Ray Tracing

Joe Fowler

Ray tracing is a widely-used technique for rendering mathematically defined geometry with a high degree of realism. Although the process requires far more processing than alternatives such as z-buffering with a simple perspective projection, it is highly parallelisable and also produces far more visually stunning images which makes it ideal for situations where the visualisation is not required to run in real-time and can instead be preprocessed, such as animated movies. The system also allows for more complicated shapes than are usually available in rendering libraries, such as true spheres and direct rendering of *Constructive Solid Geometry (CSG)*. In this essay I intend to cover the fundamental aspects of ray tracing, with specific focus on the features that are not easily reproduced with other techniques.

The General Algorithm

Unlike other rasterisation methods which determine visible surface information on a *per-scanline* or *per-polygon* basis, ray tracing determines this information on a *per-pixel* basis. The general algorithm can be described with the following pseudo-code:

```
function raytrace(Scene scene, unsigned int width, unsigned int height) {
    Image output(width, height);

    for (unsigned int y = 0; y < height; y++) {
        for (unsigned int x = 0; x < width; x++) {
            Generate ray from camera passing through pixel x, y;
            Intersect the ray with the scene, finding the closest intersection;
            Calculate the colour of light leaving this intersection point
            that leaves towards the camera, and assign it to output pixel (x, y);
        }
    }

    return output;
}
```

The contents of the inner loop can further be broken down into two important sections; intersection determination and rendering. Due to the high number of times that each of these sections is evaluated, it is highly beneficial to find algorithmically optimal solutions to each.

Intersecting Shapes

Ray tracing allows for reasonably complicated shapes to be directly rendered as objects. Theoretically, any shape that can be mathematically described in the form $f(x, y, z) = 0$ can be ray traced by substituting the definition of the ray $r(t) = \underline{o} + t\underline{d}$ into the function, giving $f'(t) = 0$ and thereby reducing the problem to a one-dimensional *root-finding problem*, which can often be numerically approximated to provide the value of t at which the intersection occurs, which can then be substituted back into the ray definition to give the point of intersection. However, finding the normal of such a surface is often more complicated so I will instead focus on specific shapes, or *primitives*, which can be very easily intersected.

Two of the simplest shapes to intersect are *planes* and *spheres*. Here I will describe the ray intersection test for a sphere as it is quite a neat derivation and also produces a more interesting outcome than a plane, which can so easily be reproduced with any other rendering technique.

To begin with, let's restate the definition of a ray, and also state the definition of a sphere in the three-dimensional vector space that the scene resides in; any intersection with the sphere must satisfy both:

$$\vec{v} = \vec{a} + t\,\vec{d}$$

$$(\vec{v} - \vec{c}) \cdot (\vec{v} - \vec{c}) = r^2$$

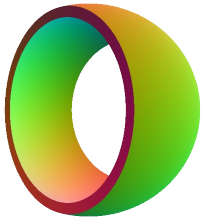
Given both of these definitions, one can substitute the former into the latter to give:

$$(\vec{a} + t\,\vec{d} - \vec{c}) \cdot (\vec{a} + t\,\vec{d} - \vec{c}) = r^2$$

$$\Rightarrow((\vec{a}-\vec{c})+t\vec{d})\cdot((\vec{a}-\vec{c})+t\vec{d})=r^2$$
$$\Rightarrow(\vec{a}-\vec{c})\cdot(\vec{a}-\vec{c})+2t\cdot(\vec{d}\cdot(\vec{a}-\vec{c}))+t^2\cdot(\vec{d}\cdot\vec{d})=r^2$$
$$\Rightarrow A=\vec{d}\cdot\vec{d}\,,B=2\cdot(\vec{d}\cdot(\vec{a}-\vec{c})),C=(\vec{a}-\vec{c})\cdot(\vec{a}-\vec{c})-r^2$$

Which then allows us to solve for t using the quadratic formula. When the determinant is negative, the ray has not intersected the sphere, when it is 0 the ray is tangential to the sphere, and when it is positive the ray has intersected the sphere and therefore has both an entry and exit intersection point. If the values of t are both negative, the intersection occurred with the ray behind the camera and so should not be counted. Since we want the earliest intersection forward from the camera, we want to select the smallest of the two values of t that is non-negative.

