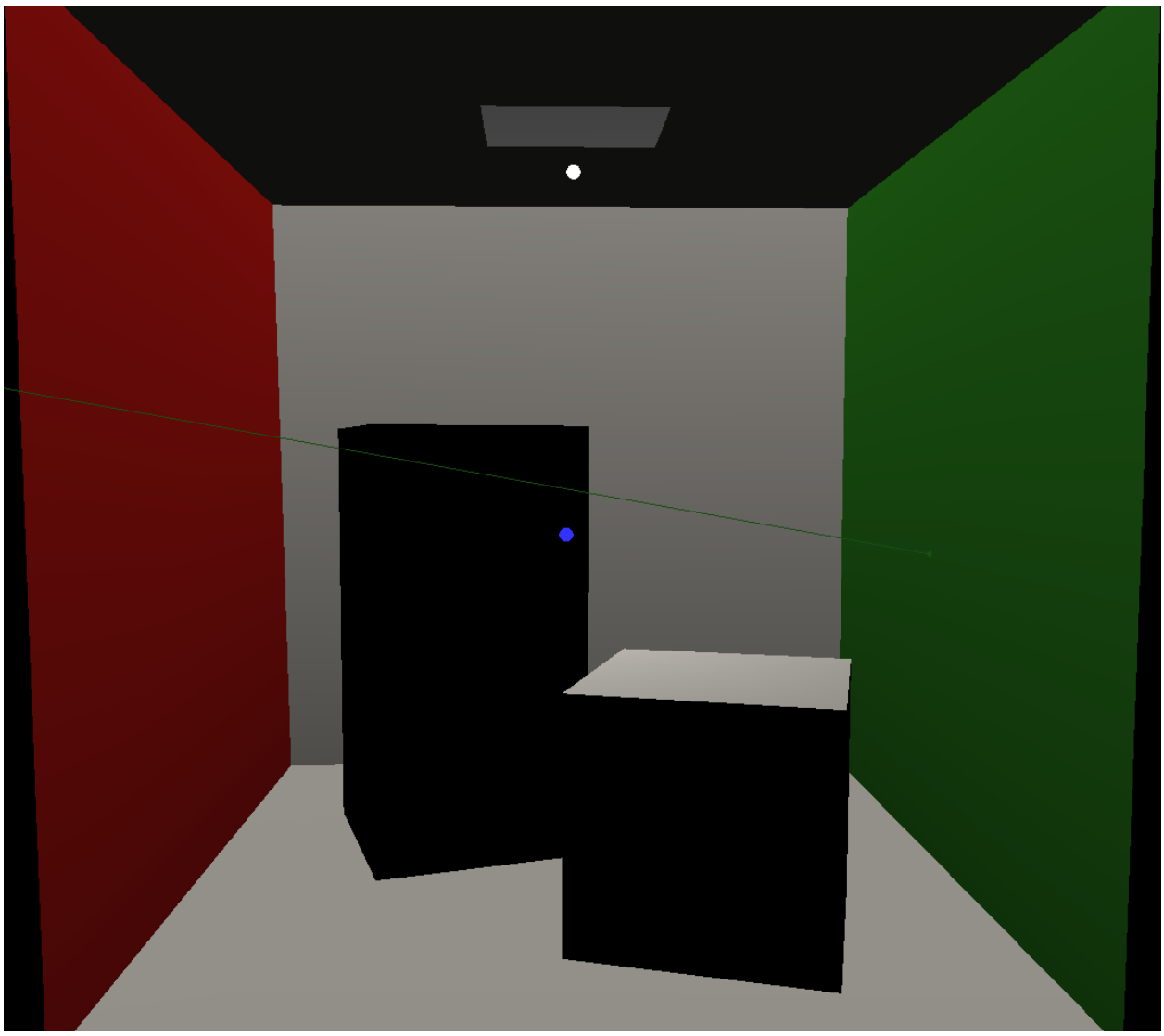
CSE2215 Final Project

Team members:

