Numerical Methods for Ray Tracing Implicitly Defined Surfaces[*]

Morgan McGuire
Williams College
September 25, 2014 (Updated October 6, 2014)

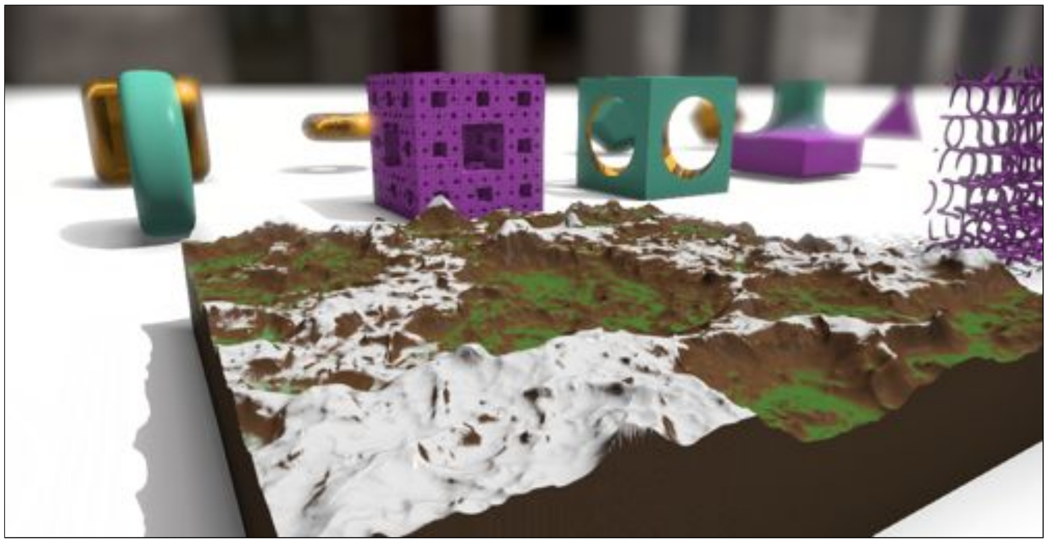


Figure 1: *A 60 fps interactive reference scene defined by implicit surfaces ray traced on GeForce 650M using the techniques described in this document.*

Figure 2: *An impressive demo of real-time ray-traced implicit surface fractals by Kali (Pablo Roman Andrioli), implemented in about 300 lines of GLSL* `https://www.shadertoy.com/view/ldjXzW`.

[*]This document is an early draft of an upcoming *Graphics Codex* chapter.

Contents

1 Primary Surfaces

The first step in a computational graphics solution for rendering an image is finding the surface points that **primary rays** traced backwards from the camera strike. These are the points that can scatter light into the camera, so they are the important ones to shade. The remaining steps are computing the incident illumination at those points, the illumination scattered back along the primary rays, and the filtering and post-processing of all rays to construct an appropriate image for display.

Two dominant methods for finding primary ray intersections with the scene are **ray casting** and **rasterization**. These methods leverage the same underlying mathematics of solving for the point closest to the ray origin that satisfies a primary ray's **explicit equation** and the primitive's (e.g., triangle's, sphere's) **implicit equation**.

An explicit equation for a surface generates points on the surface from parameters; for a ray with origin P and direction $\hat{\omega}$, it is $X(t) = P + \hat{\omega}t$. An implicit equation for a surface defines the set of points on it through an inclusion test; for a sphere with center C and radius r, it is $\{X \mid ||X - C|| = r\}$. Solving for the intersection involves substituting the explicit equation into the implicit one to form a single equation with t as the only variable. and then solving for t. This is equivalent to finding the roots of the 1D scalar version of the implicit equation that lies along the ray.

Ray casting iterates first over rays and second over primitives, while rasterization inverts that structure. This leads to different opportunities for amortization and data structures, and thus different properties for each algorithm. One advantage of ray casting is that it can more easily accommodate a wide range of primitive types.

2 Analytic Ray Intersection

An equation has an **analytic solution** when the exact solution can be found by an algorithm independent. For example, the solution to any second-order equation in variable t,

$$at^2 + bt + c = 0 \tag{1}$$

is given by the quadratic formula,

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{2}$$

Analytic solutions exist for the intersection of a ray and certain primitives. These include the sphere/ball, plane, planar polygon (notably, the triangle), cylinder, cone, and torus. It is not surprising that these surfaces admit analytic solutions, since implicit equations are first- and second-order polynomials amenable to the quadratic formula. It should also not be surprising that there is no analytic solution for the intersection of a ray with a surface defined by a fifth-order polynomial, since no closed-form solution to a general quintic in terms of radicals can exist. (Technically, third- and fourth-order surfaces could be intersected analytically, however, the closed-form solutions are computationally demanding compared to the quadratic formula.) Some non-polynomial equations (such as simple trigonometric expressions) admit analytic solutions, although many do not.

3 Numerical Ray Intersection

The existence of implicitly defined surfaces for which there is no analytic solution to ray intersection motivates a general method for ray-primitive intersection. Fortunately, iterative numerical methods for root-finding have been well studied for centuries. Techniques such as Newton-Raphson iteration, the secant method, and gradient descent perform a guided search over the space of a function to find its roots. Different methods have different convergence properties and guarantees, mostly based on what is known about the function itself.

