**COMP 4490 Assignment 1**

**Part 1 & 2 Report**

**Professor: John Braico**

Student: Kyle Seokjin Ahn

Student#: 7794966

**Part 1**

**Sphere Intersection**

* Implementation of sphere intersection was very straight forward. Its implementation is in “example.cpp” file. Sphere intersection has to be 100% correctly completed as it had to be used in many other functions throughout the assignment. I have particularly looked at an example online: <https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-sphere-intersection>.

**Triangle Intersection**

* Implementation of triangle intersection was a little more difficult than the sphere intersection as I had to care of all three vertices of the triangle in order to calculate the normal. The triangle intersection is also 100% completed.

**Box Intersection**

* For box intersection, I have used Kay-Kajiya method, which is also known as AABB. Within Box class, I have a private int bounds[2], where I have declared b0 and b1 for MIN and MAX points of the box. The downside of using Kay-Kajiya method is that it can only draw aligned-axis boxes. Box Intersection is 100% completed.

**Phong Illumination**

* Phong illumination was simpler than expected. Originally, I had ambient, specular and diffuse coefficients pre-defined as I was not aware that the input ray files wanted to specify these values. All the objects in the scene used to share the same ambient, specular and diffuse coefficients before the changes. After the major changes, the program is now able to specify these coefficients for each and every object.
* Originally, diffuse coefficient was equivalent to the object’s colour. It was done this way since this provided a very clear output for Phong Illumination.
* I have set maximum and minimum values for (ambientColour + specularColour + diffuseColour) since the value sometimes went over 1.0f or under 0.0f, which ended up providing incorrect scene with randomly scattered white or black dots.
* Phong Illumination is 100% completed.

**Multiple Light Sources**

* One light source is a point light as required. Simply, I calculated the direction of the point light by normalising the light’s position – hit point.
* The other light source is a directional light as required. Simply, I simply normalised the position of the directional light.
* These two light sources are 100% completed.

**Distance Attenuation**

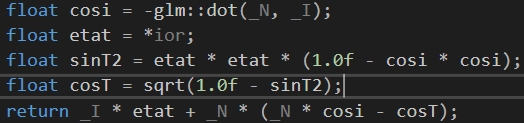
* Distance attenuation was the simplest part of the assignment. I had to plug in a few different numbers for constant, linear and quadratic attenuation coefficients to determine which values give the clearest output. There is one issue here. With the .ray files given by the professor, the coefficients does not seem to be compatible with my program. It does not even clearly render the distance attenuation effect. In order to clearly show the effect, some of the coefficient values are modified.
* Distance attenuation is 100% completed.

**Two Additional Scenes**

* Scene 1:
* Scene 2:

**Part 2**

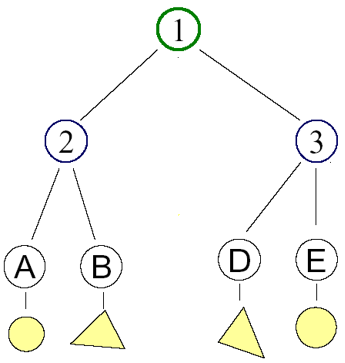
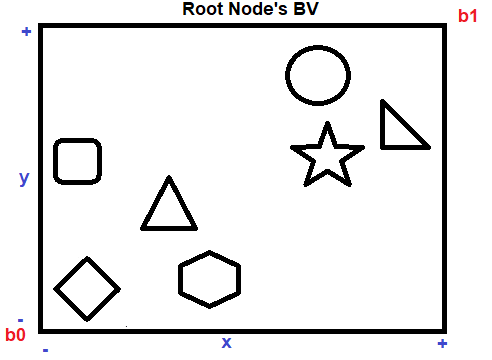
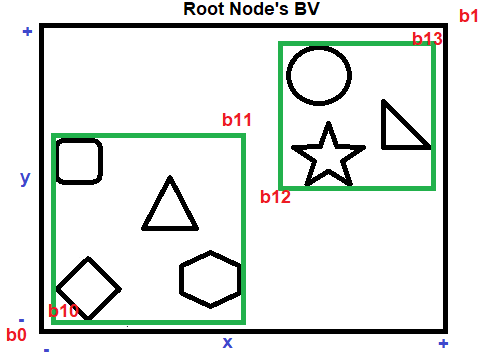
**Transmission and Refraction**

* Transmission and refraction are the parts that I struggled the most. The equation that I applied in order to get a refracted ray is correct. As you see, the code on the right side correctly computes the new direction for the refracted ray. With recursive function.