- Vasco de Graaff (Student number:5247470)
- Dan Teodor Savastre (Student number: 5212170)
- Luka Knezevic Orbovic (Student number:5253527)

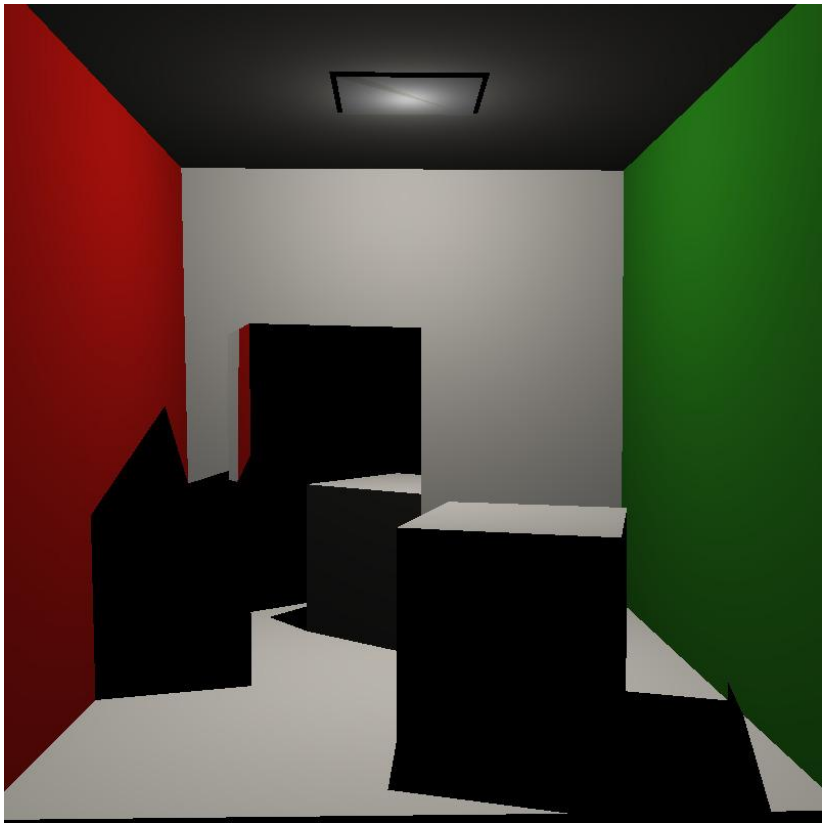
3.1 Shading and Intersection:

For this part of the assignment we created two new files “shading.cpp” and “shading.h” where we compute the color of the point where the ray intersects the scene. In the calculateColor method we loop over all lightSources and calculate the specular and diffuse terms for the hitPoint and then add them to its color. In the image below you can see a visual debug of a ray shot at the scene and it changing color to the color of the point it hit.



3.2 Recursive Ray Tracer

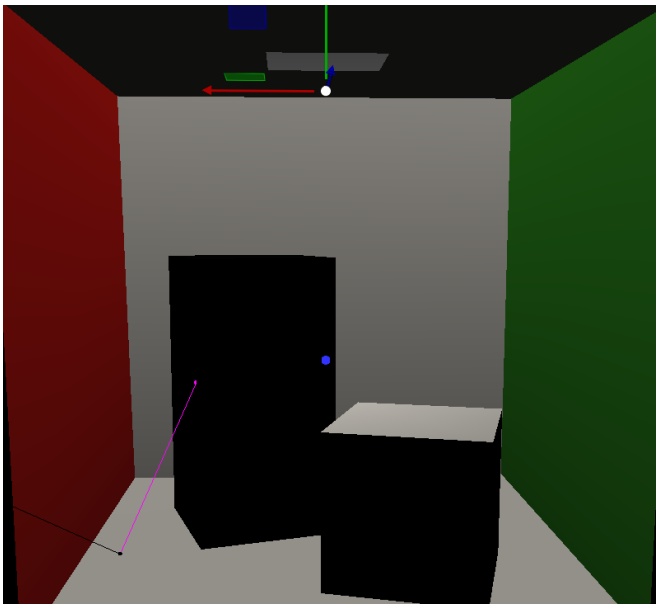
In this part of the assignment, we modified our ray_tracing.cpp file to ensure that our intersection methods were working correctly. As you can see, here we call the getFinalColor recursively and render the ray traced image. We update our ray.t value at each intersection and update it if it's smaller than the previous value. We also set the recursion limit to 3 to prevent an infinite loop which would cause a stack overflow. As you can see in the image below this also takes care of the mirror because when the ray checks the material of the hit point and if it has reflective properties it casts a new reflection ray from the point.