4 Ray Marching

Consider an example for of solving for the root of the equation that gives the intersection of a ray and a sphere:

$$g(t) = ||X + \hat{\omega}t - C|| - r = 0 \tag{3}$$

A simple approach called **ray marching** iterates t through the domain of function g in fixed steps[1]:

```
def findRootNaive(g):
    dt = 0.01
    t = 0
    while g(t) != 0:
        t = t + dt
    return t
```

This "marches" through every point $X = P + \hat{\omega}t$ on the ray until it finds a t that yields X on the surface of the sphere. Of course, this naive ray marching scheme is unlikely to exactly land on the exact value of t for which $g(t) = 0$, so the code will almost always loop forever. A better solution enforces a maximum distance bound and seeks the first value at which g crosses from positive to nonpositive, at which point the ray must have passed through the surface:

```
def findRoot(g, maxDistance):
    dt = 0.01
    t = 0
    while (t < maxDistance) and (g(t) > 0):
        t = t + dt
    return t
```

We can further refine this approach using binary search within the final fixed interval $[t_{n-1}, t_n]$ where g passed from positive to nonpositive.

Ray marching with fixed steps has three advantages over analytic ray intersections:

- Very simple to implement
- Works for *any* implicitly defined surface
- Naturally extends to integration of scattering in participating media such as fog

Ray marching with fixed steps has two obvious disadvantages over analytic ray tracing. The first is that it can miss the intersection for a non-convex shape. The second is that it takes at least linear time in the distance to the intersection for rays that do intersect the surface and linear time in `maxDistance` for rays that do not intersect. Analytic ray tracing is guaranteed to not only find the intersection, but do it in constant time.

We can address both the performance disadvantage and the missed intersection disadvantage of fixed-step ray marching with a simple improvement that only slightly constrains the surface description.

[1]I'm using Python/pseudocode here because higher-order function types have awkward syntax many other languages and would obscure the interesting parts of the iteration. I switch to GLSL for the "cookbook" sections of this document.

5 Distance Estimators

We originally defined $g(t) = 0$ as the surface and implicitly required only that $g(t) > 0$ at the ray origin P and g was continuous. Now, let us require the surface to be an embedded manifold without boundary, i.e., the surface of a solid object, and refine our constraints so that g is a **signed distance estimator**: require that $g(t)$ is less than or equal to than the distance to the surface.

Because we're working with surfaces in three dimensions and will wish to cast rays from many different origins (e.g., primary rays, shadow rays, mirror rays), it will be convenient to define a distance estimator on three-space instead of the ray parameter. Let "minimum height above the surface" $h(X) = g(P + \hat{\omega}t)$ be a signed distance estimator for a surface. For the sphere, one function that satisfies this is simply the original implicit equation: $h_{\text{sphere}}(X) = ||X - C|| - r$.

The $h = 0$ surface is sometimes called *the* **isosurface**, although technically it is the $h = 0$ isosurface. Fluids are often modeled with a large number of particles, which are collectively rendered by forming a single distance estimator that is the sum of a large number of "blob" distance estimators. In this case, the term **level set** is sometimes used to describe the surface.

6 Sphere Tracing

Given a distance estimator, $h(X)$, we now have an upper bound on how far is safe to step to avoid marching the ray over the intersection. Because h returns a bound on the distance in *any* direction, we can move at least that far along the direction $\hat{\omega}$ of the ray. In other words, if we put a sphere of radius $h(X)$ at point X, it at most touches the primitive (which is also a sphere in our example) that we're tracing against, and may not even touch it. That new virtual sphere can't possibly intersect the primitive more deeply because of the distance estimator guarantee. So, we can jump to testing the next point at the surface of the virtual sphere. This method for advancing the ray (which traces rays against arbitrary primitives and has no "spheres" necessarily involved) is called **sphere tracing** [Hart 1996].

```
def sphereTrace(P, w, h, maxDistance):
    epsilon = 0.0000001 # How close we want to get to the surface
    minStep = 0.0001 # Minimum step through space to enforce

    t = 0
    dt = h(P + w * t)
    while (t < maxDistance) and (dt > epsilon):
        t = t + max(dt, minStep)
        dt = h(P + w * t)
    return t
```

Why is iteration necessary? Doesn't h give the distance to the surface immediately? If h gave the exact distance to the surface, then the `while` loop would be unnecessary. However, recall that h is a *conservative estimate* of the distance. If the surface is 1m from a point X, then $h(X)$ might return 0.1m. So, we must iterate until $h(X)$ converges to zero, or at least very nearly so.

The `minStep` constant is necessary to ensure that the ray continues to advance even when asymptotically approaching a surface. It is particularly important for achieving reasonable performance in the face of an overly conservative distance estimator or at locations where the ray just barely passes by the surface. One could additionally impose an explicit iteration limit independent of the minimum step size. Keinert et al. [2014] recommend tracking and returning t corresponding to the closest point ever found to the ray instead of the marching end point when terminating after a fixed number of iterations.

As before, we can extend this root finding method to perform binary search to refine the end hit point. Because h is a *signed* distance estimator, we can even do so fairly efficiently, since once the ray passes through the surface we can re-approach it accurately from the inside.

7 Some Distance Estimators

This section is adapted from, and uses images from, http://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm by Iñigo Quilez, 2008.

