Computer Graphics

Final Report

Group 128

Team member: Yongcheng Huang(5560950)

Vladimir Pavlov(5451981)

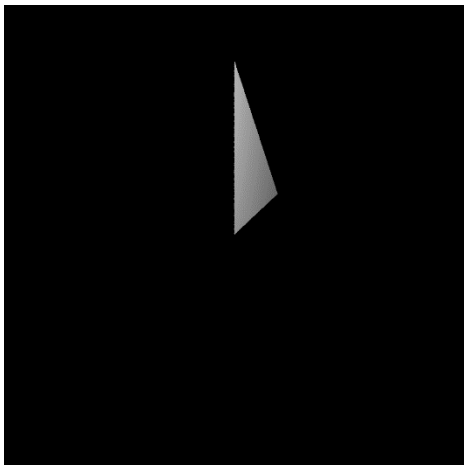
Stratan Alexei(5586550)

1.1 Shading using Phong Illumination Model

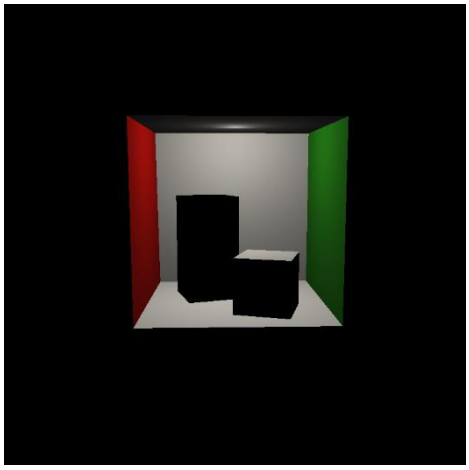
1.1.1 Description

This feature is a part of the basic features. In ‘Shading’, basic shading using phone illumination model is implemented to calculate the final color of the pixels which are being illuminated by the given light source. This is mainly related to the function **getFinalColor and in shading.cpp**.

1.1.2 Rendered Image

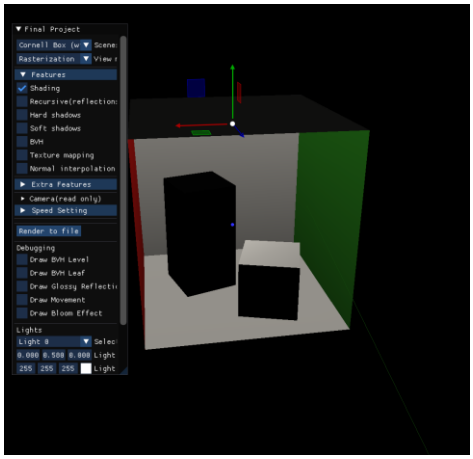


(shading for triangle)

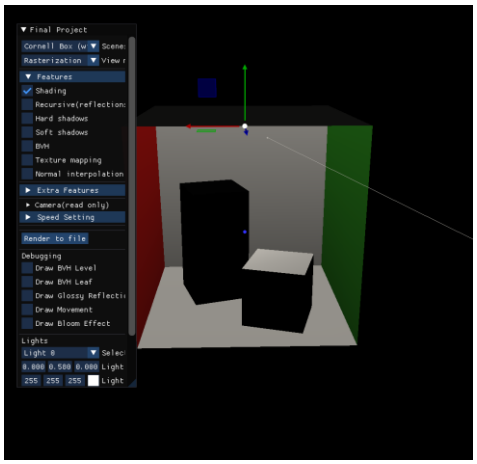


(shading for cornell box)

1.1.3 Visual Debug



(visual debug of the green wall)



(visual debug of the white wall)

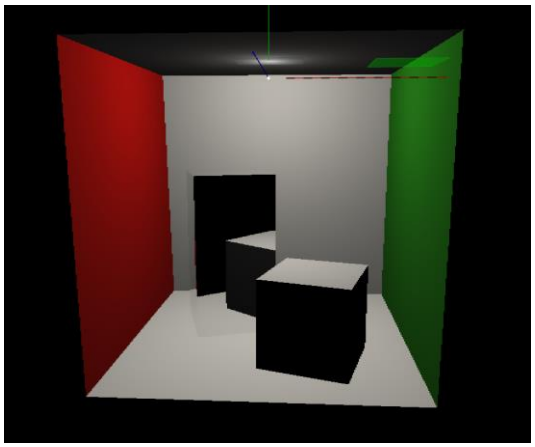
1.2 Recursive ray-tracer (reflection ray, shadow rays)

1.2.1 Description

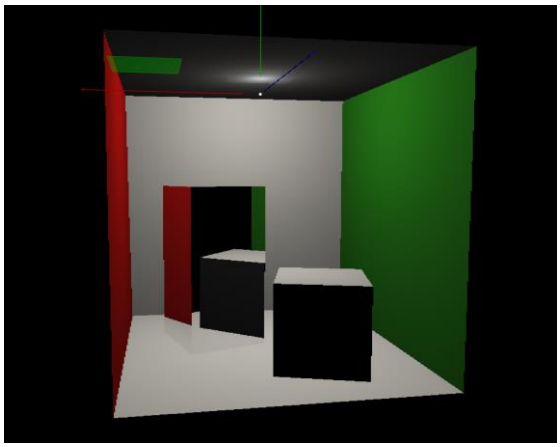
This functionality is implemented in the following two methods: computeReflectionRay and getFinalColor. In the former I have written the functionality to reflect a given ray as a parameter. I do this by creating a new ray. Its origin is the intersection point of the incident ray. I also put a very small offset to account for floating point errors. I calculate the direction using the formula for ray reflection which is provided in the textbook. I set the value of parameter t to be infinite. In the getFinalColor I call the first method to calculate the reflected ray using the incident ray and hitInfo object. Each subsequent ray accounts for the light contribution in the scene, this is done by calling

getFinalColor on the reflected ray. Also, I have put a stopping condition called ray depth, so that it can limit the recursion. The recursion will also stop if the surface is not specular.

