CPSC 290 Final Report: Raytracer Modifications

Wendy Chen
CPSC 290, Fall 2015
Advising Professor: Holly Rushmeier

Introduction

The focus of this CPSC 290 project was to add further modifications to a raytracer base, namely, to implement an acceleration structure. I chose to implement a hierarchal axis-aligned bounding box system, partitioning space using a k-d tree.

The raytracer base I started with was taken from Tom MacWright's open source literate raytracer and modified by Professor Rushmeier (**References, 1**). This raytracer was able to handle simple scenes composed of spheres and triangles, area light sources, and lambertian and diffuse shading. In a homework assignment for CPSC 478, I further modified the raytracer to handle more light sources and specular shading for shiny, metallic materials. For my CPSC 290 project, in order to build geometrically interesting scenes to test my accelerated raytracer on, I also wrote a Python parsing script to import OBJ files into my raytracer. Additionally, I implemented ray refraction in order to support rendering glass materials.

Implementation

The unaccelerated raytracer shoots a ray through each pixel and iterates through all objects in the scene, checking for the closest object intersected by the ray. However, each pixel does not necessarily contain an object, and complicated scenes can contain thousands of objects, so the process of iterating through all objects per pixel becomes slow very quickly. To accelerate the process, we would like to reduce how many intersections between the ray and objects we check for at each pixel. By subdividing the space into bounding boxes, we can assign an appropriate list of objects found in each bounding box. Thus, for each ray shot through each pixel, we first traverse the bounding boxes and check if they contain objects. If the bounding box intersected by the ray doesn't contain any objects, we can quickly move on to the next pixel. If the bounding box does contain objects, we can check the list of objects contained in the bounding box instead of the list of objects contained in the entire scene.

To accelerate the raytracer, I implemented a k-d tree to subdivide the space into bounding boxes. A k-d tree is a special type of binary tree used for partitioning k-dimensional space (in this case, 3D space). Each node of my k-d tree contained the bounds of the box, stored as two minimum and maximum (x, y, z) triples. Each node also contained a list of scene objects that could be found in each box, a split point and a split axis, the node's depth, and pointers to the left and right children.

To populate the tree, each non-leaf node generates a split point by averaging its minimum and maximum bounds. The longest axis of each bounding box is chosen as its split axis for generating its left and right child. Points to the left of the split axis cascade into the left-child bounding box, and points to the right of the split axis cascade into the

right-child bounding box. Subdivisions stop when a bounding box either contains less than ten objects or when the node has reached a maximum depth of 50.

At each split, the code needs to determine what objects are now contained in the newly made bounding boxes. To determine whether a sphere object should belong to a bounding box, my code calculates an intersection by approximating the bounding box as a sphere, and then comparing the distance between the box's center and the sphere object's center with the sum of radii. To determine whether a triangle object should belong to a bounding box, I used code refactored from Tomas Akenine-Möller's publication on triangle-box overlap tests (**References, 2**).

Thus, before rendering, my raytracer uses the scheme described above to preprocess the scene's geometry and construct a k-d tree.

Before modification, the code that checked for scene intersections would take a ray and a list of objects in the scene as input, check for intersections between objects and the ray, and return the closest object with its distance. To accelerate the code, I built helper functions that allowed a shorter list of objects to be generated as input by traversing the k-d tree. The modified code that checks for scene intersections traverses the k-d tree and checks for intersections at each box. If the ray hits a bounding box, then the objects contained within the box are added to the list of objects to check for intersections with. After all boxes along the ray's path are checked and the list of objects is generated, then the code checks for intersections between the ray and the shorter list of objects.

To check for intersections between a ray and a bounding box, I refactored code from ScratchPixel's ray-box intersection implementation (**References, 3**). If there is a hit, then the list of objects contained at the node are simply copied into a running list of objects to check. After traversing the tree, this list of objects is then passed on to the original function that checks for intersections between rays and objects.

While implementing an acceleration structure was the main focus of this project, I also decided to implement ray refraction because glass materials create interesting light paths and are pretty to look at. I calculated the refraction vector of a ray by following the geometric construction in a lecture posted by Ohio State's CSE 681 (**References, 4**), and I calculated the color contribution of the reflection and refraction rays by following ScratchPixel's algorithm (**References, 5**). Thus, my raytracer is able to create the refracted reflections one would expect to find in glass materials. However, my modified code does not account for caustics, so the shadows remain opaque.

To import complex geometries, I wrote a Python script to parse OBJ files into an array of vertices and an array of triangle indices. Because various OBJ's have different formats, I first imported OBJ files into Blender, and then re-exported the OBJ files in order to get a consistent file format across all my test scenes. I packaged the results of the Python parser into Javascript files that I then loaded into my scenes. The OBJ files I have used in this project were taken from Florida State University and Stanford University (**References, 6, 7**).

Speculative Times

- k-d tree build time should be $O(n \log n)$, where n represents number of objects in the scene. Since subdividing the space and putting objects into bounding boxes is

essentially a sort, the predicted time should be the similar to a merge sort or the like.

