Ray tracing of Implicit Surfaces

Normal ray tracing into implicit surfaces

Surface is solution set to $f(P)=0$
Ray is $R(t)$
Intersections are any t which satisfies $f(R(t))=0$
That is: find roots of 1D problem $f(R(t))$
Many root finders exist:
- May need to solve high degree equations
- Some use derivatives
- Many assume continuity

Sphere tracing
- removes many restrictions (no need for derivatives)
- guarantees solution

Signed Distance Field tracing

Define "distance from point P to a surface A" as
$d(P,A)=min_{Q\in A}\|P-Q\|$
Need a function $f(P)$ which **measures** or **bounds** distance to a surface
Several possibilities exist:

- Direct measure (distance function):
  $f(P)=d(P,A)$
- Bounded measure (signed distance bound: SDB):
  $|f(P)|\le d(P,A)$
- *Lipschitz* constant to create a signed distance bound:
  If $|f(P)-f(Q)|\le\lambda\|P-Q\|$ then f is *Lipschitz,* and
  $f(P)/\lambda$ is signed distance bound to $f^{-1}(0)$, and
  $\lambda = Lip\ f$ is the minimal such λ

Algorithm: Ray marching loop pseudo-code

Intersect(Object B, Ray O,D)
$t=\epsilon$ // $\epsilon=10^{-3}$ is a small step to get past the starting point
while true:
$P=O+tD$ // Evaluate ray O,D at t
$dt=B.Distance(P)$ // Distance of P from object B
$t=t+|dt|$ // The meat of the algorithm: Step forward by $|dt|$;

// Various termination conditions
if $|dt|<10^{-6}$: break; // Done, found intersection, lets calculate normal
if TOO MANY STEPS: return NO INTERSECTION // Suggest 2500 max
if $t\ge DistanceLimit$: return NO INTERSECTION // Suggest 10e4

if $t\le\epsilon$: return NO INTERSECTION // Avoid intersecting the originating object

// Calculate Normal (via central difference)
$P=O+tD$
$h=10^{-3}$
$nx=B.Distance(P_x+h,P_y,P_z)-B.Distance(P_x-h,P_y,P_z)$
$ny=B.Distance(P_x,P_y+h,P_z)-B.Distance(P_x,P_y-h,P_z)$
$nz=B.Distance(P_x,P_y,P_z+h)-B.Distance(P_x,P_y,P_z-h)$
$N=Normalize(nx,ny,nz)$
return INTERSECTION at P, N

Distance estimate for CSG operations

Let f_A and f_B be distance estimates for two objects A and B

Then a distance estimate for various CSG operations is:

$f_{A\cup B}(P) = min(f_A(P), f_B(P))$

$f_{A\cap B}(P) = max(f_A(P), f_B(P))$

$f_{A-B}(P) = max(f_A(P), -f_B(P))$

Distance estimate for many simple objects

Plane N, d: $f(P) = N\cdot P + d$

Sphere C, r: $f(P) = \|P-C\| - r$

Cylinder on Z-axis, radius r: $f(P) = \|(P_x, P_y)\| - r$

Cone on Z-axis, angle θ: $f(P) = \|(P_x, P_y)\|\cos\theta - |P_z|\sin\theta$

Torus around Z -axis, radii R, r: $f(P) = \|(\|(P_x, P_y)\| - R,\ P_z)\| - r$

AABox: $max(P_x - x_{max},\ x_{min} - P_x,\ P_y - y_{max},\ y_{min} - P_y,\ P_z - z_{max},\ z_{min} - P_z)$

Distance estimate for transformed objects

For a transformation via $T(P)$ of a surface $f(P)=0$, the transformed surface is

$f(T^{-1}(P))=0$

and so we need the Lipschitz constant $\lambda = Lip\, T^{-1}$

Procedure to calculate $d'(P)$, the distance to an object B transformed by $T()$

Reverse transform P to say, $P' = T^{-1}(P)$

Compute distance bound of P' from object using B's distance $d(P')$

Scale distance by Lipschitz constant $1/\lambda$

That is: $d'(t) = d(T^{-1}(P))/\lambda$

Specific transformations:

