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:

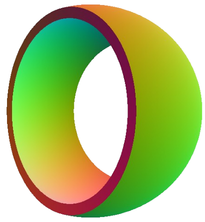
$$\vec{v} = \vec{a} + t\vec{d}$$
$$(\vec{v} - \vec{c}) \cdot (\vec{v} - \vec{c}) = r^2$$

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].isEnteringIntersection()) 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.

Intersecting Scenes

Now that we have a way of intersecting one shape, we need a way of finding the earliest intersection in a set of more than one shape so that we can render a scene.

```
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 is reduced from O(n) to O(log n), where n represents the number of shapes in the scene. Whilst this is clearly a massive improvement for ray tracing static scenes, the k-d tree cannot easily be rebalanced after insertions or deletions, making it a less attractive solution if one requires the ability to modify or update the scene on-the-fly.