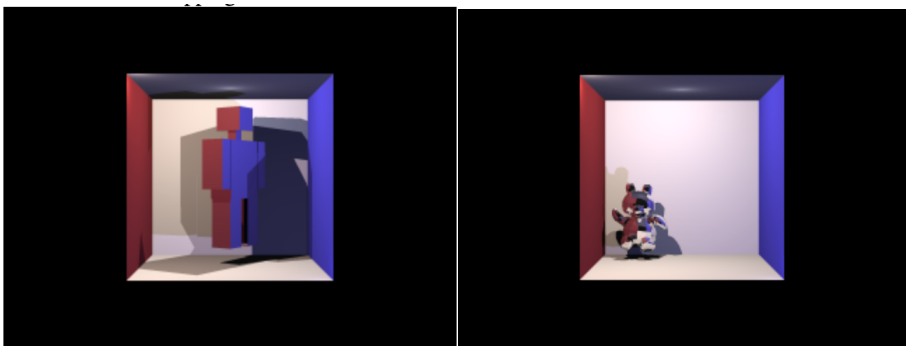
- Unaccelerated render time should be *O(mn)*, where *m* represents number of pixels in the image, and *n* represents number of objects in the scene. Since the code has to iterate through the entire list of objects in the scene per pixel, the predicted time should be roughly quadratic.
- Accelerated render times should be *O(mdt)*, where *m* represents the nmber of pixels in the image, *d* represents the maximum number of steps to traverse the k-d tree, and *t* represents the maximum number of objects at each leaf node. The predicted times only factor in caps for *d* and *t* since implementation of a k-d tree was constrained by a maximum depth and a threshold for number of objects. However, the accelerated time *O(mdt)* will still be faster than *O(mn)* because while *n* has no bound, *d* and *t* are bound to a small number by my program.

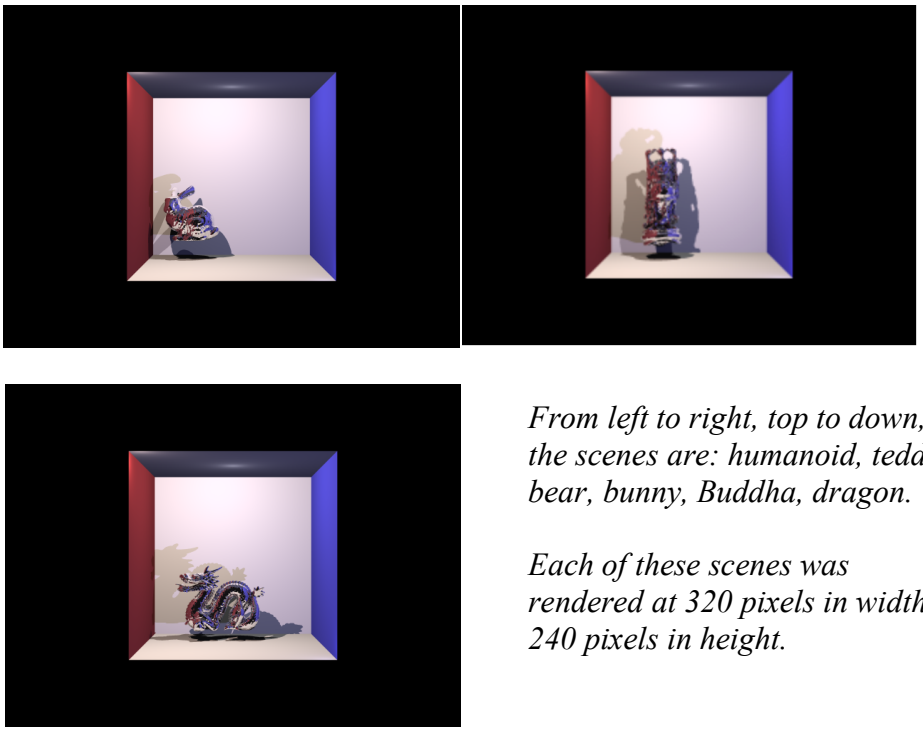
Results

In order to determine if my modified code actually delivered acceleration, I recorded the render times of the accelerated raytracer and the original unaccelerated raytracer for five sample scenes. In the case of the unaccelerated raytracer, I extrapolated three of the render times from the first two, because the unaccelerated raytracer should slow down linearly in proportion to how much geometry is in the scene. Originally, I tried to record the actual unaccelerated render times for the bunny, dragon, and Buddha scenes, but after eight hours, I killed the renders.

Below are figures showing the five sample scenes, and tables containing information about render times:

Figure 1. Five Sample Scenes





From left to right, top to down, the scenes are: humanoid, teddy bear, bunny, Buddha, dragon.

Each of these scenes was rendered at 320 pixels in width, 240 pixels in height.