1.2.2 Rendered Image

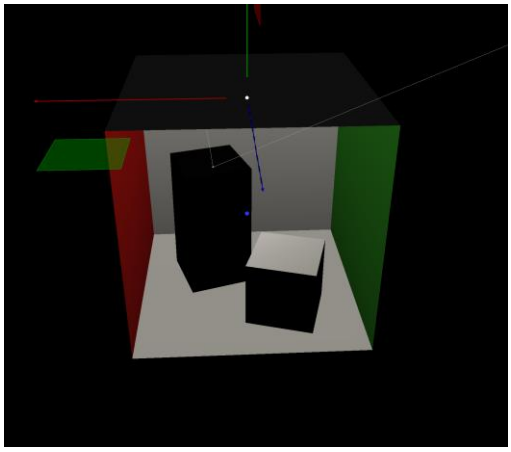


(recursive ray-tracer with shading)

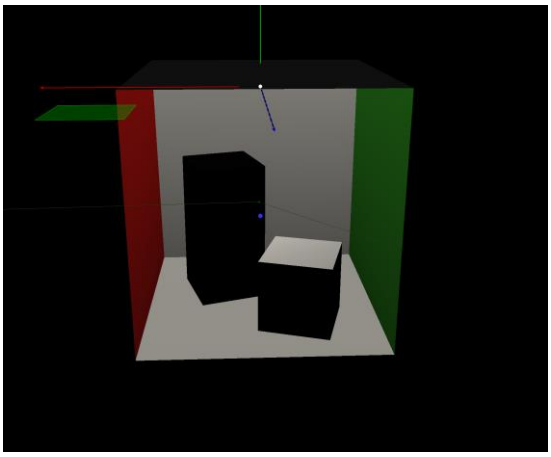


(recursive ray-tracer with shading 2.0)

1.2.3 Visual Debug



(visual debug reflected ray into back wall, white)



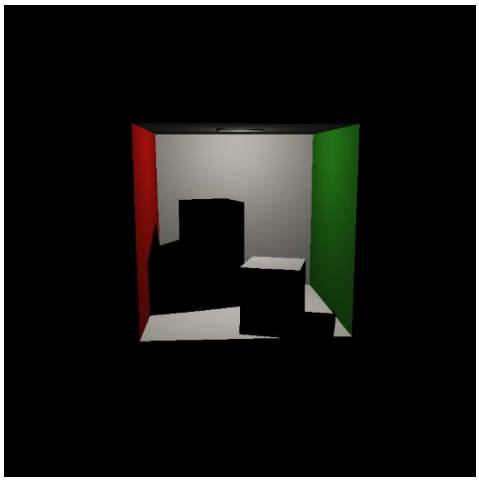
(visual debug reflected ray into right wall, green)

1.3 Hard shadows

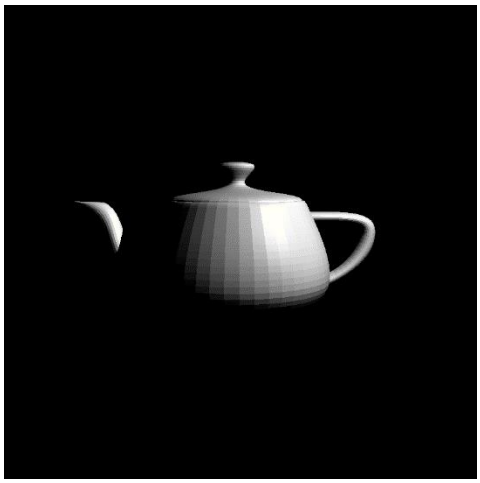
1.3.1 Description

Hard shadow is the basic part of shadowing. It means adding the shadow effect to the object if it is illuminated by some kind of light source. If an object blocks the light, there will be a black part behind/near the object and that is called shadow. This part is implemented by checking whether the point on objects block the light rays. It is mainly in **light.cpp**.