Rigid (rotation) transformation $R(\cdots)$: $\lambda=1$

$d'(P) = d(R^{-1}(P))$

Uniform scale by (s, s, s): $\lambda=1/s$

$d'(P) = s\, d(P/s)$

Non-uniform scale by (s_x, s_y, s_z): $\lambda=1/min(s_x, s_y, s_z)$

$d'(P) = min(s_x, s_y, s_z)\, d(P_x/s_x, P_y/s_y, P_z/s_z)$

General linear transformation:

λ is the inverse of the largest Eigen value

Taper

via $r(z)$: $taper(P) = (r(P_z)P_x, r(P_z)P_y, P_z)$

and $\lambda = min_z\, r^{-1}(z)$

Twist:

$$twist(P, \alpha) = \begin{pmatrix} P_x\cos(\alpha P_z) - P_y\sin(\alpha P_z) \\ P_x\sin(\alpha P_z) + P_y\cos(\alpha P_z) \\ P_z \end{pmatrix}$$ where α is the speed of the twist

the Lipschitz constant is $\lambda = Lip\, twist = \sqrt{4 + (\alpha\pi)^2}$ if constrained to a unit cylinder.

So $d'(P) = d(twist(P))/\lambda$

Distance estimate for complex objects

Fields (1D,2D,3D array of an object repeated indefinitely):
d(P,ob):
To get repetition at unit intervals along some axis, say a,
replace // Or use fract or mod
for each axis that want's repetition.

Super Quadrics:

define p-norm in 2D as $\left\|\left(P_x, P_y\right)\right\|^p = \left(\left(|P_x|^p, |P_y|^p\right)\right)^{1/p}$

Generalized 3D spheres: $\left\|\left(\left\|\left(P_x, P_y\right)\right\|^p, P_z\right)\right\|^q$

The largest Euclidean sphere of radius r_e
inscribed withing the generalized sphere of radius r_s is:

$$r_e = \begin{cases} r_s / \left\|\left(\sqrt{3}/3, \sqrt{3}/3, \sqrt{3}/3\right)\right\|^{pq} \\ r_s \end{cases}$$

Soft metablobs:

Defined via a list of n key points P_i with associated radii r_i, and a threshold T:

$$f(P) = T - \sum_{i=1}^{n} C_{r_i}\left(\left\|P - P_i\right\|\right)$$

where $C_r(d) = 2d^3/r^3 - 3d^2/r^2 + 1$ if $d < r$; 0 otherwise

This needs a Lipszhitz const: $\frac{2}{3}\sum r_i$

Distance estimate for fractals:

See: http://blog.hvidtfeldts.net/index.php/2011/06/distance-estimated-3d-fractals-part-i/

Implementation suggestions:

Suggested input language and code organization (think Reverse Polish stack)

Input objects as normal, followed by operations on objects, all in reverse polish notation

objA ...
objB ...
intersection (or **union** or **difference**)

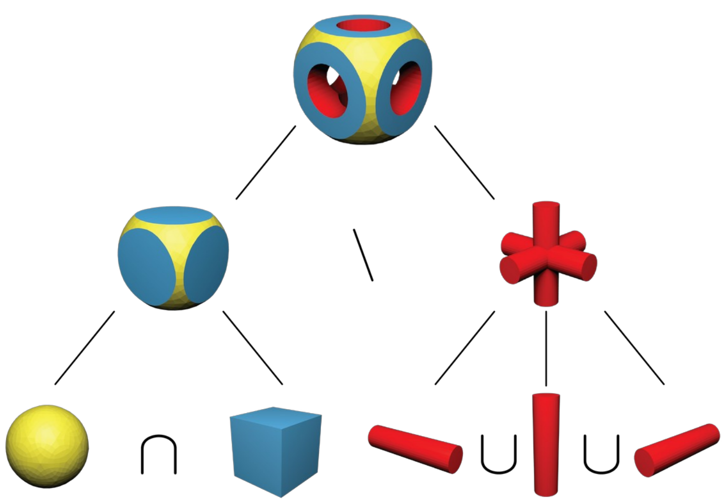
means pop two objects, and push one CSG operator with the two objects as children, and

objA ...
transform ... (could be **rotate**, **scale**, **translate**, **twist**, **taper**, ...)

