Sean Higgins

March 6, 2017

CS 500 – Ray Tracing

Prof. Herron

CS 500 Ray Tracing – Proj. 1 Report

Note: as .hdr files cannot be included into a Word document, I will be including them in the provided .ZIP file

Overview:

After numerous setbacks and challenges, I was finally able to properly set up and implement the beginnings of the ray tracer required throughout the rest of the class. The primary challenges I encountered were primarily fine-tuning my ray intersections with bounding volumes (for the Eigen-provided KD Tree) as well as properly understanding some of the finer points of how the various intervals work with regards to the various intersection algorithms.

Overall, the project contains the necessary aspects required to read in a variety of basic shapes (triangles, spheres, cylinders, and axis-aligned boxes), to be able to intersect rays with said objects to provide various data to be used in future lighting calculations, such as the point of intersection, which shape has been intersected, and the normal at the point of intersection.

Shape Representations:

Sphere: center point, radius—very straightforward. I also precalculate the square of the radius in an attempt to speed things up

Triangle: three points, two edges calculated between two of those points. I also allow for the additional specification of a normal at each point, to allow for a more accurate calculation of the normal at the an intersection point

Axis-Aligned Box: a corner point and a diagonal, extended into three Slabs (two points and a normal, to represent two parallel planes in space) for intersection purposes

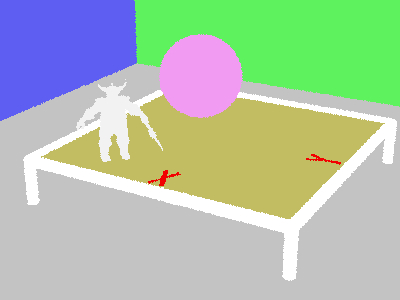
Cylinder: a center point for one of the bases, an axis to the other base, and a radius. I calculate another slab to use for intersections, as well as the square of the radius and some quaternions to rotate the cylinder to and from the Z-axis, to make life easier to intersect with it

KD Tree: To summarize, the KD Tree uses bounding boxes for all of the provided objects and intersects rays with them, then orders the shapes based on those intersections before having the ray go through the actual intersection calculations. The overhead tends to be less effective when the number of shapes is very small, especially when there are no meshes of triangles involved. However, when the medium or larger sized meshes start being used, the speedup is very, very significant, by a factor of ten or more.

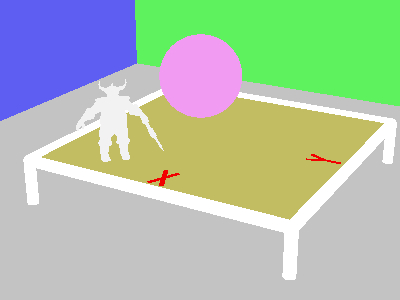
The largest difficulty I was running into with regards to this aspect was accurately intersecting my rays with the bounding boxes due to odd floating-point precision errors. For example, one of my bounding boxes would have X, Y, and Z intervals as such: [-3, 15], [-3, 15], [-0.1, 0]. My intersection point with that object, however, would be something along the lines of (-2.95, -2.95, 2.789 x 10^-7). Now, looking at the numbers, it is obvious that this point is close enough to be considered within the box—the X and Y values are within the required interval, and the Z value is very, very close to zero. The built-in intersections, however, are not smart enough to realize this, and thus return ‘no intersection’ when there really should be one. This resulted in some minor slicing of some of my spheres, as well as various few-pixel gaps in my meshes / triangles.

To solve this, I simply scaled the bounding boxes up slightly for both spheres and triangles, as they were the only offending objects. After calculating the bounding box properly for each of these objects, I took the maximal and minimal corners, subtracted those points to get the vector between them, and added and subtracted that vector to those same maximal and minimal points. I then forced the bounding box to include those points, and after some fine-tuning of the scale of that diagonal vector, I was able to successfully fix the slicing and gap issues. To give a more succinct explanation, I liken it to extending or enlarging a window on an operating system—you grab one (or both) corners of the window, and pull and extend them out in a diagonal to increase the size of the window and cover more area. My method is similar in principle, just extended to 3D.

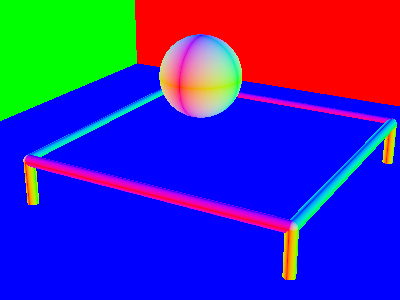
Object Kd (inherent color) values - With Tree



Object Kd (inherent color) values - Without tree



Object normals (absolute value)



Object distances (t-values) from camera, scaled to look nice