1.3.2 Rendered Image

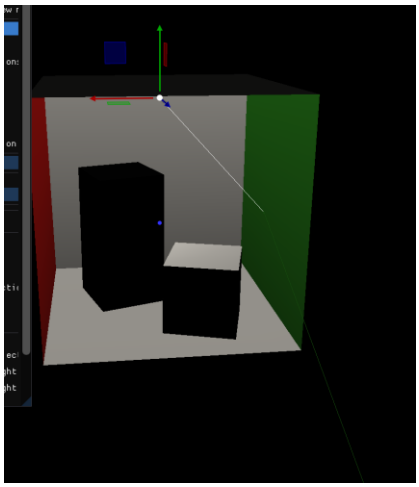


(hard shadow of cornell box)

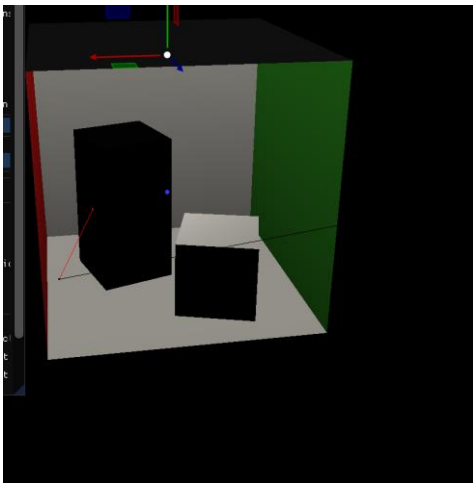


(hard shadow of teapot)

1.3.3 Visual Debug



(white line: not blocked)



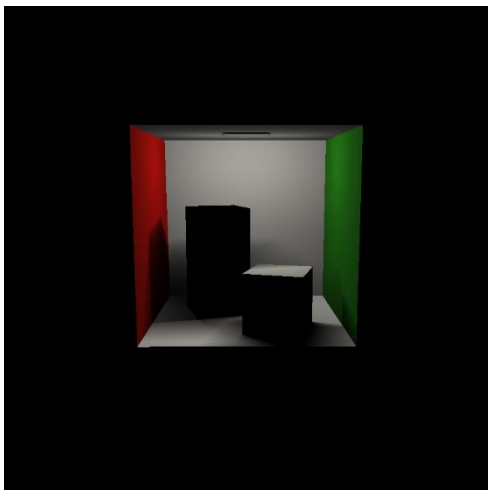
(red line: blocked)

1.4 Area lights

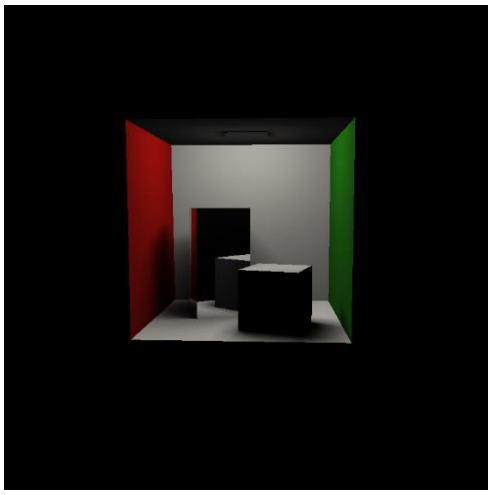
1.4.1 Description

Area light can be taken as a lot of point light in a special shape. There are two kinds of lights: segment and parallelogram. This feature is implemented by sampling the area light source as a predefined number of point light, calculating the shadow and finally combining them together. The final result can be seen as a soft shadow. This is implemented in the **light.cpp**. Adding a random epsilon to make the aliasing effect eliminated.

1.4.2 Rendered Image

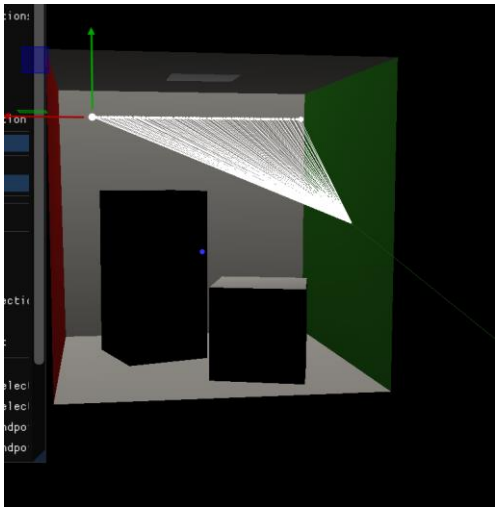


(soft shadow, segment light)

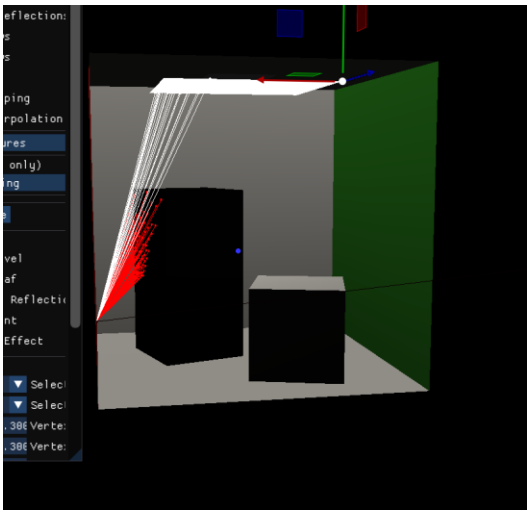


(soft shadow, parallelogram light)

1.4.3 Visual Debug



(segment light visual debug)



(parallelogram light visual debug with rays blocked)