Figure 2. Accelerated Render Times

Scene	Max depth	# Objects	k-d tree time (milliseconds)	Render time (milliseconds)	Machine used
humanoid	50	106	95	37111	my laptop, chrome
teddy bear	50	3203	1689	39214	my laptop, chrome
bunny	50	69676	4626	49899	my laptop, chrome
buddha	50	100010	12248	146700	my laptop, chrome
dragon	50	100016	10877	139151	my laptop, chrome

Figure 3. Unaccelerated Render Times

Scene	# objects	Render time (milliseconds)	Machine used
humanoid	106	79000	my laptop, chrome
teddy bear	3203	2180710	my laptop, chrome
bunny	69676	51928339.62	LINEAR EXTRAPOLATION
buddha	100010	74535754.72	LINEAR EXTRAPOLATION
dragon	100016	74540226.42	LINEAR EXTRAPOLATION

Figure 4. Observed Speed-up for Entire Scene

Scene	Ratio (unaccelerated time: accelerated time)
humanoid	2.128748888
teddy bear	55.61049625
bunny	1040.668944
buddha	508.0828542
dragon	535.6786974

Figure 5. Observed Speed-up per Object

Scene	# objects	(Unaccelerated/Accelerated)/#objects
humanoid	106	0.020082537
teddy bear	3203	0.017362003
bunny	69676	0.014935831
buddha	100010	0.005080321
dragon	100016	0.00535593