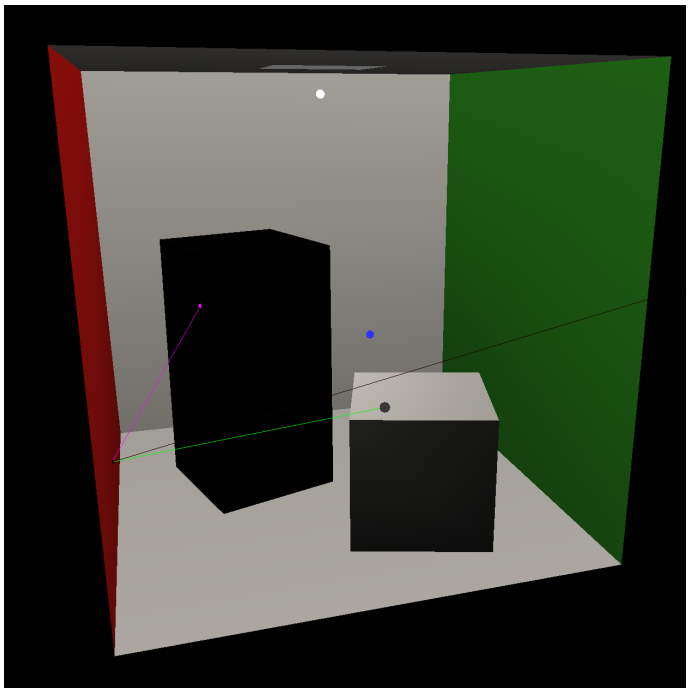


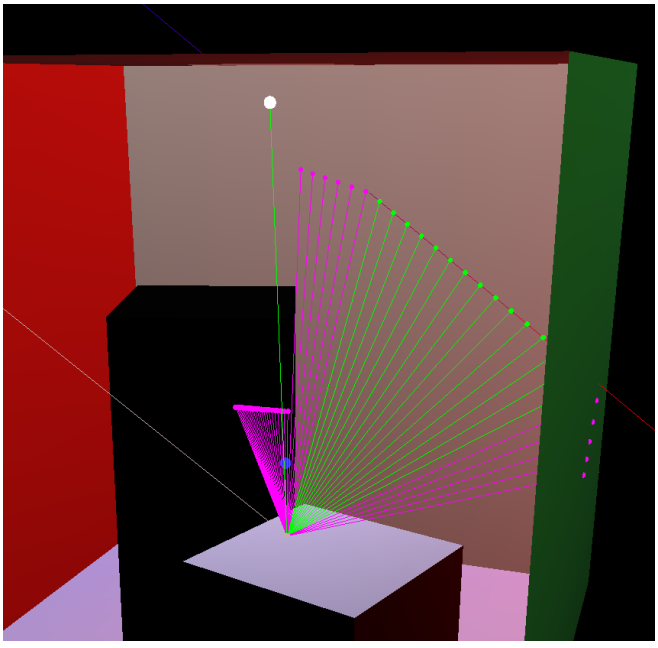
3.3 Hard shadows

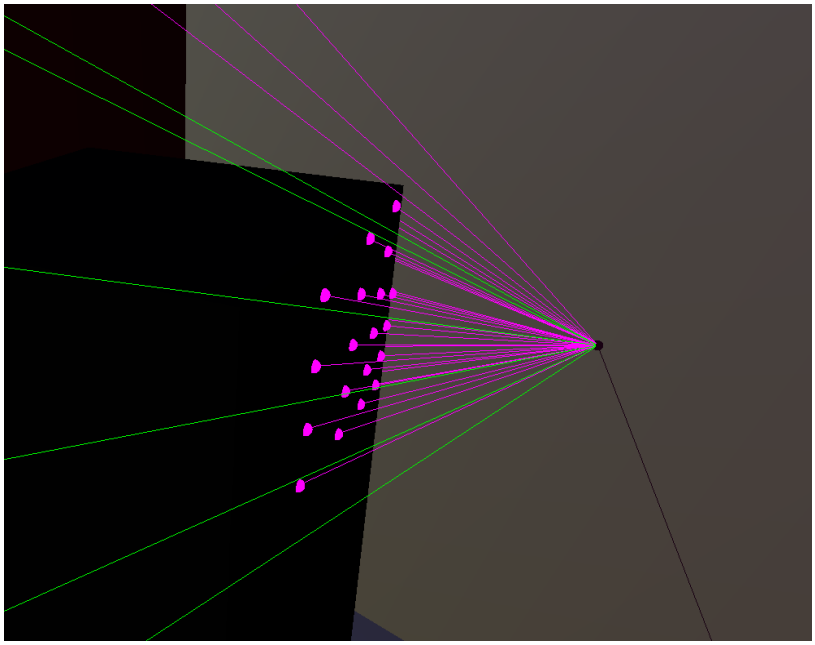
This part of the project is closely related to the shading methods we created before. In order to achieve hard shadows we shoot a ray from the position of the hitPoint towards the light source in order to determine if that light source should contribute to the shading of the point. This makes sense because if no light source hits this point then it will just have the default color black. In the following images we will see debug rays. There are two types of debug rays:

- Green - when the light source contributes to the shading
- Purple - when the point is in the shadow



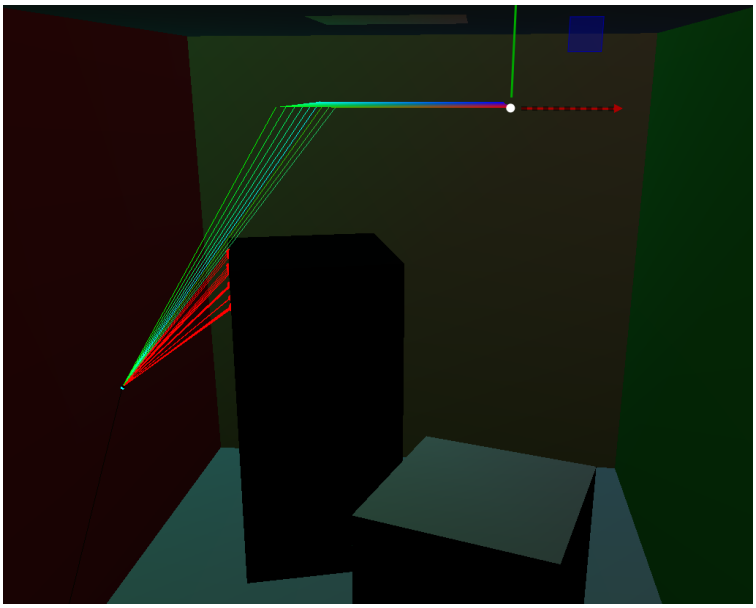






3.4 Area Lights

In this part of the assignment, we computed the area lights by splitting the light into equally distanced segments that then each contribute light to the point. When the light doesn't hit the point, we draw a red debug ray to the intersection point from the selected point. In combination, the light rays contribute specular and diffuse shading to the point.