For these definitions, let $\max(\vec{v}) = \max(x_v, y_v, z_v)$, $|\vec{v}| = (|x_v|, |y_v|, |z_v|)$, and let $\max(\vec{a}, \vec{b}) = (\max(x_a, x_b), \max(y_a, y_b), \max(z_a, z_b))$. The code samples are in GLSL using some common utilities and HLSL compatibility definitions. These are defined in the G3D Innovation Engine's `g3dmath.glsl` and `compatibility.glsl` files. For example,

```
#define Point3 vec3
#define lerp mix
float maxComponent(vec3 v) { return max(v.x, max(v.y, v.z)); }
float saturate(float x) { return clamp(x, 0.0, 1.0); }
```

7.1 Sphere

The sphere about point C with radius r:

$$h(X) = \|X - C\| - r \tag{4}$$

```
float sphereDistance(Point3 X, Point3 C, float r) {
    return length(X - C) - r;
}
```

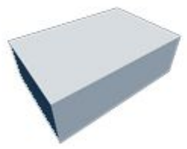
7.2 Plane

The plane with point C and normal $\hat{n}$:

$$h(X) = (X - C) \cdot \hat{n} \tag{5}$$

```
float planeDistance(Point3 X, Point3 C, Vector3 n) {
    return dot(X - C, r);
}
```

7.3 Box



The box with center C and vector $\vec{b}$ from the center to the first-quadrant corner:

$$\text{Let } \vec{d} = |X - C| - \vec{b}$$

$$h(X) = \min(\max(d), 0) + \|\max(d, (0,0,0))\|\,; \tag{6}$$

```
float boxDistance(Point3 X, Vector3 b) {
    Vector3 d = abs(X - C) - b;
    return min(maxComponent(d), 0) + length(max(d, Vector3(0, 0, 0)));
}
```

7.4 Rounded Box

The hollow rounded box with center C, edge half-lengths in vector $\vec{b}$, and rounding radius r:

$$h(X) = \left\| \max\left(|X - C| - \vec{b}, (0,0,0) \right) \right\| - r; \tag{7}$$

```
float roundedBoxDistance(Point3 X, Vector3 b, float r) {
  return length(max(abs(X - C) - b, Vector3(0, 0, 0))) - r;
}
```

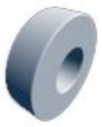
7.5 Torus

The torus about the x-axis with centroid C, minor radius r, and major radius R:

$$h(X) = \left\| \left(\left\| \sqrt{(x_X - x_C)^2 + (z_X - z_C)^2} - r \right\|, y_X - y_C \right) \right\| - R \tag{8}$$

```
float torusDistance(Point3 X, Point3 C, float r, float R) {
    return length(vec2(length(X.xz - C.xz) - r, X.y - C.y)) - R;
}
```

7.6 Wheel



The "wheel" about the x-axis with centroid C, minor radius r, and major radius R is the same as the torus above, but with a non-Euclidean definition of "distance":

$$h(X) = \left\| \left(\left\| \sqrt{(x_X - x_C)^2 + (z_X - z_C)^2} - r \right\|, y_X - y_C \right) \right\|_8 - R \tag{9}$$

Here, $\|\vec{v}\|_n$ is the n-norm: $\sqrt[n]{x_v^n + y_v^n + z_v^n}$. Higher-order norms like this can be substituted into most distance estimators to square off edges; compare the wheel on the right to the torus above it.

```
float pow8(float x) {
    x *= x; // x^2
    x *= x; // x^4
    return x * x;
}

float length8(Vector2 v) {
    return pow(pow8(v.x) + pow8(v.y), Vector2(1.0 / 8.0, 1.0 / 8.0));
}

float wheelDistance(Point3 X, Point3 C, float r, float R) {
    return length8(Vector2(length(X.xz - C.xz) - r, X.y - C.y)) - R;
}
```

7.7 Cylinder

The cylinder with centroid C, radius r, and half-extent e:

$$\text{Let } \vec{d} = \left|\left(\left\|\sqrt{(x_X - x_C)^2 + (z_X - z_C)^2} - r\right\|, y_X - y_C\right)\right| - (r, e) \tag{10}$$

$$h(X) = \min(\max(x_d, y_d), 0) + \|\max(\vec{d}, (0,0))\| \tag{11}$$

```
float cylinderDistance(Point3 X, Point3 C, float r, float e) {
    Vector2 d = abs(Vector2(length(X.xz - C.xz), X.y - C.y)) - Vector2(r, e);
    return min(maxComponent(d), 0) + length(max(d, Vector2(0, 0)));
}
```

8 Computing Normals

The **gradient** of any function on 3-space can be approximated by numerical differentiation:

$$\begin{aligned}\nabla h(X) &= \left(\frac{\partial h(X)}{\partial x}, \frac{\partial h(X)}{\partial y}, \frac{\partial h(X)}{\partial z}\right) \\ &\approx \tfrac{1}{2\epsilon}\left(h(X + \hat{x}\epsilon) - h(X - \hat{x}\epsilon), h(X + \hat{y}\epsilon) - h(X - \hat{y}\epsilon), h(X + \hat{z}\epsilon) - h(X - \hat{z}\epsilon)\right) \\ &\approx \tfrac{1}{\epsilon}\left[h(X) - (h(X + \hat{x}\epsilon), h(X + \hat{y}\epsilon), h(X + \hat{z}\epsilon))\right] \end{aligned} \tag{12}$$

If the distance estimator is not too conservative near the surface, or at least is conservative by the same factor in all directions, we can use this to find the surface normal.