Figure 6. Graph of k-d tree build times from *Figure 2*

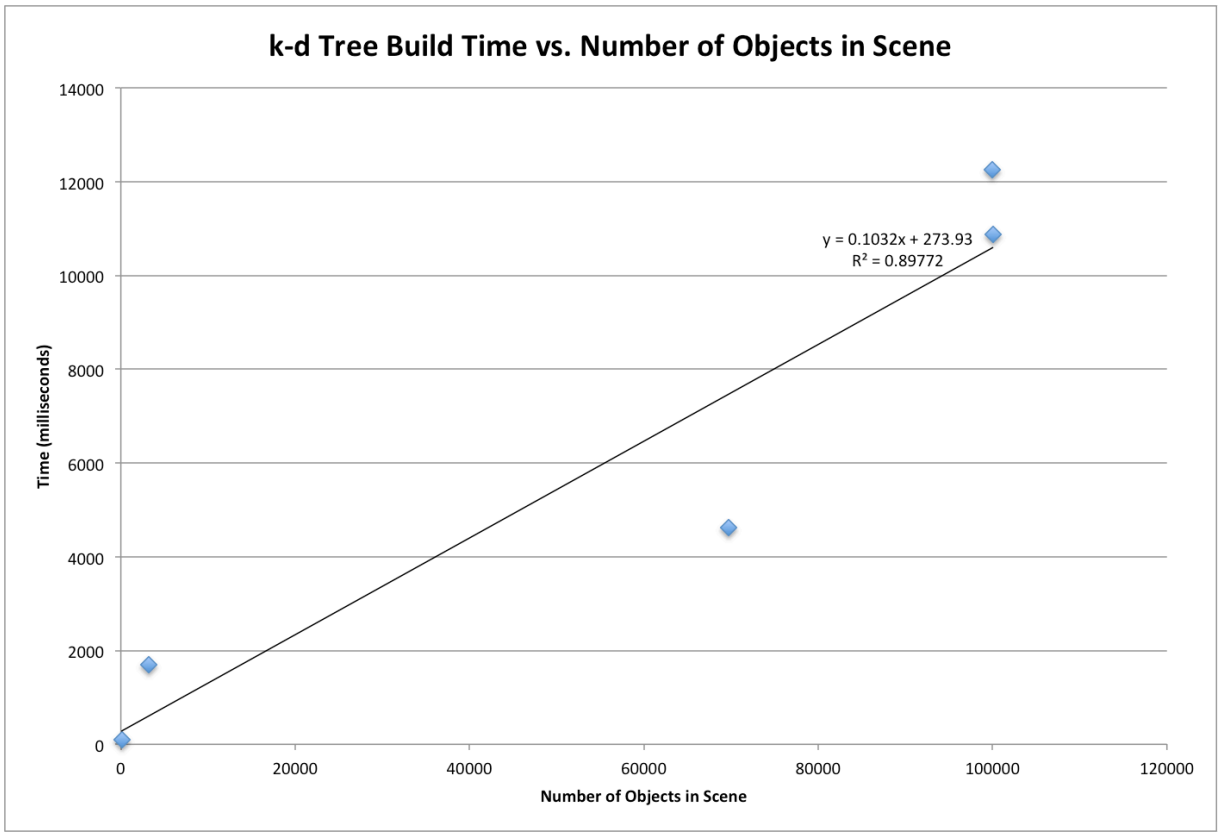


Figure 7. Graph of Accelerated Render Times from *Figure 2*

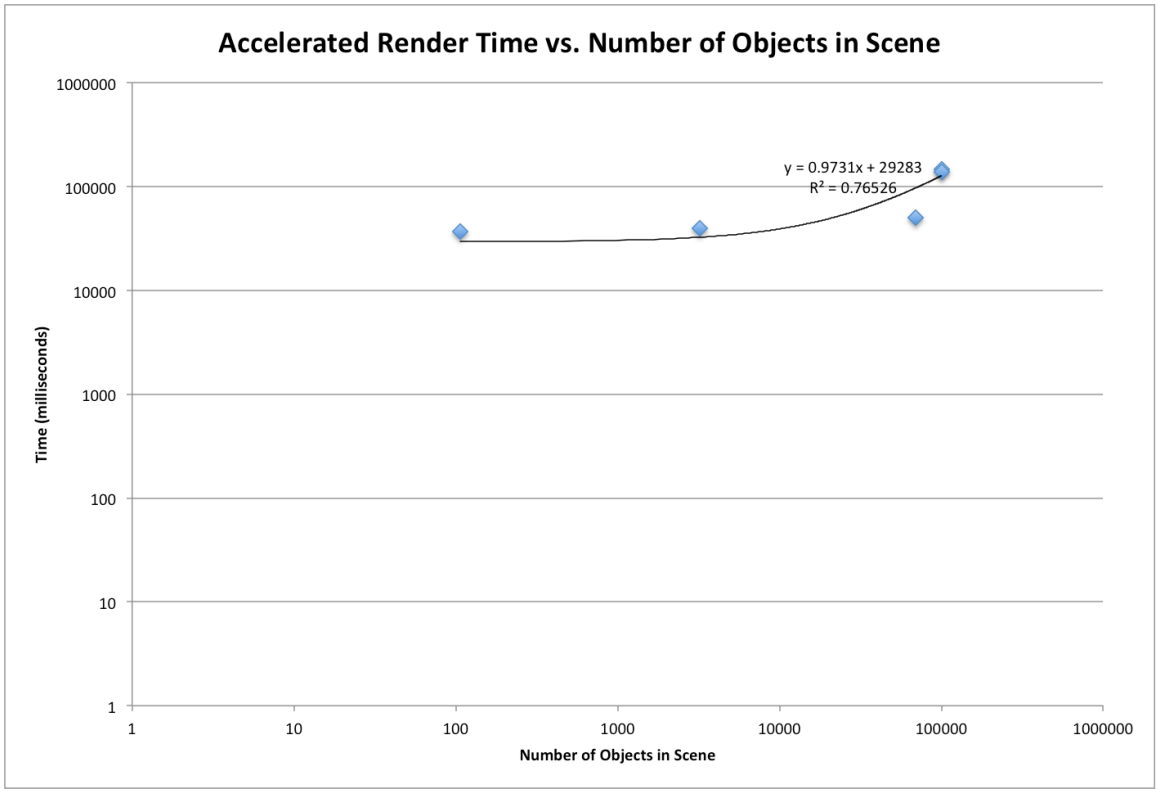


Figure 8. Graph of Unaccelerated Render Times from *Figure 3*

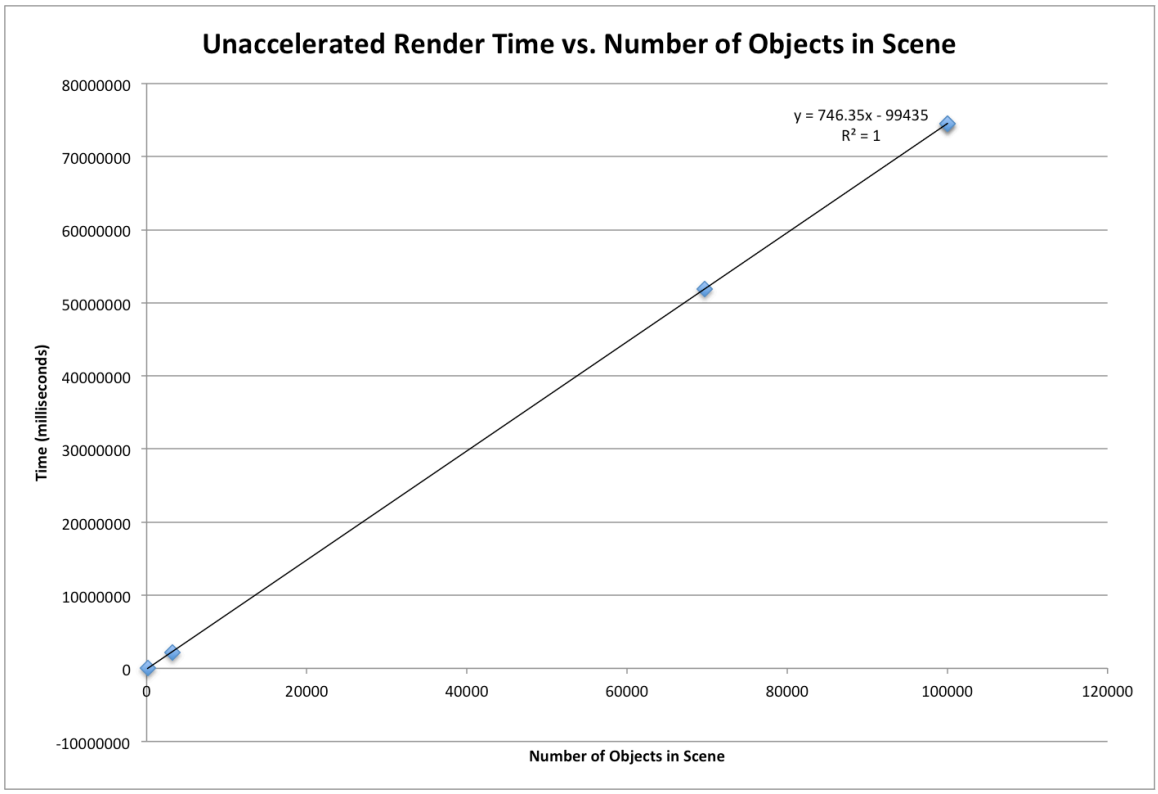
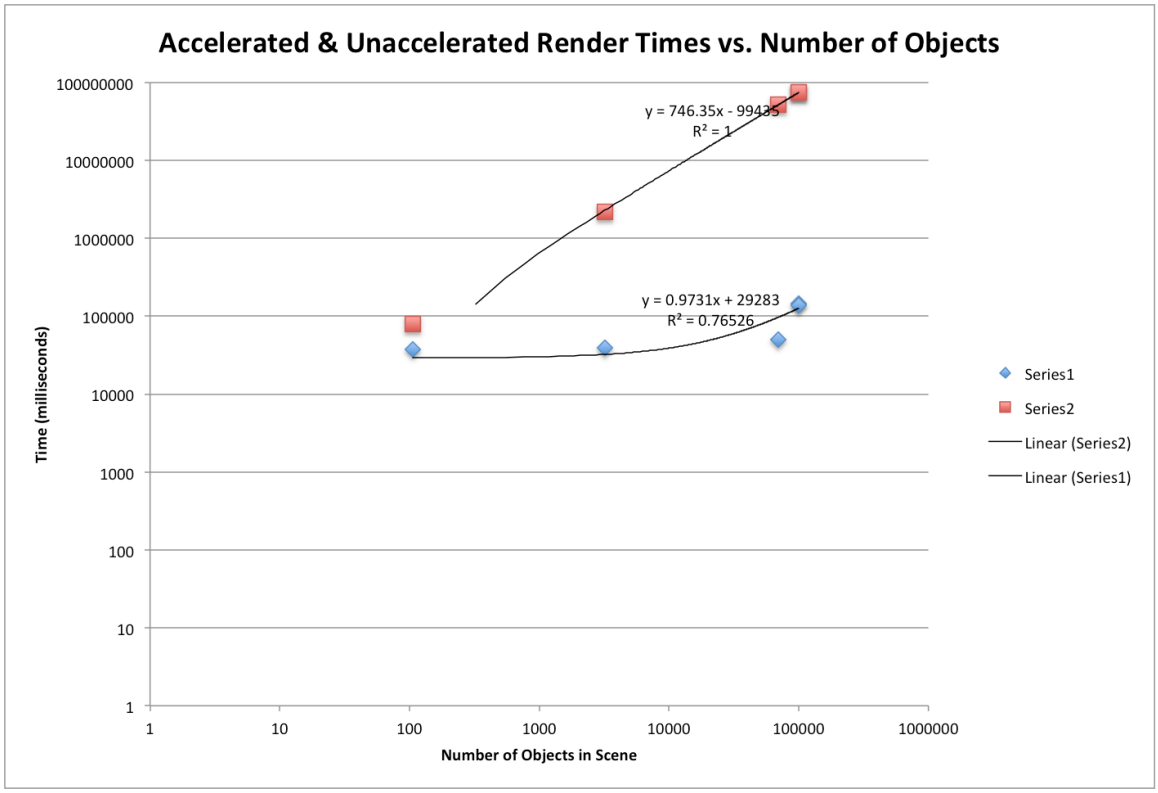