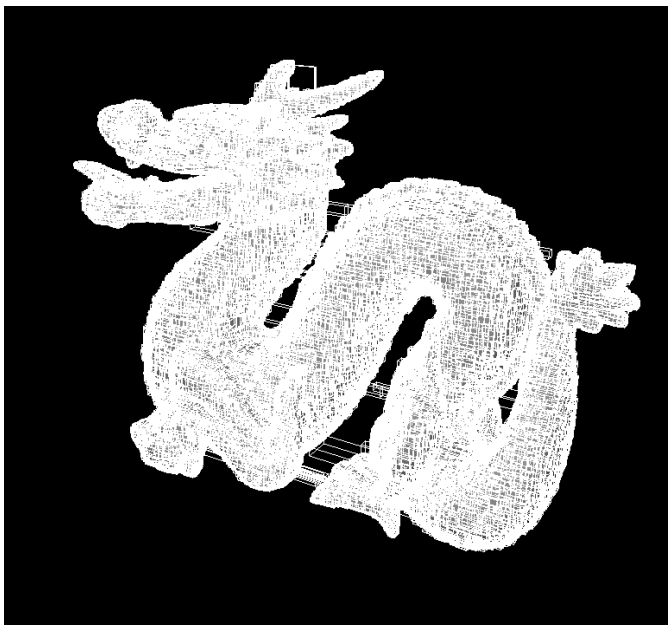


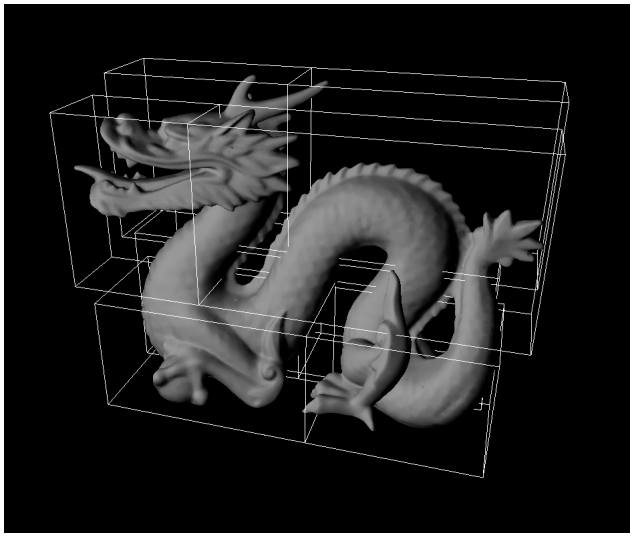
3.5 Acceleration data-structure without pointers(Generation):

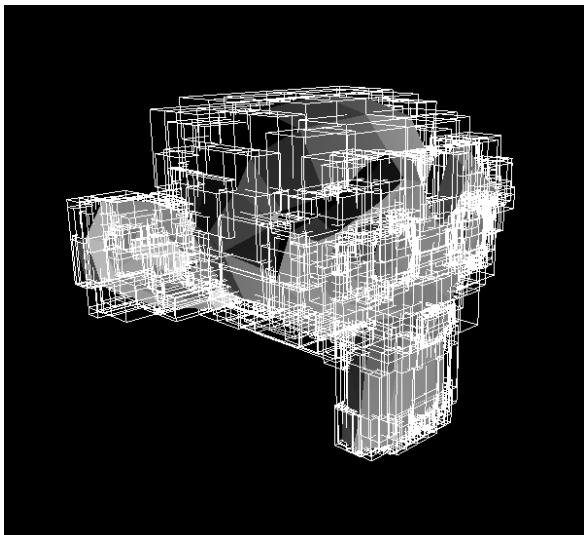
In this part of the assignment we create a tree that splits our meshes at every level and this way we only check the leaf nodes for intersections. We split nodes until the node only has one triangle. In order to split by all axis we use a switch statement for the level % 3 and this way we have 3 cases and for each one we split by a different axis. The way we store the tree is a standard library vector of nodes and for any node i its children will be 2 * i + 1 for the left child and 2 * i + 2 for the right child. After finishing the generation we update the levels variable to be the maximum level we reached.

