    return d1+d2;
}
```

Blend

```
float opBlend( vec3 p )
{
    float d1 = primitiveA(p);
    float d2 = primitiveB(p);
    return smin( d1, d2 );
}
```

domain deformations

Domain deformation functions do not preserve distances neither. You must decrease your marching step to properly sample these functions (proportionally to the maximun derivative of the domain distortion function). Of course, any distortion function can be used, from twists, bends, to random noise driven deformations.

Twist

```
float opTwist( vec3 p )
{
    float c = cos(20.0*p.y);
    float s = sin(20.0*p.y);
    mat2  m = mat2(c,-s,s,c);
    vec3  q = vec3(m*p.xz,p.y);
    return primitive(q);
}
```

Cheap Bend

```
float opCheapBend( vec3 p )
{
    float c = cos(20.0*p.y);
    float s = sin(20.0*p.y);
    mat2  m = mat2(c,-s,s,c);
    vec3  q = vec3(m*p.xy,p.z);
    return primitive(q);
}
```