For a point X *near* the surface, normalizing the gradient by dividing through by its length gives the **surface normal** vector to the nearby surface. Placing X exactly *on* the surface can be problematic for this computation, however. For example, for a sufficiently curvy surface, a particular ϵ value chosen may place all samples of h at another location on the surface, for which $h = 0$.

9 A Simple GLSL Ray Caster

This section describes the ray casting portion of a simple GLSL ray marching tracer that runs on the GPU using the G3D Innovation Engine 10. See `http://codeheartjs.com/examples/raytrace/` for a comparable Javascript version that runs in a web browser, and `http://codeheartjs.com/examples/fastraytrace/` for an optimized Javascript version. See `http://graphics.cs.williams.edu/courses/cs371/f14/reading/gpu-tracing-tutorial.zip` for a working example of an *analytic* ray tracer in C++ and GLSL.

The GLSL ray marcher below uses rasterization to submit two giant triangles that cover the entire viewport with a 2D projection matrix. It then launches the ray tracing shader kernel at each pixel, passing it the 3D camera's orientation and projection matrix. The C++ syntax for setting up this call in OpenGL is:

```
rd->push2D();
Args args;

args.setUniform("cameraToWorldMatrix",    activeCamera()->frame());

args.setUniform("tanHalfFieldOfViewY",
    float(tan(activeCamera()->projection().fieldOfViewAngle() / 2.0f)));

// Projection matrix, for writing to the depth buffer.
Matrix4 projectionMatrix;
activeCamera()->getProjectUnitMatrix(rd->viewport(), projectionMatrix);
args.setUniform("projectionMatrix22", projectionMatrix[2][2]);
args.setUniform("projectionMatrix23", projectionMatrix[2][3]);

// A cube map Texture
m_skybox->setShaderArgs(args, "skybox_", Sampler::cubeMap());

// Set the domain of the shader to the viewport rectangle
args.setRect(rd->viewport());

LAUNCH_SHADER("trace.pix", args);
rd->pop2D();
```

The GLSL code within the referenced `trace.pix` shader sets up the primary ray and then marches it through the scene. I've also rigged it to write a depth buffer value that is compatible with rasterization so that rasterized and ray traced primitives may be mixed in the frame buffer. Furthermore, this enables the use of standard post-processing tricks such as single-image depth of field, or even depth-buffer based ambient occlusion as a post process.

```
#version 330 // -*- c++ -*-
#include <g3dmath.glsl>
#include <Texture/Texture.glsl>

// Input arguments from the C++ program
uniform mat4x3         cameraToWorldMatrix;
uniform float          tanHalfFieldOfViewY;
uniform float          projectionMatrix22, projectionMatrix23;
uniform_Texture(samplerCube, skybox_);

// Output to the App::m_framebuffer
out Color3 pixelColor;

void main() {
    // Generate an eye ray in camera space, and then transform to world space

    // Primary ray origin
    Point3 P  = cameraToWorldMatrix[3];

    // Primary ray direction
    Vector3 w = Matrix3(cameraToWorldMatrix) *
        normalize(Vector3((gl_FragCoord.xy - g3d_FragCoordExtent / 2.0) *
                Vector2(1, -1), g3d_FragCoordExtent.y /
        ( -2.0 * tanHalfFieldOfViewY)));

    float distance = inf;
    pixelColor = traceRay(P, w, distance);

    // Camera space z value
    float csZ = maxDist / w.z;

    // Pack into standard OpenGL depth buffer format to make the result
    // compatible with rasterization and post-processing.
    gl_FragDepth = (maxDist == inf) ? 1.0 :
        ((projectionMatrix22 * csZ + projectionMatrix23) / -csZ);
}
```

The actual tracing code is below. This example defines the scene to be a single, yellow sphere at the origin with a radius of 1. To make it easier to orient the viewer when investigating the scene with the default camera, I also placed a cube map skybox in the background.

For a full ray tracer, we'd shade the intersection ray and possibly spawn additional rays for reflection, refraction, and shadows.

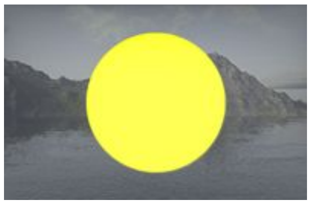


Figure 3: *The result of the simple GLSL ray marcher.*

```
float sceneDistance(Point3 X) {
    const Point3  C = Point3(0,0,0);
    const float   r = 1.0;
    return length(X - C) - r;
}

// Returns true if something was hit.
// Sets L_o if the ray reached infinity or if it hit something before distance
// If something was hit, updates distance
bool traceRay(Point3 P, Vector3 w, out L_o, inout float distance) {
    const float maxDistance = 1e10;
    const int maxIterations = 50;
    const float closeEnough = 1e-2;
    float t = 0;
    for (int i = 0; i < maxIterations; ++i) {
    float dt = sceneDistance(P + w * t);
        t += dt;
        if (dt < closeEnough) {
            distance = t;
            L_o = Radiance3(1, 1, 0);
            return true;
        } else if (t > distance) {
            // Too far; there is some known closer intersection
            return false;
        }
    }

    // Reached infinity
    L_o = texture(skybox_buffer, w).rgb * skybox_readMultiplyFirst.rgb;
    return false;
}
```

This kind of setup is common in the modern demoscene and many examples of artistically impressive demos and intros based around this technique can be found at `pouet.net`. Smaller single-shader examples with source code can be found at `shadertoy.com` and `glslsandbox.com`.