Figure 9. Graph of Accelerated & Unaccelerated Render Times



As evidenced by the graphs, the acceleration structure did work and provided significant speedups for complicated scenes. However, the speedup factor falls off when rendering large scenes like the dragon or Buddha, which both have about 100,000 triangles each. The fall off in speedup is most likely due to the constraints placed on the size of the k-d trees.

Difficulties & Further Work

The main difficulties I encountered were:

- Splitting: To determine the split point for creating two children bounding boxes from one parent, I naively averaged the minimum and maximum bounds of the parent box. However, a more efficient way to split the boxes would be to find the average midpoint of all the points contained within the box. Finding this average midpoint is difficult due to the problem of picking which points to sample. For example, if an object is only partially in the box, how should the points on the part of the object inside of the box be selected for midpoint averaging? My original approach was to assign a midpoint to each object in the scene, and take the average of the midpoints of the objects that intersected each box. This averaging approach fails if the assigned midpoint actually lies outside of the box. For further work, I would write a smarter way to take the average of midpoints. One idea is to assign a midpoint to each object in the scene, but throw out the midpoint if it lies outside of the bounding box even while the object intersects the bounding

box. If all midpoints are thrown out, then naively return the average of the minimum and maximum bounds of the bounding box.

- Thresholds: To determine when the k-d tree should cease recursively populating itself, I originally set a threshold and a maximum depth. If the length of the list of objects stored at a child node went below the threshold, or if the child's depth hit the maximum depth, then the node would cease subdividing. I set a maximum depth in order to prevent a stack overflow for memory; however, I realize that this constraint can reduce the amount of acceleration for geometry-heavy scenes. For further work, I would replace the maximum depth constraint with a different condition, and I would rewrite the k-d tree population to be non-recursive.
- Glass: To determine the color at each pixel of glass, I created a reflection and refraction ray for each eye ray that intersected a glass object. However, the shadows do not display any refracted light or caustics because I did not modify the way the code rendered shadows. For further work, I would add another function that checks for intersections between rays and objects, but instead of checking for merely intersection, I would check if the intersected ray could pass through the object or not. Additionally, I had some instances where the refraction ray color would return as NaN, inexplicably. I could not find this bug, so I have temporarily patched it to return the color black instead.