1.5 Acceleration data-structure generation

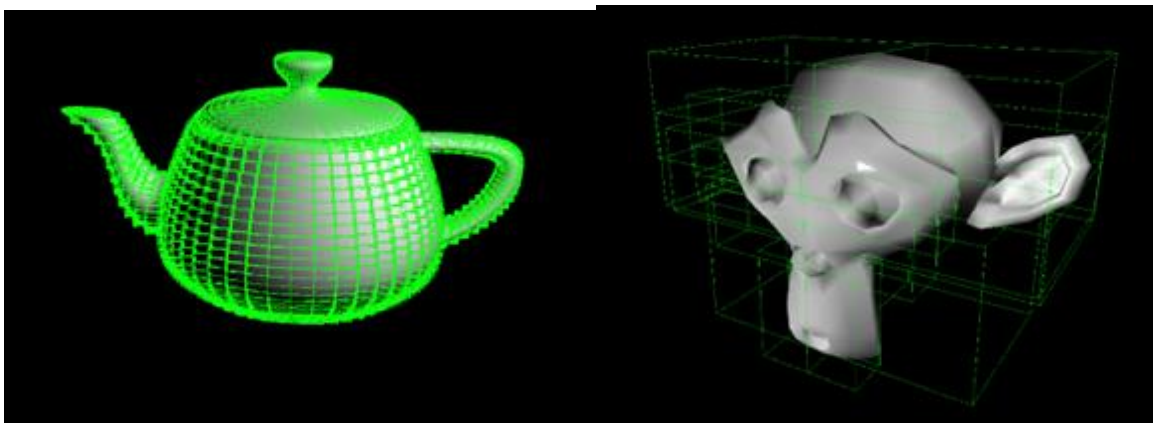
1.5.1 Description

The bounding volume hierarchy(BVH) is used to speed up the intersect method with a mesh. It is implemented in the constructor method of the bounding_volume_hierachy class and the method updateNodeBounds, which updates the bounds of a BVHNode based on the smallest and largest coordinate values of the triangles it contains, and the method subdivide which recursively creates the tree, splitting all triangles a node contains between 2 new children nodes based on the median centroid of all triangles contained by the node, which are calculated at the start when the triangles indices are loaded into a vector. The subdivide method's exit condition is either the current level reaching the max level(specified in the header file) or all nodes having only 1 triangle. The node struct contains a 2 float values, the lower and upper bound of the node's AABB, a Boolean value which indicates whether the node is a leaf, and a vector which stores the indices to the node's children, or information about the triangles contained if the node is a leaf. The triangles are stored in a vector inside of leaf nodes, which are indicated by having the boolean value isLeaf equal to true, as

indices to the mesh they are in, followed by the indices of the 3 vertices of the triangle as they are stored in the Vertices vector inside the Mesh struct.

1.5.2 Rendered Image

1.5.3 Visual Debug



Teapot show levels at maximum level

Monkey with show levels and show leaf debug

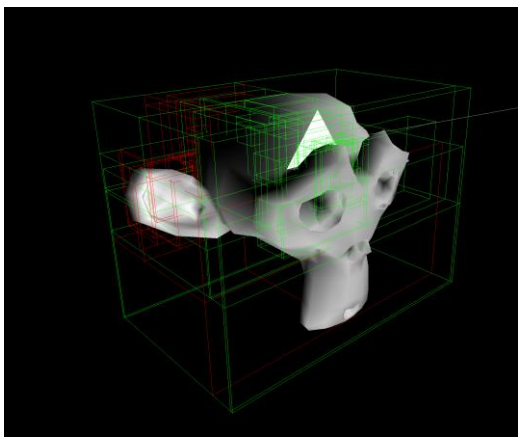
1.6 Acceleration data-structure traversal

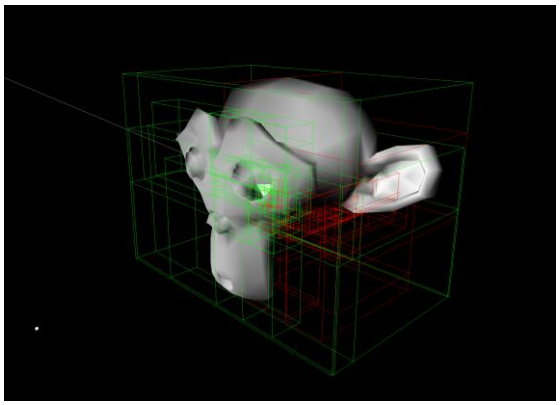
1.6.1 Description

The implementation of the feature is implemented in the interHelper method which is invoked in the intersect method. Essentially this is a recursive where it accepts a specific node. If that node is a leaf node, it stops the recursion, gets the intersected triangle from the mesh and returns true. If the node is not a leaf node, it gets the children of the node, calculates the distance to each one of the children, and invokes the interHelper method recursively on them, depending on which one is closer. Then it returns true/false depending on whether there has been an intersection somewhere in the children nodes. The visual debug is implemented within the interHelper method. The intersected nodes are colored in green while the ones that are intersected but not traversed are colored in red. I keep the nodes that have interHelper called upon in a list. Whenever the method traverses them, I color them in green. The red nodes are the ones that the ray would have intersected if there was no intersection, because they are essentially intersected, but not traversed.