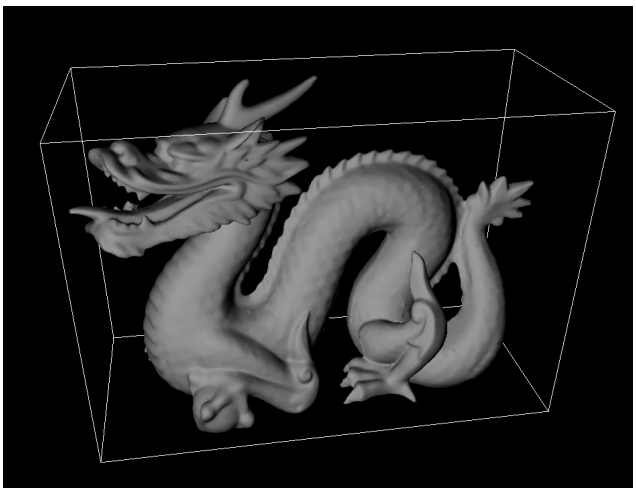
3.5 Acceleration data-structure without pointers(Traversal):

To traverse our tree, we utilized a priority queue data structure to efficiently traverse the tree. The priority queue sorts the elements by distance from the ray origin. If the node is a leaf, we go through its meshes and check if it intersects with the triangle constructed using the vertices from the mesh. If it intersects, we update the ray.t value. On the other hand, if it isn't a leaf node, we get its left and right nodes and add them to the priority queue. In the last two images we can see the first and the third level of the dragon bvh.