10 Operations on Distance Estimators

An advantage of modeling surfaces with distance estimators is that it is much easier to perform operations on whole shapes in that model than it is when they have explicit triangle representations or direct implicit definitions.

Operators on distance estimators are higher-order functions: they take functions as input and return functions as output. In a language such as Javascript or Scheme that has full support for closures, this can be implemented directly with functions. See `http://codeheartjs.com/`

examples/raytrace for an example that runs in a web browser. In a language such as Java or C++, classes and inheritance can mimic the design pattern. A simple substitution-compiled language like GLSL with provides none of these language features for abstracting computation. They can be emulated using macros and overloading or one can directly expand that code by hand, for example,

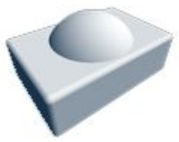
```
float unionDistance(float d1, float d2) {
    return min(d1, d2);
}

float doubleSphere(Point3 X) {
    return unionDistance(sphereDistance(X, Point3(-1, 0, 0), 1),
                         sphereDistance(X, Point3(1, 0, 0), 1));
}
```

11 Some Useful Operators

This section is adapted from, and uses images from, http://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm by Iñigo Quilez, 2008.

11.1 Union



Union selects the closer surface. This is equivalent to a set sum (but not a distance sum).

$$(h \cup g)(X) = \min(h(X), g(X)) \tag{13}$$

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

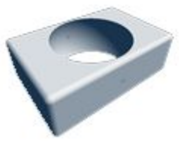
11.2 Intersection

Intersection selects the farther surface.

$$(h \cap g)(X) = \min(h(X), g(X)) \tag{14}$$

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

11.3 Subtraction



Set subtraction inverts one of the functions and then intersects it with the other.

$$(h - g)(X) = \max(h(X), -g(X)) \tag{15}$$

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

11.4 Repetition

Any distance function can be tiled across space with period $\vec{v}$ across the dimensions.

$$\text{repeat}(h, \vec{p})(X) = h\left((X \bmod \vec{v}) - \vec{v}/2\right) \tag{16}$$

Note that floating-point modulo in GLSL is notated with the `mod` function, not the `%` operator.

11.5 Transformation

Transforming a shape is equivalent (from the reference frame of the distance estimator) to inversely transforming the points tested against it. In doing so, we need to be careful about how we have scaled space, however. To transform a shape by an invertible 4×4 rotation-translation-dilation (uniform, non-zero scale) matrix M,

$$(Mh)(X) = h(M^{-1}X) \cdot \det M, \tag{17}$$

where the expression on the right is the determinant M.

Because rotation, translation, and non-zero dilation all have trivial inverses and determinants it is not usually necessary to apply the full inverse and determinant operations. For example scaling a primitive by scalar factor $s > 0$ is simply:

$$\text{scale}(h, s)(X) = h(X/s) \cdot s. \tag{18}$$

Note that if a transformation of a simple primitive is desired, it may be more efficient to let $C = (0, 0, 0)$ in the original primitive's definition so that it is centered at the origin and then perform a single transformation to the desired reference frame.

11.6 Blending

With some care to preserve the conservative property of distance estimation, shapes can be blended for a kind of smooth union. Hoffman and Hopcroft [1985] derive the general case of this, which is commonly referred to as a union using a "smooth min" (smin) function in place of a straight min function. There are many smooth min functions, with a variety of performance and quality tradeoffs. Quilez and I recommend a fast polynomial version:

```
float smin(float a, float b, float blendRadius) {
    float c = saturate(0.5 + (b - a) * (0.5 / blendRadius));
    return lerp(b, a, c) - blendRadius * c * (1.0 - c);
}
```

12 Increasing Performance

Performance is always a concern in rendering. The performance of ray marching can be increased at the cost of image quality by increasing the minimum step size and accepted distance from the surface, and by decreasing the maximum iterations permitted and resolution. However, it can also be increased by algorithmic improvements that do not affect image quality. Instead, the cost is some of the elegant simplicity of the naive method. However, Compared to analytic ray casting and rasterization, optimized ray marching remains remarkably straightforward and accessible to implement.

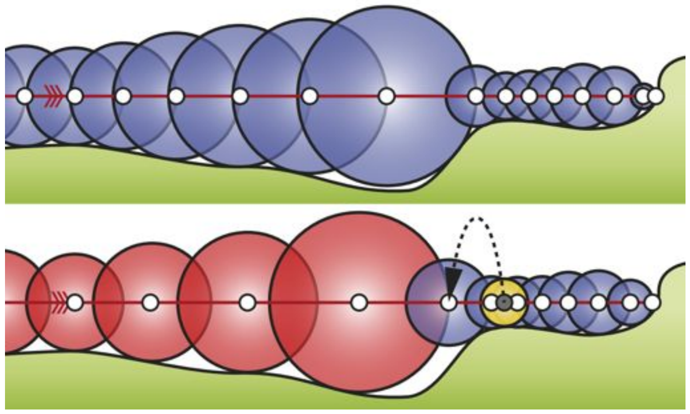


Figure 4: *Top: sample points in white and "unbounding spheres" in purple from sphere tracing. Bottom: Over-relaxed sphere tracing can speculatively take larger steps (red) until it fails to find consecutive bounds (yellow) and falls back to sphere tracing.*

12.1 Over-Relaxation

Keinert et al. [2014] make an observation that allows decreasing the number of iterations before binary search by up to a factor of two. Hart's sphere tracing uses the distance estimator h creates a series of "unbounding" spheres as the ray approaches a surface. The $h(X) = 0$ isosurface cannot lie within any of these spheres. Therefore, if the unbounding spheres at X and Y overlap, the surface cannot intersect the line segment XY.