1.6.2 Rendered Image

1.6.3 Visual Debug



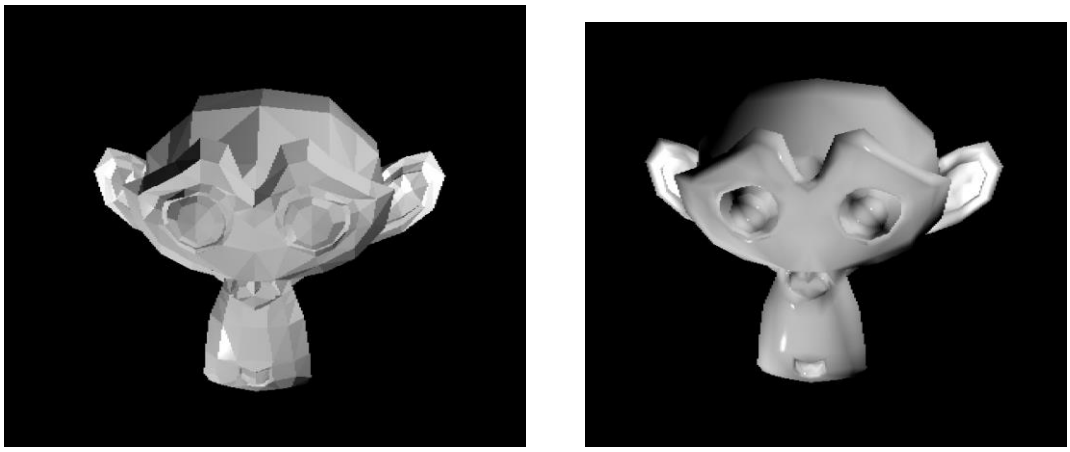


1.7 Normal interpolation with barycentric coordinates

1.7.1 Description

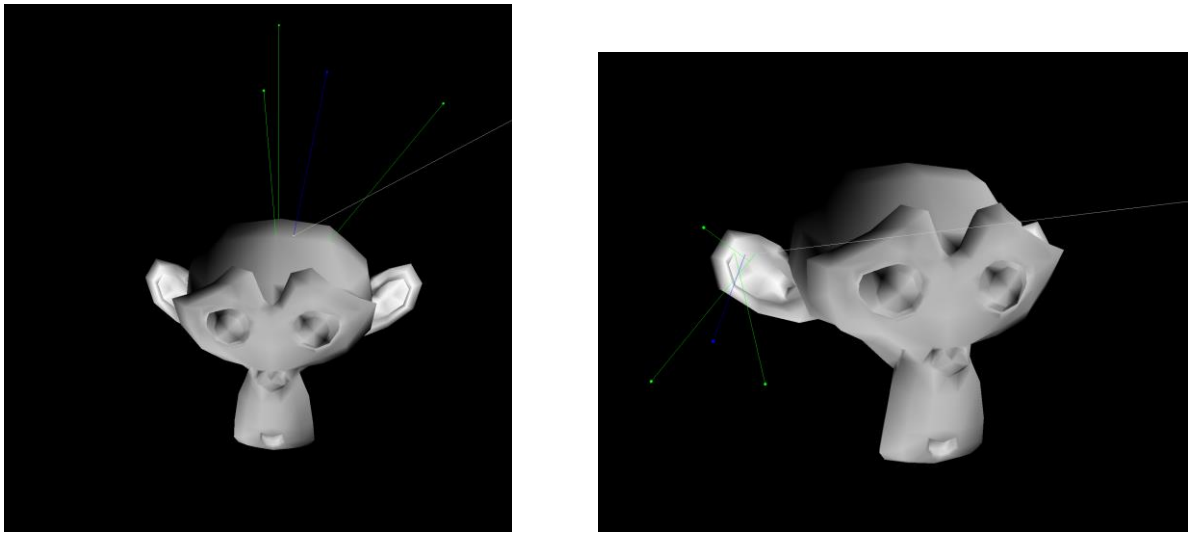
The functionality is implemented within the following methods: computeBarycentricCoord and interpolateNormal. In the first method, given three vertices of a triangle, I first calculate the surface normal of the triangle, then I use it to find the areas of the smaller triangles as a fraction of the intersected triangle. By doing this, I find the parameters alpha, beta and gamma. In the second method I compute the interpolated normal by getting the components of the barycentric coordinates and multiplying it with the corresponding normal. Then I add the resulting vectors, normalize the final vector and then return it. The visual debug functionality is implemented in the intersect method. I get the vertices of the intersected triangle then I draw all the rays and the interpolated normal.

1.7.2 Rendered Image



(without and with normal interpolation)