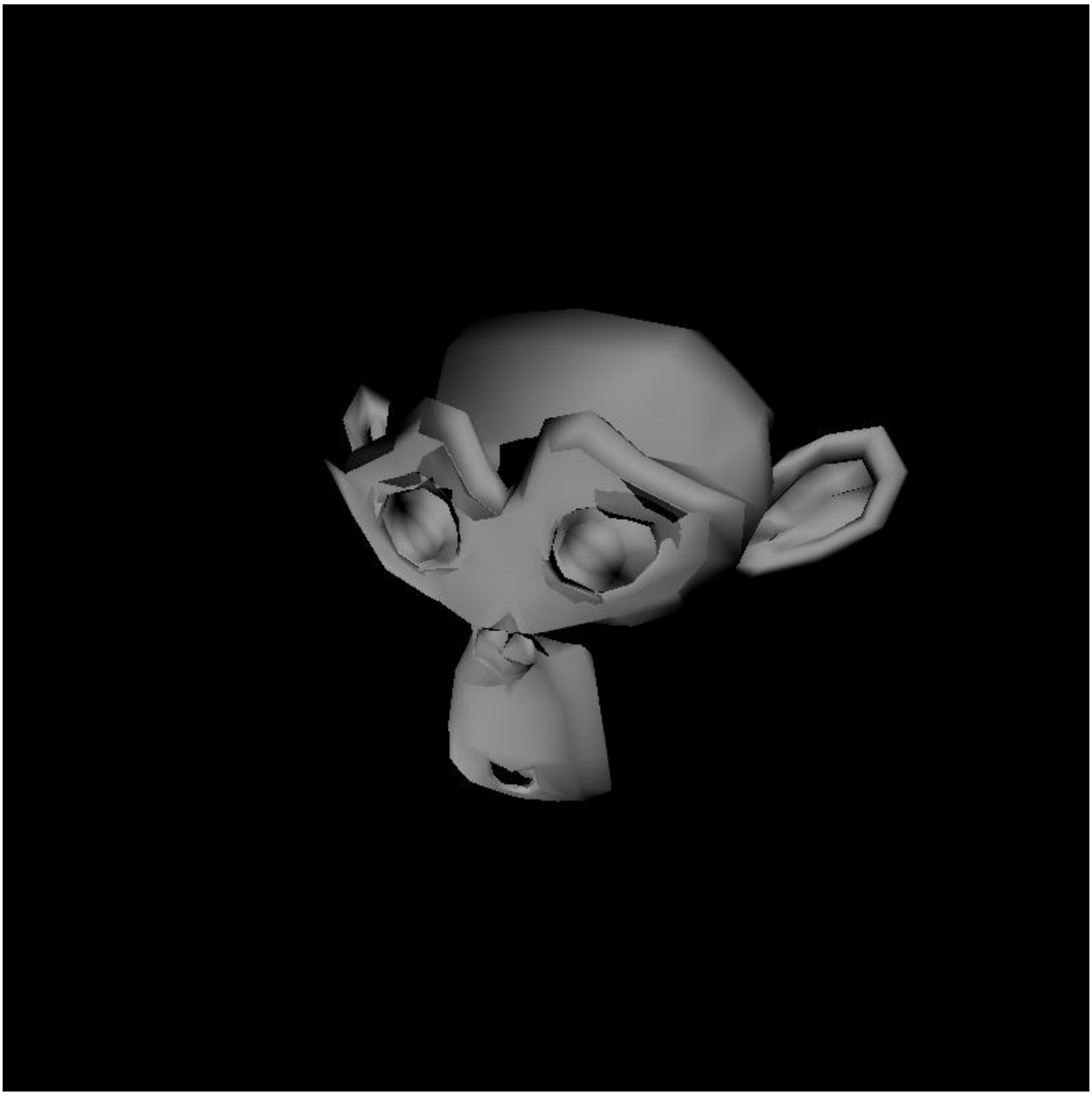






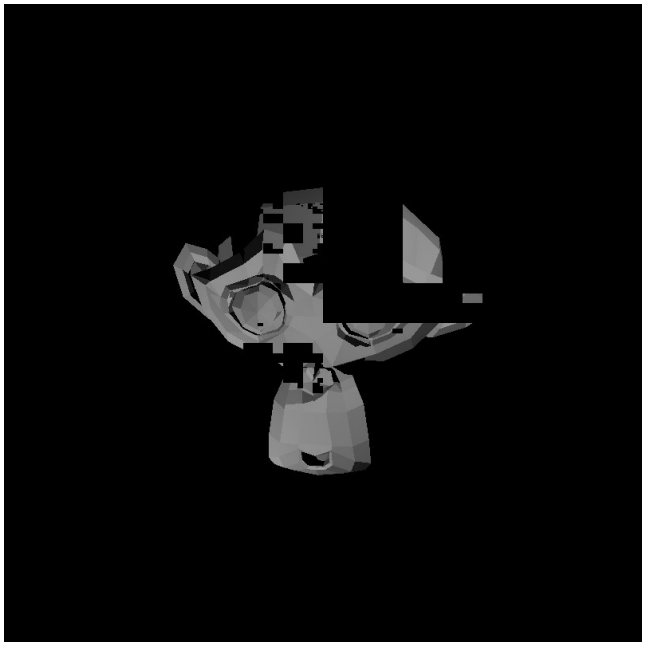
3.6 Barycentric coordinates for normal interpolation

For this part we work mainly in ray_tracing.cpp. Where we create two new methods called calculateBarycentric which calculate the barycentric coefficients and calculateBarycentricNormal which we use these barycentric coefficients to compute the interpolated normal. These two methods are then used in the already created method: intersectRayWithTriangle. In here we use the two previously talked methods to update the hitInfo.normal into this new interpolated normal.