**Recursive Reflection**

* Recursive reflection was quite easy. With the existing castRay function, another parameter for depth was added with the base case for depth < max\_depth. New direction for the reflected light was calculated by the equation inside reflect function. This reflected ray direction was then passed as a new parameter to the recursive castRay function. For my program, I have set the maximum depth as 5 as there was not really more than 5 reflective objects. In my program, I had pre-defined the objects with “REFLECTION” surfaceMaterial so that only these objects can have the mirror-like effects in the scene. Unlike the scene files provided by the professor, I do not specify a specific reflective vector as my program simply has “REFLECTIVE” or “NON-REFLECTIVE” objects only. I wanted to change this to include reflective vector, however, I did not have enough time as I had to change all the codes in my program.
* One more special feature in my reflection is that I use “Frensel” function to give a more realistic effect on reflecting objects. From my experiment, using this function could give a glowing effect to some extent.
* Recursive reflection function is 100% completed.

**Acceleration Structure – Bounding Volume Hierarchy**

* For the acceleration structure of this program, BVH was chosen as it seemed to be not only simpler but also performant than the other acceleration structures discussed in the class. As discussed in the class, the goal of BVH is to rule out intersections that don’t occur as soon as possible, without having to do detailed geometry intersection tests. In this program, I have created a BVH class where I could initialise the BVH with given objects(meshes) in the scene. As you see the tree diagram on the right hand side, each leaf node contains an object. The interior nodes such as 2 and 3 contain a list of objects. The root, 1, simply contains all the objects in the scene. This BVH is pre-construted when the objects are created so that it could be used throughout the program. This part is quite straight forward.
* In my implementation of BVH, listSplit function is the one that makes my function a little different from others. The idea is that I divide a list of objects into two sub-lists according to the distance from each object to the bounding point, namely b0 and b1 of the current bounding volume. For example, there is a single list of all the objects in the scene in the root node obviously. This is the state after intialiseBVH() is called. Then, recursive buildBVH(), which contains listSplit(), is called to make the hierarchy structure.
* When the first listSplit is finished, as mentioned above, the current root’s list of objects is divided into two sub-lists. So in this case, rectangular, diamond, hexagon and right triangle will be one sub-list and circle, star and the other triangle will be another sub-list. Notice that the distance between “b0 or b1” and each object is calculated by the centroid( centre position of the object ). The new bounds are now b10 & b11 for the left sub-list, and b12 & b13 for the right sub-list.
* Every time the buildBVH() is called recursively, setBBOX() is also called in order to make the bounding volume box with the existing Kay-Kajiya method. As described previously, Kay-Kajiya method takes in one minimum and one maximum point to render an axis-aligned box. Using this method, when the list is divided, setBBOX() finds the minimum and maximum x, y and z out of all the objects in the sub-list in order to construct a new BBOX. As you see in the second diagram, the green boxes are created as a result of setBBOX() function. All of these information, including list of objects, bbox, left and right child nodes, are stored in a node to create a tree structure.
* Using the BVH, there is a very obvious improvement in performance. The number of calling traceRay() function is reduced by nearly one third. Accordingly, the running time of the program is reduced by one third, which is quite a significant performance improvement. Although the degree of improvement really depends on different scenes, it can be concluded that this BVH is working correctly to give a better performance.

**User Interface**

* This program is build on Visual Studio 2017, and the user is able to run the program by running the .sln file. Upon running, the user is able to select the scene that they want to render as well as the option for BVH or non-BVH to see the clear difference in performance.