1.7.3 Visual Debug



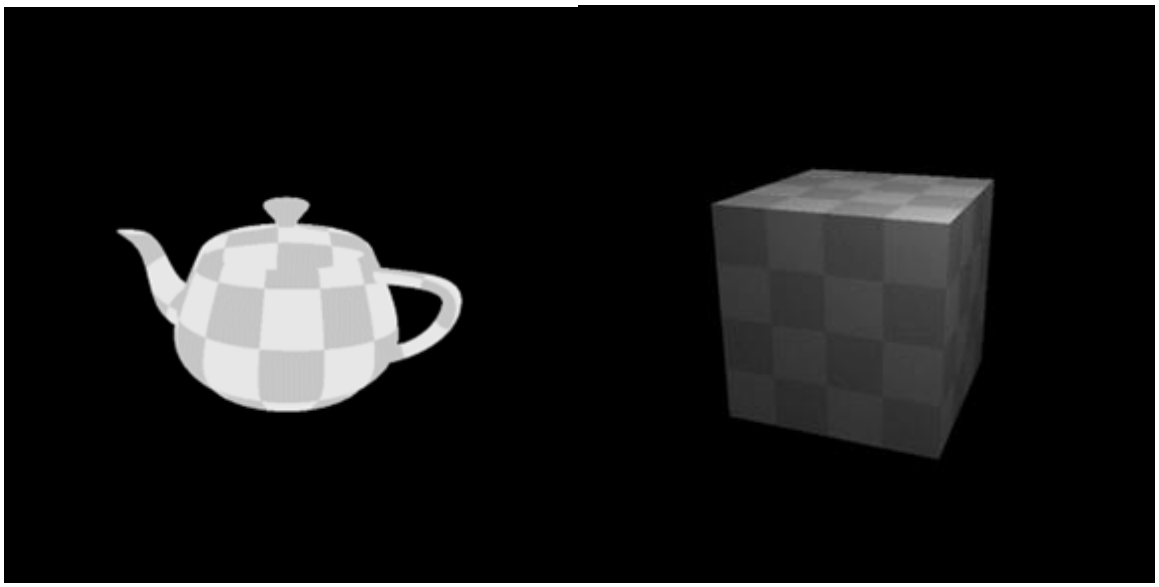
(Visual debug of the barycentric interpolation)

1.8 Texture mapping

1.8.1 Description

Texture mapping is implemented by first determining the tex coord of the intersection point by interpolating between the tex coords at the 3 vertices using the calculated barycentric cooridinates. Once that is calculated the material.kd value is changed for every point based on the corresponding value in the mesh.material.image vector, changing the value of the color used when drawing the objects and calculating their shading.

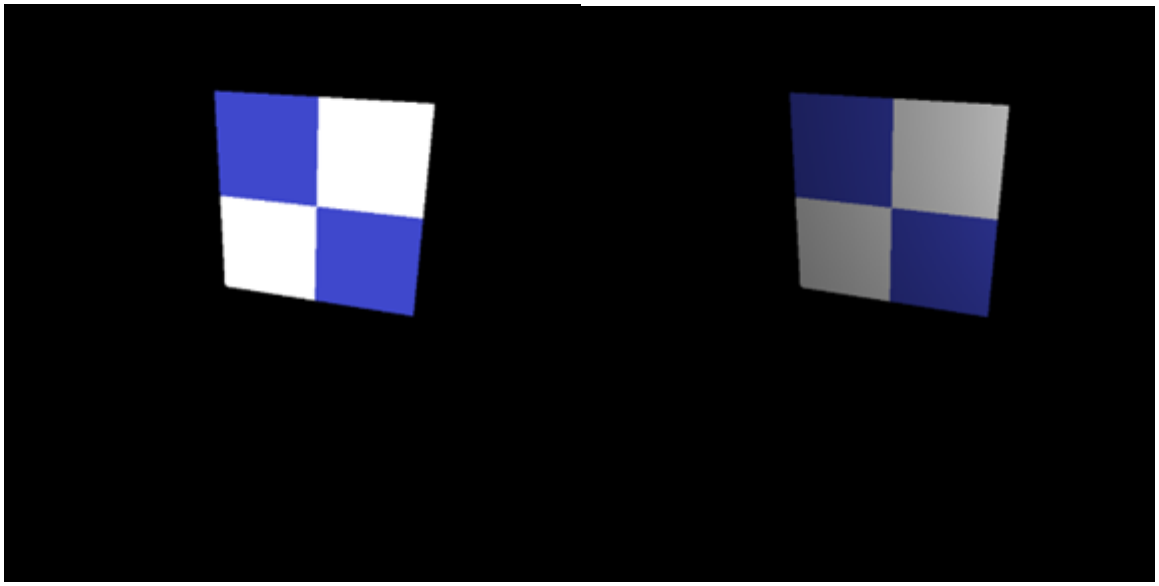
1.8.2 Rendered Image



Teapot with textures

Cube with textures and shading

1.8.3 Visual Debug



Quad with 2x2 texture showing clean border between the 2 triangles with shading off and on

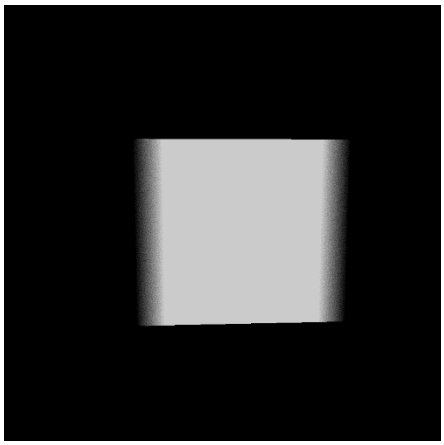
2.1 Motion blur

2.1.1 Description

Motion blur is for simulating the moving effect of the objects. This is implemented by adding a random distance to the origin of the ray which is used for getting the final color, and then combining all the effects together with lots of times. I choose to do this pixel by pixel since the layer by layer way will be destroyed in Multi-threading(can't decide the sequence of executing threads). In 1000 epoches, calculate the color of the specific pixel and save it to a vector and then set the pixel color as the value in the specific position of the vector, which is a sum of color in 1000 loops divided by 1000.