References

1. The original literate raytracer (http://www.macwright.org/literate-raytracer/) was modified by Professor Rushmeier for CPSC 478 Assignment 6. I used this modified raytracer as my starting base.
2. This site (http://fileadmin.cs.lth.se/cs/Personal/Tomas_Akenine-Moller/code/) contains both the paper and the original C code for triangle-box intersection. I refactored this code for my Javascript program.
3. I refactored the code from this site (http://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-box-intersection) for ray-box intersection.
4. I followed the calculations described here (http://web.cse.ohio-state.edu/~hwshen/681/Site/Slides_files/reflection_refraction.pdf) for determining the refraction ray.
5. I followed the algorithm here (http://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/ray-tracing-practical-example) for tracing both reflection and refraction rays for glass materials.
6. I found the humanoid and teddy bear OBJ files from this site: http://people.sc.fsu.edu/~jburkardt/data/obj/obj.html
7. I found the bunny, dragon, and Buddha OBJ files from this site: https://www.d.umn.edu/~ddunham/cs5721f07/schedule/resources/lab_opengl07.html

Code Modifications

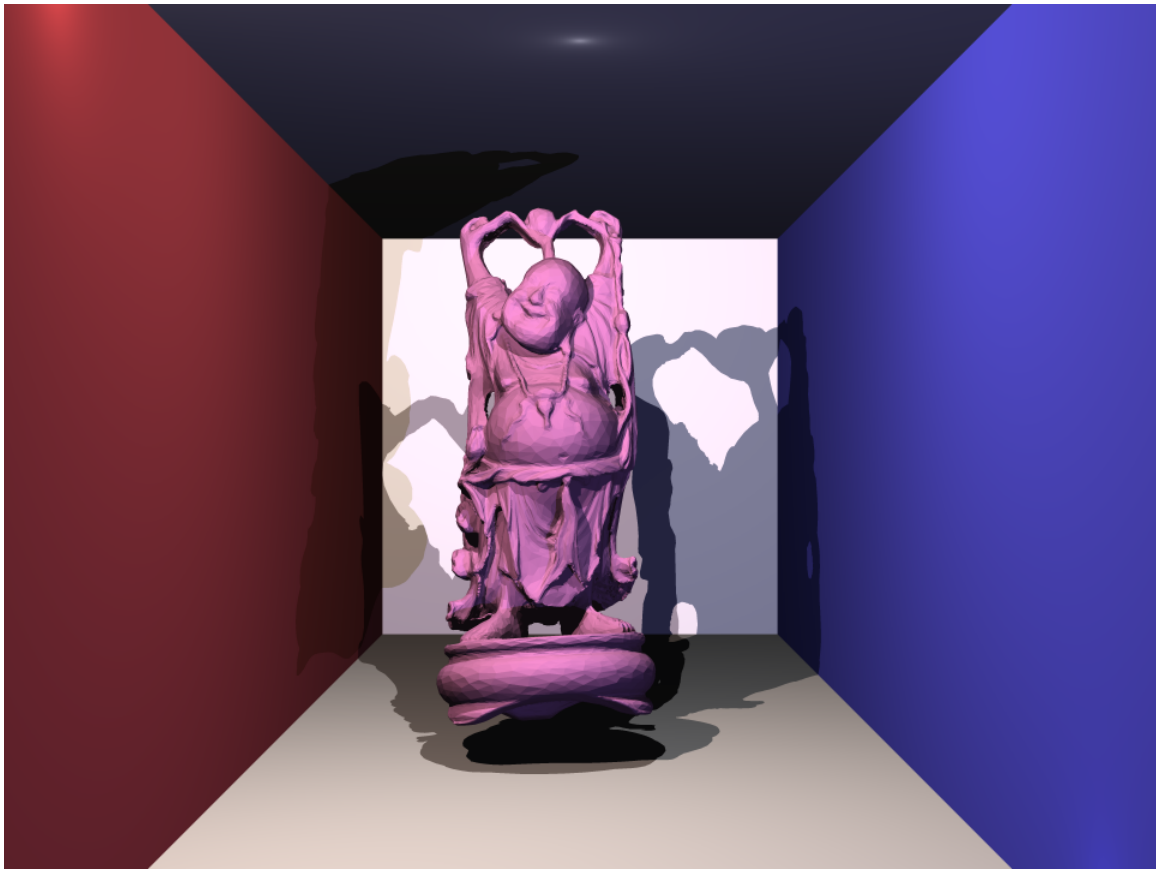
List of files that I added or modified:

- acceleration.js – I created this file for the k-d tree acceleration code.
- intersectScene.js – I modified this file for the traversal of the k-d tree.
- scene-orig.js – I modified this file, adding complex geometries and initializing the k-d tree.
- triangleBB.js – I refactored the triangle-bounding box intersection code.
- surface.js – I modified this file to account for ray refraction for glass materials.
- obj_[primitive].js – I created these OBJ files with the help of my Python parser.