This observation allows speculatively advancing the ray by $\Delta t = k \cdot h(X)$ for $1 \leq k \leq 2$ as follows. Let the new point be $Y = X + \hat{\omega}\Delta t$. If $h(Y) \geq k \cdot h(X)$, then there was no intersection on XY and the speculative advancement was conservative and can be accepted. Otherwise the ray can still be advanced from X by $\Delta t = h(X)$. Keinert et al. call this over-relaxed sphere tracing (figure 4. Naive sphere tracing advances the ray with only $k = 1$, often discovering that the next point redundantly covers the interval just traversed.

Over-relaxation is obviously most useful when the distance estimator in *space* is overly conservative for distance to the surface along the *ray*. There is a tradeoff between setting k close to 2 to gain the largest possible step and setting k close to 1 to reduce the number of times the speculative advancement fails. Because success doubles the step distance and failure doubles the number of evaluations of distance estimator h per iteration, over-relaxed sphere tracing can double or halve the performance compared to naive sphere tracing. Keinert et al. observe that $k = 1.2$ gives a net 25% reduction in tracing time for the complex urban environment in figure 5 that appeared in the Mercury demoscene production *The Timeless* (`http://www.pouet.net/prod.php?which=62935`).

Figure 5: *The Timeless*

12.2 Bounding Spheres

When working with implicit surfaces, operations on distance estimators make it possible to build complex scenes by layering operations on simple primitives. Doing so offers great expressive power and programming elegance at a potentially high performance cost. This is because rays that pass nowhere near a surface must still evaluate the distance function for it.

A complicated set of operations on distance estimators inherently forms a **scene graph** tree of computation through function invocation. Performance can be increased by avoiding descending branches of that tree that do not affect the result. This kind of pruning is a common computer science operation and yields order-of-growth increases, unlike the kind of small constant factor that can be obtained by micro-optimizing within a single distance estimator.

The easiest way to prune the tree is to abort computation within a distance estimator when an early, inexpensive test determines that the ray is very far from the surface. A simple bounding sphere accomplishes this and correctly returns a still-conservative value. For example, let `expensiveDE` be a computationally-expensive distance estimator for an surface that is known to be entirely contained within the sphere with center `C` and radius `r`. A fast bounding estimator first tests how close the point X is to the bounding sphere. This must be a distance less than that to the true surface. It then only invokes the full, expensive estimator if X lies within the bounding sphere.

```
float boundingDE(Point3 X, Point3 C, float r) {
    float t = length(X - C) - r;
    if (t > 0) { return t; } else { return expensiveDE(X); }
}
```

Square root is one of the most expensive operations to perform on a GPU. We can slightly increase the performance of the slow case without affecting the fast branch by delaying the square root until the inside of the branch:

```
float boundingDE(Point3 X, Point3 C, float rSquared) {
    Point3 v = X - C;
    float tSquared = dot(v, v) - rSquared;
    if (tSquared > 0) {
        return sqrt(tSquared);
    } else {
        return expensiveDE(X);
    }
}
```

It is also a good idea to branch to the full, expensive distance estimator if X is even close to the sphere. That is because the silhouettes of objects are the most expensive to trace using a numerical root finder. A ray passing by in a direction tangent to the surface appears to be very close to the surface to the omni-directional distance estimator, so the ray marching iterator takes very small steps, even though the ray is not actually approaching the surface quickly. Introducing a false silhouette around each object through the bounding sphere can exacerbate this problem. We can separate the bounding radius from the test value to avoid this problem:

```
float boundingDE(Point3 X, Point3 C, float rSquared) {
    Point3 v = X - C;
    float tSquared = dot(v, v) - rSquared;
    if (tSquared > rSquared * 0.2) {
        return sqrt(tSquared);
    } else {
        return expensiveDE(X);
    }
}
```

12.3 Reintroducing Analytic Roots

Geometric planes are are large surfaces that can cover a large amount of the screen. A ray caster following any ray near the plane will be constrained to march in steps no larger than the elevation of the ray above the plane, which may be quite small compared to the true distance to the next intersection with the scene. We can therefore obtain a good speedup for scenes with ground planes by performing the (efficient) analytic intersection with the plane and then using the ray marcher only for other surfaces. Spheres are a border case. The distance estimator is extremely fast and yields good estimates along the ray on the interior and far from the sphere. Yet for rays that pass near the silhouette, an analytic solution is far faster than a ray marched one.

Figure 6 shows on the left a rendering of a scene composed of implicit surfaces. On the right is a visualization of the number of invocations of the scene distance estimator at each pixel, where brighter pixels are more expensive. This scene uses the analytic ground plane and bounding sphere optimizations.