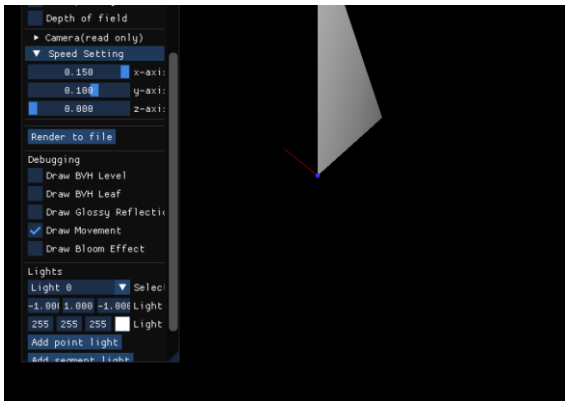
2.1.2 Rendered Image

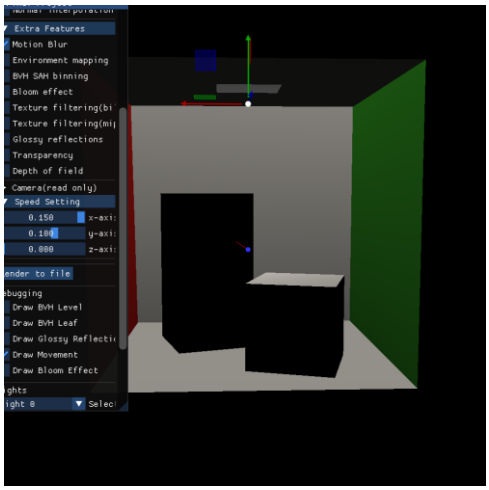
(motion blur effect of cornell box)



(motion blur effect of square)

2.1.3 Visual Debug





(when choosing draw movement, a red light in the length of the speed value and the direction of the speed will be drawn to represent the movement)

2.1.4 Location

This method is implemented in **render.cpp** (method **motionBlurRayTracing**)and the visual debug is implemented in **main.cpp** (line 373, if statement).

2.1.5 Reference

Wikipedia: https://en.wikipedia.org/wiki/Motion_blur

Answers(Basic TH): https://answers.ewi.tudelft.nl/posts/17883

LearnOpenGL CN:

https://learnopengl-cn.github.io/04%20Advanced%20OpenGL/05%20Framebuffers/#_10

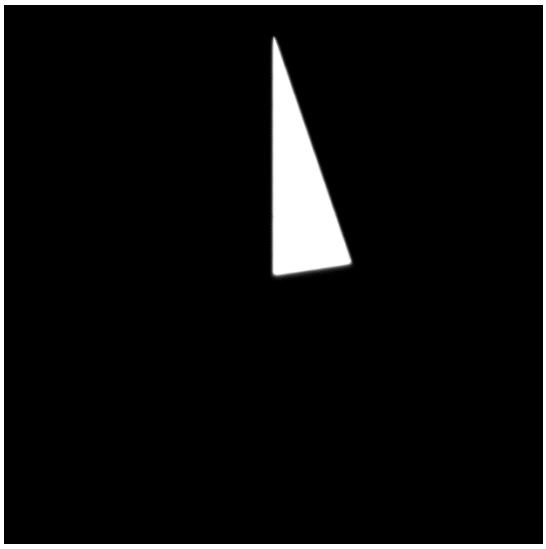
ZhiHu: https://zhuanlan.zhihu.com/p/446316624

2.2 Bloom filter on the final image

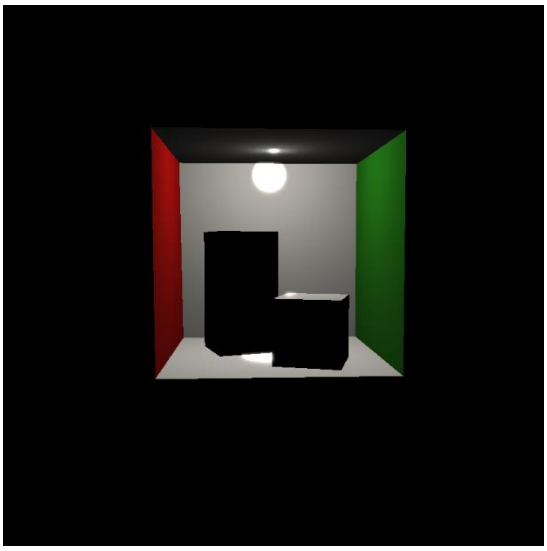
2.2.1 Description

Bloom effect is to simulate the halo effect of the light source, as what we can see in the real world. This is implemented by checking a pixel whether its *lightness is higher* than a specific value(here it is 0.7), if true then giving a bloom effect (using a box filter in 7x7 area), if not keeping it the same. After calculating the bloom effect, combine the bloom effect with the original screen pixels by adding the color together.