Constructive Solid Geometry



CSG allows for more complicated shapes to be defined by expressing them as a combination of more primitive components, and combining them using familiar union, intersection, and complement operations. The resultant shapes are often much more aesthetically pleasing (and frequently require less intersection tests) than attempting to produce the same shape using a combination of triangles. To the right is an example of a shape constructed using CSG. The shape is defined as:

$$Sphere((0,0,0),3)\cap\neg Sphere((0,0,0),2.75)\cap\neg Sphere((-10,0,0),9)\cap\neg Sphere((10,0,0),9)$$

The rendering function used here simply maps the surface normal to a colour.

In order to generate these geometric shapes, an alteration must be made to the intersection testing code so that instead of finding only the earliest intersection, we instead find *all* intersections between the ray and the shape, and additionally note whether the intersection is an *entry* or *exit*. The resultant list of entry and exit points will give us a collection of intervals that tell us when the ray is inside the shape, which in turn give us a subset of the real numbers S where $x\in S\Leftrightarrow shape\ contains\ x$

In order to perform a ray-intersection test with the CSG-intersection of two shapes A and B, we must first perform ray-intersection tests separately for A and B and then take the set-intersection of the result, giving us the ray-intersection with the CSG-intersection of A and B. This process can be expressed in the following pseudocode:

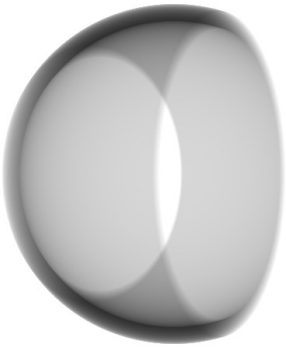
```
function intersectCSGIntersect(Shape A, Shape B, Ray ray) {
    Array<Intersection> intersectA = A.trace(ray);
    // Intersection of any set with the empty set is the empty set.
    if (intersectA.length == 0) return Array<Intersection>();
    Array<Intersection> intersectB = B.trace(ray);
    // Merge the two arrays together, sorted by distance min->max.
    Array<Intersection> merged = merge(intersectA, intersectB);
    Array<Intersection> out;
    int counter = 0;
    if (A.contains(ray.start)) counter++;
    if (B.contains(ray.start)) counter++;
    for (unsigned int i = 0; i < merged.length; i++) {
        if (counter == 2) out.push(merged[i]);
        if (merged[i].isEnteringShape) counter++;
        else counter--;
        if (counter == 2) out.push(merged[i]);
    }
}
```

CSG-union implementation is identical except the check is for *counter == 0* instead, and there is no early exit check. The implementation for CSG-complement, which trivially iterates through each intersection toggling entry with exit and negating surface normals, is also easily produced.

CSG not only allows us to describe more complicated shapes, but also enables us to render *volumetric effects* such as clouds of smoke or refractions and internal reflections that are often very hard to replicate with shapes that have been modelled as hollow triangle meshes. For example, a simple approximation of smoke can be produced by performing an intersection test with a CSG object and summing

the lengths of the intervals during which the ray was inside the volume. An example algorithm and screenshot from my own ray tracer are provided below:

```
function generateShadingFromTrace(Array<TraceRes> trace) {
    real dist = 0;
    // The intersections in trace will toggle between entry and exit.
    if (trace.length > 0) {
        unsigned int start = trace[0].isEnteringShape ? 1 : 2;
        for (unsigned int i = start; i < trace.length; i += 2) {
            // Add the distance that the ray was inside the shape for.
            dist += trace[i].distance - trace[i - 1].distance;
        }
        // If any of these is true, the last intersection is an entry.
        // This means the ray travels inside the volume forever.
        if ((trace[0].isEnteringShape && trace.length is odd)
            || (!trace[0].isEnteringShape && trace.length is even))
                dist = Infinity;
    }
    // 1 is white, 0 is black.
    return pow(0.5, dist);
}
```



Intersecting Scenes

Using the methods described above for rendering CSG, we can model any scene by taking the union of every shape it contains. However, when we come to render the scene, we usually only need the closest intersection instead of all of them; secondary rays tend to branch off in different directions from the first point instead of continuing along the same path. Significant savings can be made by treating the scene differently from how we would treat it if it were a single shape. Below is outlined a naïve algorithm for finding the closest intersection:

```
function intersect(Ray ray, Scene scene) {
    Intersection closest = infinitelyFarAwayIntersection;
    for each (Shape shape in scene) {
        Intersection temp = shape.intersect(ray);
        if (intersection occurred && temp.distance < closest.distance) {
            closest = temp;
        }
    }
    return closest;
}
```

Presented above is a naïve algorithm for finding the closest intersection. Unless the rendering section of the program is exceptionally primitive and does not produce any child rays whilst generating the final colour for the pixel, it is very likely that the time spent performing ray-scene intersection tests will significantly outweigh the time spent performing the actual colour calculations. It is therefore highly beneficial to reduce the complexity of the intersection function as much as possible. By utilising a carefully implemented *spatial partitioning system*, the expected asymptotic complexity of the intersection test can be appreciably reduced.