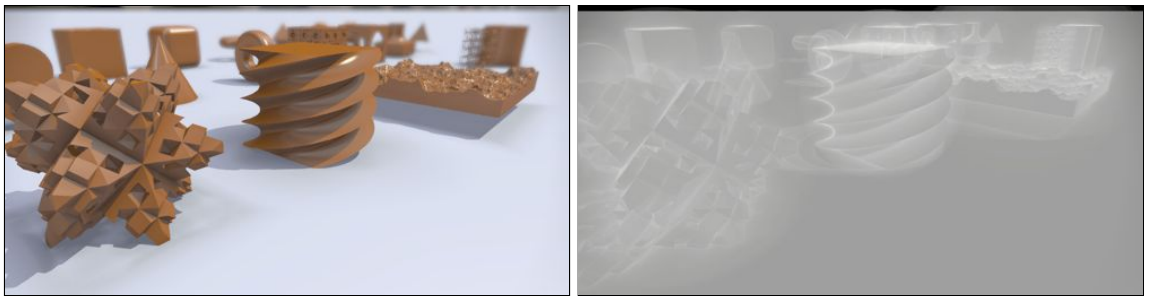


Figure 6: *Left: A reference scene of many different shapes defined by implicit surfaces. The distance estimator is used for primary rays, (soft) shadow rays, and ambient occlusion. Left: Visualization of the number of scene distance estimator invocations at each pixel; black = 0, white = 20.*

Note that in this figure, silhouettes and surfaces seen at glancing angles are far more expensive than the interiors of objects, which are almost as efficient to trace as the ground plane itself. The screw-like shape and the heightfield are expensive because they have very conservative distance estimators. The silhouettes of shadows are also expensive–those are places where the ray for the visibility query to the light passes close to a surface and thus slows down as it approaches.

12.4 Other Optimization Strategies

Soft shadows, ambient occlusion, and in some cases antialiased pixels and depth of field can all be approximated efficiently using distance estimators because they track how close a point is to all surfaces in the scene. Under analytic ray tracing, these instead must be estimated using tens or hundreds of rays per pixel.

Once objects are surrounded in bounding spheres, traditional spatial subdivision data structures can be applied to them. For example, it is simple to compute a bounding volume hierarchy on the bounding spheres. However, using a data structure on a GPU in graphics mode is a bit tricky because GLSL executes function invocation by inlining. Thus there is no true recursion for traversing a tree in a natural way. This also makes it hard to implement reflection and refraction at the same surface under Whitted ray tracing, since that creates a tree of rays. (Path tracing, photon tracing, and one of reflection or refraction without the other can all be implemented using loops.)

There are three solutions for the lack of recursion. One is to build a stack global memory using

OpenGL 4 image buffer operations. This has awkward syntax and can be slow. Another is to build a stack in local memory. This can be accomplished for recursive rays on recent hardware using code like the listing below. A similar explicit stack can be used to traverse data structures.

```
struct RayStackFrame {
   Point3 origin;
   Vector3 direction;
   Color3 weight;
};

RayStackFrame stack[MAX_STACK];
int stackTop = 0;

...
stack[0] = RayStackFrame(X, w, Color3(1, 1, 1));
while (stackTop >= 0) {
   Color3 weight = stack[stackTop].weight;
   ...
   if (reflection) {
      // cast a recursive ray by pushing onto the stack
      // and attenuating by the magnitude of the BSDF impulse
      ...
     ++stackTop;
   }
   L_o += weight * L_scattered;
}
```

Beware that local memory is significantly slower than register memory (where other local variables are stored), although significantly faster than global memory. Local memory is frequently implemented as a portion of the L1 cache. Also beware that on older hardware without local memory support, relative indexing into an array can compile to a chain of conditionals because there are no relative memory operations on arrays. Some register-only tree traversal strategies developed the introduction of local and global memory write operations are appropriate for older hardware and may find renewed value for performance even on newer targets [Horn et al. 2007; Popov et al. 2007].

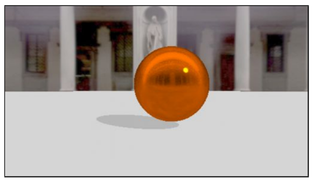
Analytic and numerical roots can be mixed within a scene, as we did for the ground plane. Our simple ray tracer produces depth buffer values compatible with OpenGL's triangle rasterization, so implicit surfaces may be mixed freely with all primary ray strategies.

13 Some Online Examples

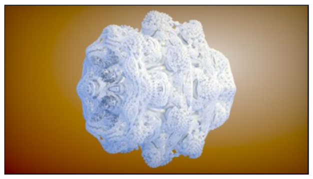
13.1 Educational

I wrote these three heavily-documented examples to demonstrate how to set up an analytic and a ray marching renderer. These are on the Shadertoy website, which uses WebGL, a limited version of the full OpenGL supported by G3D. So, the code is a little less efficient and clear than the best case, but has the advantage of running in most current web browsers.

Analytic Tracer
https://www.shadertoy.com/view/XdsGWS



Mandelbulb Explained
https://www.shadertoy.com/view/XsXXWS



Legos
https://www.shadertoy.com/view/Xdl3Dj

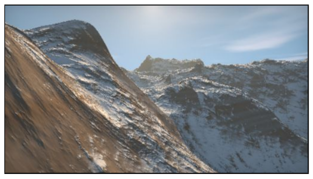
13.2 Impressive

These three examples are by Iñigo Quilez, a graphics professional who works at Pixar on feature films and has long contributed to the demoscene and online graphics community. These are three of the most impressive demos on the Shadertoy website and good (if a bit hard for the novice to parse) examples of how to use the implicit surface techniques described in this document.