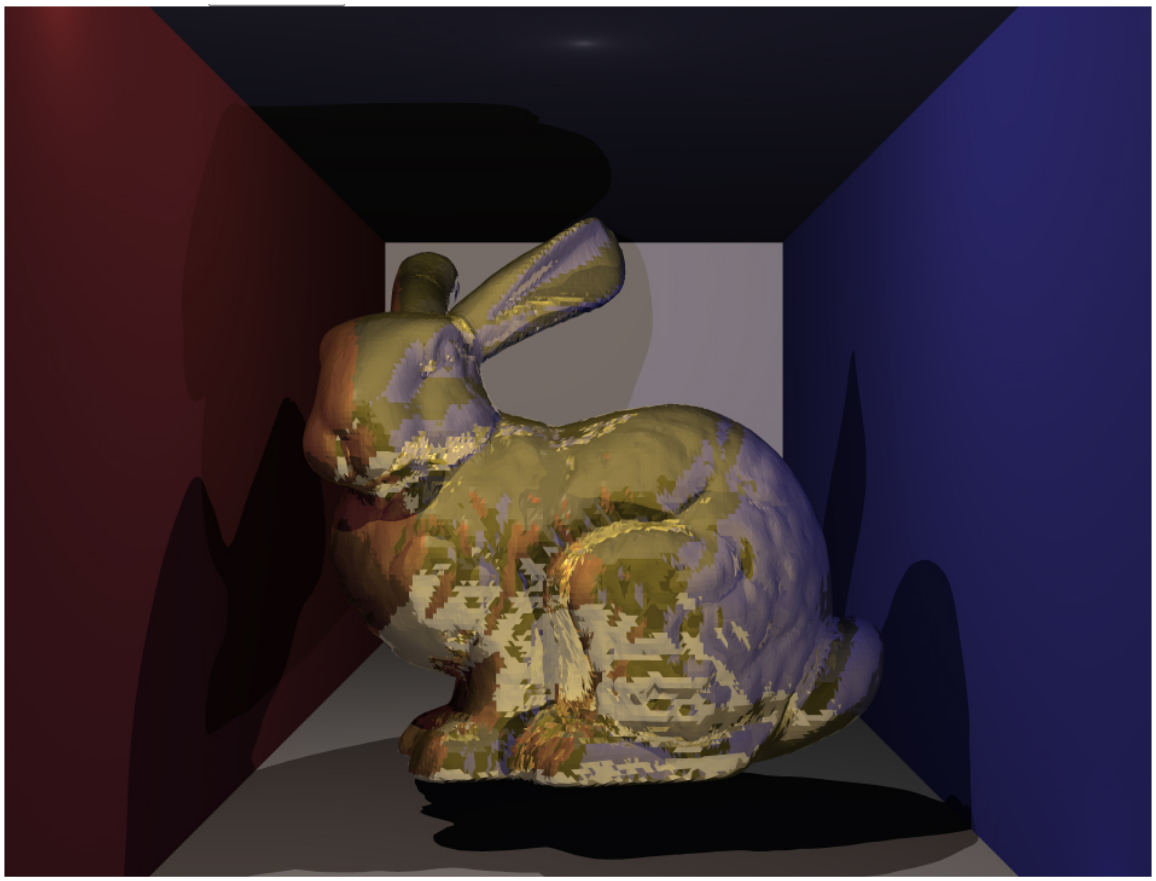
Additional Notes

This project built off of previous knowledge gained from the CPSC 478 homework assignments. I will upload the completed assignments to the classesv2 Dropbox.

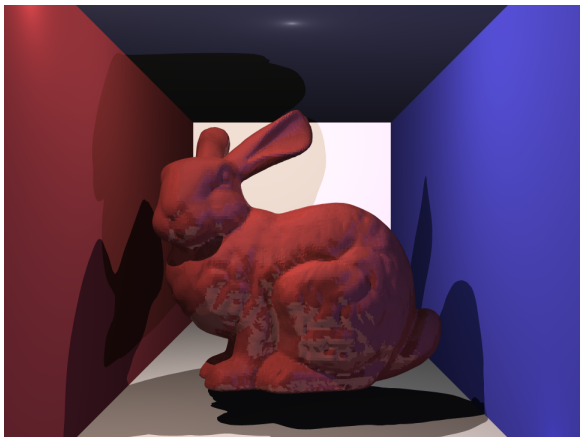
High-Res Pretty Pictures (for fun!): 960 pixels width, 720 pixels height



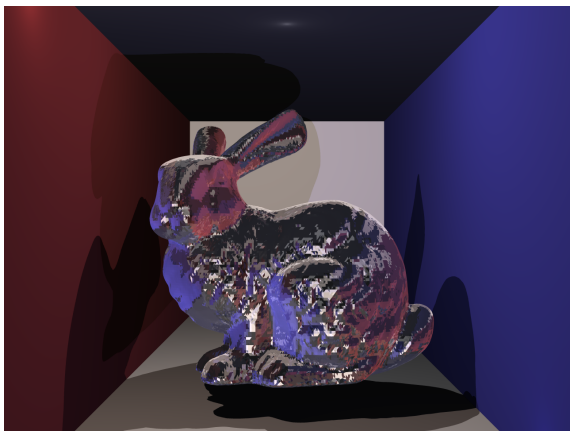
Above: Happy Buddha, matte pink material