A specific example of one such structure would be the *k-d tree*, applied to 3 dimensions. With very recognisable ties to regular *binary trees*, the k-d tree divides the scene space recursively by splitting it into two parts, separated by an *axis-aligned* plane. At each level of the tree, the plane's normal rotates to another axis. By carefully selecting splitting planes that balance the number of shapes in each half whilst attempting to avoid having shapes that intersect the splitting plane, the expected cost of ray traversal can be reduced dramatically. In the case where shapes do intersect the splitting plane, the shape must be included in both branches to avoid false negatives, so avoiding this situation is beneficial. Traversal of the tree involves recursively selecting the half-space which the ray intersects first, until a leaf node is reached and the intersection tests can be performed on the primitives it contains. If the ray does not intersect with any shape in this leaf node, then it must move on to the next leaf node. Discovery of the first leaf node in a balanced k-

d tree has O(log n) time complexity, and therefore provides us with an efficient method of testing for intersections with the most likely shapes first. Whilst this is clearly a massive improvement for ray tracing static scenes, the k-d tree cannot easily be rebalanced after shapes have been inserted or deleted, making it a less attractive solution if one requires the ability to vastly modify the scene on-the-fly.

The Rendering Equation

Intersection tests are only half the story when it comes to ray tracing. Once the *primary ray* for the pixel has found its intersection, the next step is to determine the colour which should be emitted from it towards the camera. Both of the example images displayed so far have had very simple functions to perform this step – the functions have only derived their results from the array of intersection points provided by the initial intersection test. Usually this form of ray tracing is referred to as *ray casting* (not to be confused with the alternative meaning of ray casting, which is the actual intersection determination process), as it never results in secondary rays being traced. To achieve more complicated effects such as shadows and refraction, secondary (and potentially terniary and upwards) rays must be generated spreading out from the intersection point in order to determine how much light has arrived at this point. Many different techniques can be used to perform this step; *ambient occlusion*, *radiosity*, and a *bidirectional* path tracing algorithm such as *Metropolis Light Transport (MLT)* are a few examples of aesthetically pleasing choices. Each process attempts to model (in the case of ambient occlusion, quite poorly) the behaviour of light in the hope of achieving *photorealistic* visualisations. This problem effectively reduces to finding an efficient way of approximating the *rendering equation*:

$$L_o(\vec{x}, \vec{\omega}_o, \lambda, t) = L_e(\vec{x}, \vec{\omega}_o, \lambda, t) + \int_{\Omega} f_r(\vec{x}, \vec{\omega}_i, \vec{\omega}_o, \lambda, t) L_i(x, \vec{\omega}_i, \lambda, t)(\vec{\omega}_i \cdot \vec{n}) \; d\vec{\omega}_i$$

where (in order of appearance):

- L_o is the *total spectral radiance* of wavelength λ leaving position $\vec{x}$ in direction $\vec{\omega}_o$ at time t
- $\vec{x}$ is the position of the light source concerned (ie. the intersection point).
- $\vec{\omega}_o$ is the direction of the light leaving the position.
- λ is the wavelength of the light leaving the position.
- t is time. This is usually constant, but by integrating over an interval of t we can render motion blur.
- L_e is the *emitted spectral radiance*.
- Ω is the unit hemisphere centred around $\vec{x}$ and pointing in direction $\vec{n}$
- f_r is the *Bidirectional Reflectance Distribution Function (BRDF)* of the surface.
- $\vec{\omega}_i$ is the **negative** direction of the incoming light at position $\vec{x}$
- L_i is the *incoming spectral radiance*.
- $\vec{n}$ is the normal of the surface at position $\vec{x}$

This equation is an accurate model of light constructed so that it obeys *conservation of energy*, and a large number of rendering algorithms are attempts to approximate the solution to this equation. At the highest level of abstraction, the rendering equation models the outgoing light as the sum of the light emitted and the light reflected. Although this overlooks effects such as *transmission* (light passing through the material, such as with glass) and *subsurface scattering* (a more specific case of transmission where light is scattered inside a volume, such as the red glow visible when a bright light is pointed at fingers or ears) unless adapted, it produces realistic renderings when the objects concerned are supposed to be opaque.