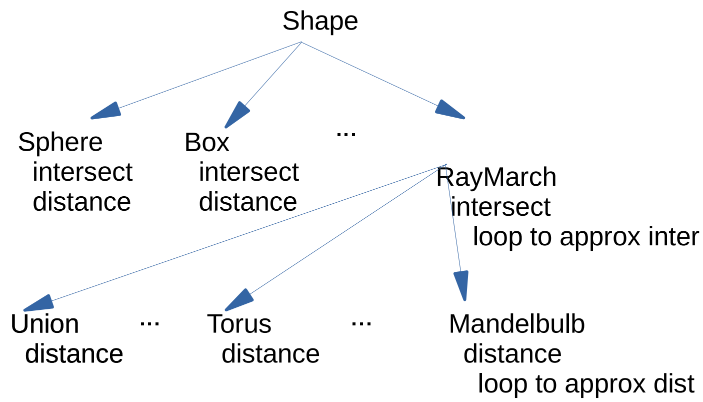
means pop one object, and push one transform object with the the object as a child

Example

sphere ...
box ...
intersect
cylinder ...
cylinder ...
cylinder ...
union
union
difference



Suggested code organization



class Shape: //Pure virtual
    virtual intersect(...)
    virtual float distance(P)

class Sphere: public Shape
    virtual intersect(...)
    virtual float distance(P) { ...}

class RayMarchShape: public Shape //Pure virtual
    virtual intersect(...):
        implements the **Ray marching loop** from page 1,
        calling **distance** as needed
    virtual float distance(P) = 0;

class Union: public RayMarchShape
    virtual float distance(P) { ... }

Any shape for which you can **intersect** directly and calculate **distance:**
- derive from **Shape**
- implement both **intersect** and **distance**
- Examples: sphere, box, cyl, ...

Any shape for which you **can't intersect**, but can calculate **distance**
- derive from **RayMarchShape**
- implement **distance**
- Examples: fractal, transformations(?), CSG, and lots more

Examples:

The Sphere class grows a distance method

```
class Sphere: public Shape
    ...
    virtual double distance(const vec3& P) const {
        return norm((P-center)) - radius; }
```

A new class for a torus

```
class Torus: public RayMarchShape
{
    float R, r;

    Torus(const float _R, const float _r, Material* p): R(_R), r(_r), RayMarch(p)

    distance(const Vector3f& P) const {
        float m = sqrt(P[0]*P[0]+P[1]*P[1]) - R;
        return sqrt(m*m+P[2]*P[2]) - r ; }
}
```

A new Union class provides only a distance function
(Intersection and Difference are similar)

```
class Union: public RayMarchShape
{
    Shape* A;
    Shape* B;

    Union(Shape*  _A, Shape*  _B):  A(_A),  B(_B)   {}

    virtual float distance(const vec3& P) const {
        return min(A->distance(P), B->distance(P)); }
}
```

An object of type Union is created for a “union” line in the input:
(Intersection and Difference are similar)

```
...
else if (c == "union") {
    Shape* obj2 = objs.back();
    objs.pop_back();

    Shape* obj1 = objs.back();
    objs.pop_back();

    objs.push_back(new Union(obj1, obj2, currentMat)); }
...
```

A signed distance calculation for the Mandelbulb

(From http://blog.hvidtfeldts.net/index.php/2011/09/distance-estimated-3d-fractals-v-the-mandelbulb-different-de-approximations/)

```
float DE(vec3 pos) {
  vec3 z = pos;
  float dr = 1.0;
  float r = 0.0;
  for (int i = 0; i < Iterations ; i++) {
        r = length(z);
        if (r>Bailout) break;

        // convert to polar coordinates
        float theta = acos(z.z/r);
        float phi = atan(z.y,z.x);
        dr =  pow( r, Power-1.0)*Power*dr + 1.0;

        // scale and rotate the point
        float zr = pow( r,Power);
        theta = theta*Power;
        phi = phi*Power;

        // convert back to cartesian coordinates
        z = zr*vec3(sin(theta)*cos(phi), sin(phi)*sin(theta), cos(theta));
        z+=pos;
  }
  return 0.5*log(r)*r/dr;
}
```