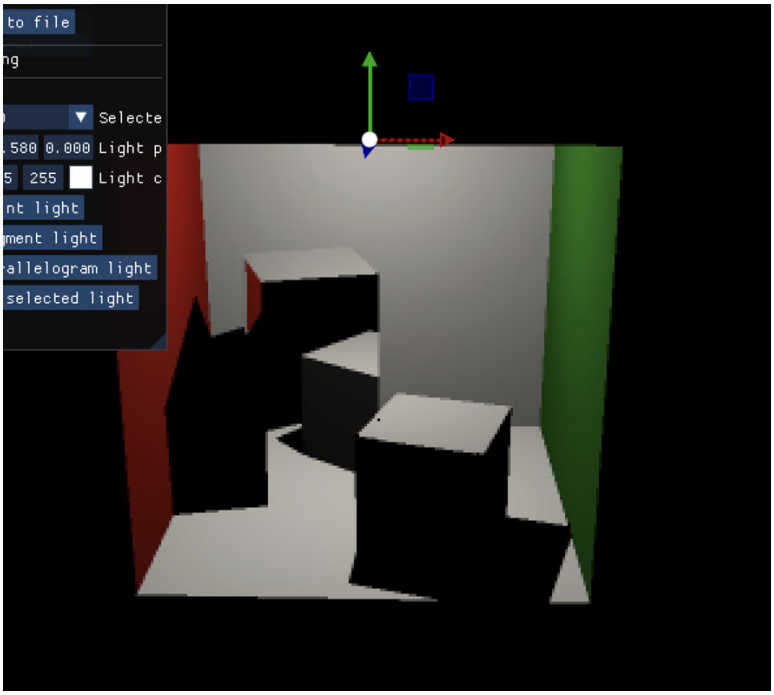
3.7 Textures

For this part we work mainly in ray_tracing.cpp. Where we create one method called calculateTexturePos which works in a similar way as calculateBarycentricNormal but we grab the texture coordinates to return those texture coordinates adapted with the barycentric. This method is then used in the already created method: intersectRayWithTriangle. In here we get texCoord from the vertices and use the previously talked method to uptake the hitInfo.texture which is what we have to use when the hitInfo has a texture if not we just use hitInfo.material.kd.



Texture filtering: Bilinear interpolation.

We achieve our bilinear interpolation by sampling the 3 adjacent pixels to the current pixel and using their color values to average them out. We call the method inside the main function where the pixels get colored. Additionally, we also added a button so that you can toggle bilinear interpolation mode.

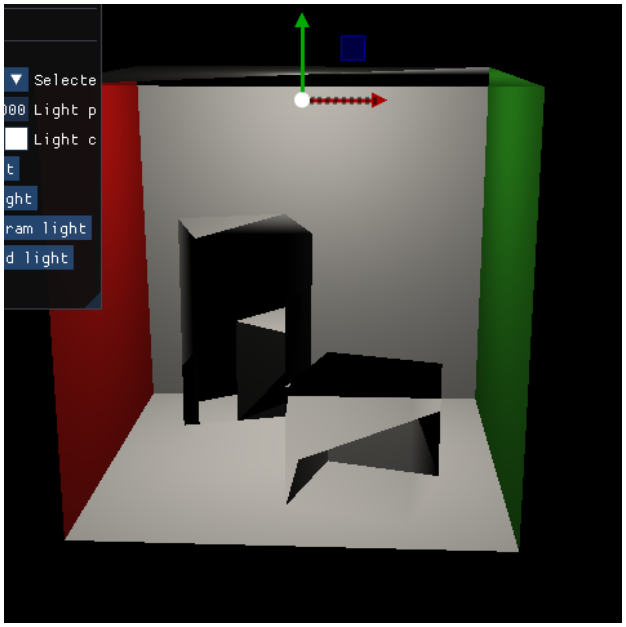


Performance tests:

	Cornell Box	Cornell Box with parallelogram light	Monkey
Number of triangles	32	32	958
Time to render	1144.04 ms	12902.7 ms	4655.48 ms
BVH levels	3	3	3
Max triangles / Leaf node	11	16	287

Known issues/ bugs:

BVH generation causes some rendering issues when the bvh level is over 5.
Bilinear interpolation doesn’t work on release mode.



In Texture the object renders but has a hole. It doesn't work properly