Dolphin
https://www.shadertoy.com/view/4sS3zG

Mike
https://www.shadertoy.com/view/MsXGWr

Elevated
https://www.shadertoy.com/view/MdX3Rr



14 Further Reading

Implicit surfaces were first rendered using numerical root finding by Blinn [1982]. They've since been advanced significantly through research by many others. Gomes et al. give a good survey in their book [2009].

The academic graphics community has largely worked in parallel with the demoscene community, which has been ray tracing these in real time with similar techniques since at least the early 2000's, but which has only recently begun to document and share their methods widely. Quilez has been a leading voice in spreading information about demoscene techniques through his personal website http://www.iquilezles.org/, the Shadertoy website, and recently some public presentations [Quilez 2008] and online videos [Quilez 2014]. Swoboda's GDC 2012 presentation [2012] overviews the material from this document and then describes some of the performance issues on modern GPUs.

I've simplified some of the details of sphere tracing and distance estimators in this explanation. See Hart's SIGGRAPH course notes [1993] or Liktor's survey paper [2008] for more details on the mathematical constraints and ways of choosing optimal step sizes based on derivatives. See Keinert et al.'s [2014] paper for the current state of the art.

One of the most interesting applications of numerical methods for ray intersection is rendering fractals, which by definition have difficult surfaces to intersect (since they have unbounded area!) [Hart et al. 1989].

An alternative approach for rendering surfaces that do not admit analytic intersection solutions is to convert them to an approximation that does, for example, a triangle mesh. A popular method for doing so is Lorensen's and Cline's marching cubes algorithm [1987]. Conversely, it is often desirable to convert an existing triangle mesh into a signed distance function for integration with a ray marcher (e.g., to achieve warping and smooth blending). Akleman and Chen [1999] give one implementation of such an operation. Swoboda [2012] discusses both triangle-to-implicit conversion and reverse in the context of modern GPUs.

The particular form of implicit surface defined by distance from a point set is very popular for modeling fluids. Fedkiw worked extensively with these and the book that he coauthored is a good reference [2003]. There's an entire subfield of rendering called point-based graphics that directly ray traces such representations, sometimes using implicit surface methods.

References

AKLEMAN, E., AND CHEN, J. 1999. Generalized distance functions. In *1999 Shape Modeling International (SMI '99), 1-4 March 1999, Aizu, Japan*, 72–79. 19

BLINN, J. F. 1982. A generalization of algebraic surface drawing. *ACM Trans. Graph. 1*, 3 (July), 235–256. 18

GOMES, A., VOICULESCU, I., JORGE, J., WYVILL, B., AND GALBRAITH, C. 2009. *Implicit Curves and Surfaces: Mathematics, Data Structures and Algorithms*, 1st ed. Springer Publishing Company, Incorporated. 18

HART, J. C., SANDIN, D. J., AND KAUFFMAN, L. H. 1989. Ray tracing deterministic 3-d fractals. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '89, 289–296. 19

HART, J. C. 1993. Ray tracing implicit surfaces. *Siggraph 93 Course Notes: Design, Visualization and Animation of Implicit Surfaces*, 1–16. 18

HART, J. C. 1996. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer 12*, 10, 527–545. 5

HOFFMAN, C., AND HOPCROFT, J. 1985. Automatic surface generation in computer aided design. 92–100. http://www.cs.purdue.edu/homes/cmh/distribution/papers/Geometry/geo1.pdf. 13

HORN, D. R., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. 2007. Interactive k-d tree GPU raytracing. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '07, 167–174. 17

KEINERT, B., SCHÄFER, H., KORNDÖRFER, J., AND ANDMARC STAMMINGER, U. G. 2014. Enhanced sphere tracing. *Smart Tools & Apps for Graphics*. http://erleuchtet.org/~cupe/permanent/enhanced_sphere_tracing.pdf. 5, 14, 18

LIKTOR, G. 2008. Ray tracing implicit surfaces on the GPU. *Computer Graphics and Geometry Journal 10*, 3. http://www.cescg.org/CESCG-2008/papers/TUBudapest-Liktor-Gabor.pdf. 18

LORENSEN, W. E., AND CLINE, H. E. 1987. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '87, 163–169. 19

OSHER, S., AND FEDKIW, R. P. 2003. *Level set methods and dynamic implicit surfaces*. Applied mathematical science. Springer, New York, N.Y. 19

POPOV, S., GUNTHER, J., SEIDEL, H.-P., AND SLUSALLEK, P. 2007. Stackless kd-tree traversal for high performance gpu ray tracing. *Comput. Graph. Forum 26*, 3, 415–424. 17

QUILEZ, I. 2008. Rendering worlds with two triangles with raytracing on the GPU in 4096 bytes. *NVScene 2008* (August). http://www.iquilezles.org/www/material/nvscene2008/rwwtt.pdf. 18

QUILEZ, I. 2014. formulanimations tutorial :: the principles of painting with maths. https://www.youtube.com/watch?v=0ifChJ0nJfM. 18

SWOBODA, M. 2012. Advanced procedural rendering in DirectX 11. *GDC12*. http://directtovideo.wordpress.com/2012/03/15/get-my-slides-from-gdc2012/. 18, 19

Index