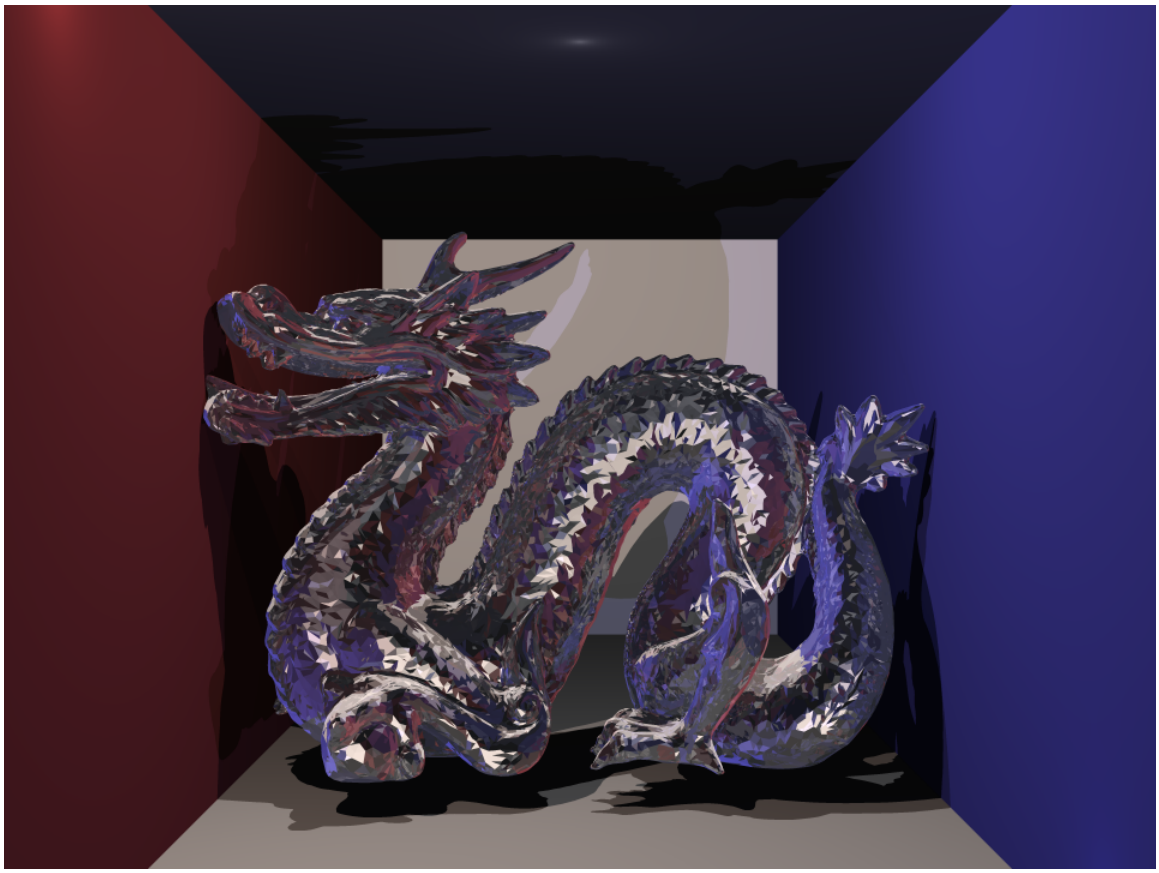
Above: Bunny, gold metallic material



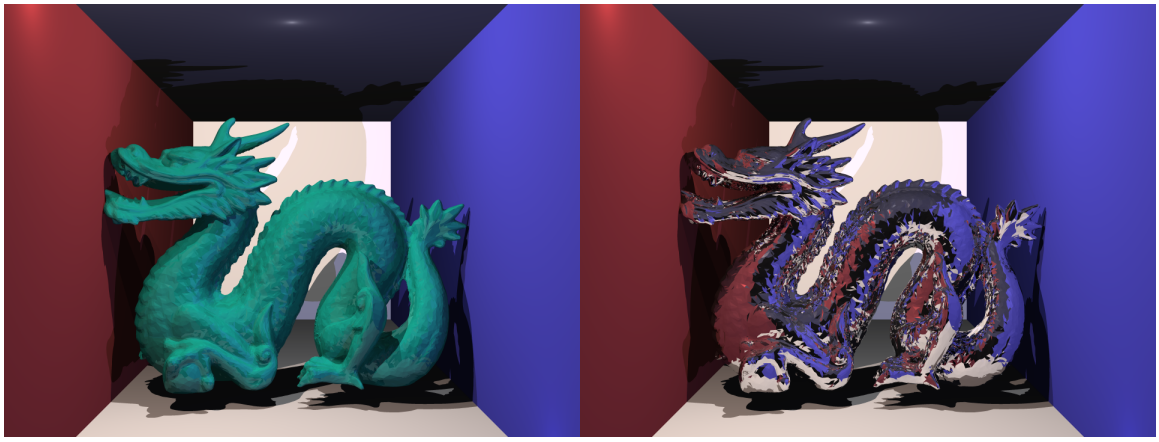
Left: Bunny, red glossy material



Right: Bunny, glass material (no caustics)



Above: Dragon, glass material (no caustics)



Left: Dragon, glossy aquamarine *Right: Dragon, mirror material*