2.2.2 Rendered Image



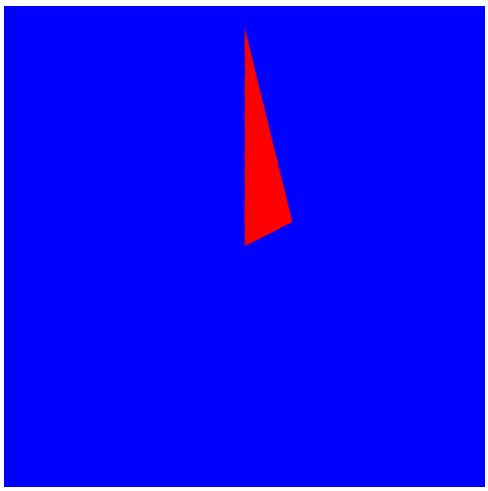
(bloom effect triangle)

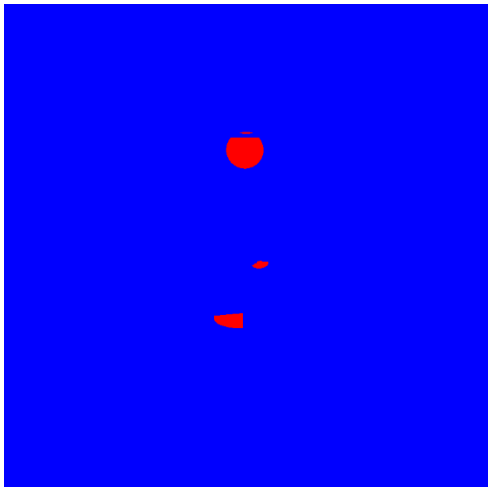


(bloom effect cornell box)

2.2.3 Visual Debug

The bloom effect visual debug is hard to implement. Thus I found a way to do this by showing the area which is considered as ‘light’(lightness > 0.7). All pixels will be in red if it is considered as light, or in blue if it is not.





(the image rendered when choosing debug bloom effect for the scenes above)

2.2.4 Location

This feature is mainly implemented in **render.cpp**, consisting of the method **computeBloomEffect** and the if statement in **renderRayTracing.**

2.2.5 Reference

LearnOpenGL CN:

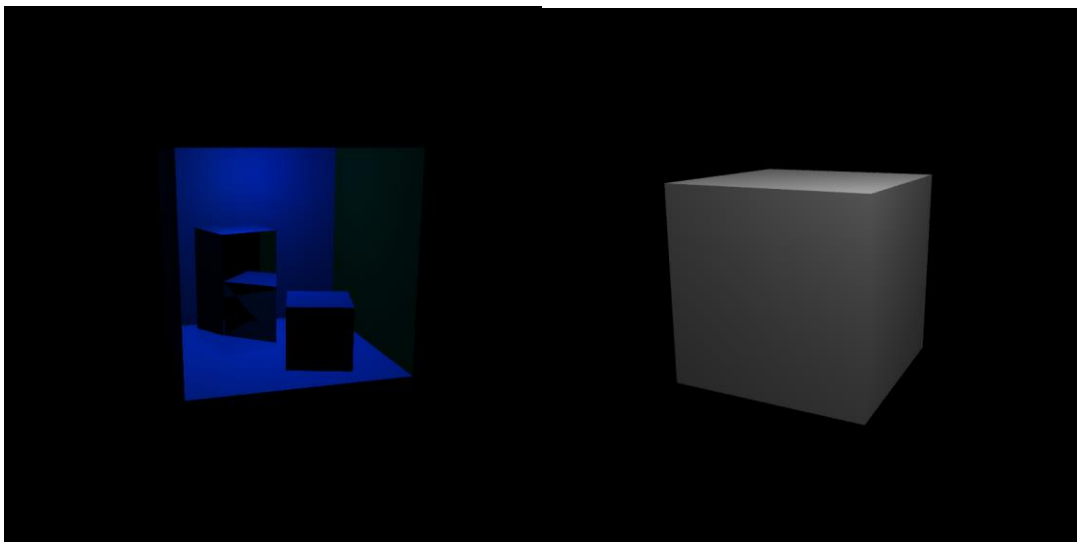
https://learnopengl-cn.github.io/05%20Advanced%20Lighting/07%20Bloom/#_1

2.3 Cast multiple rays per pixel with irregular sampling

2.3.1 Description

To make the edges smoother we can cast multiple rays per pixel at random position and averaging the results of the final color. This is done by creating 5 rays on the pixel that are shifted from the center by a random value generated using rand. The value between 0 and 1 is added to the pixel coordinates, which are then divided by window resolution, multiplied by 2 after which 1 subtracted, leaving a value between -1.0 and 1.0. For every ray the color is calculated with all the effects, and the pixel color is set to the averaged value.

2.3.2 Rendered Image



Cornell box with parallelogram lights Shading, reflections, transparency and multiple Rays per pixel

Cube with shading and multiple rays per pixel

2.3.3 Visual Debug

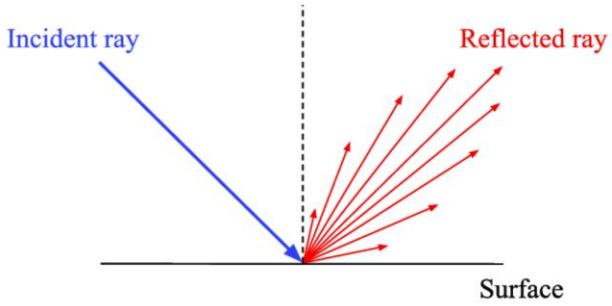
2.3.4 Location

The method is implemented in the renderRayTracing method in render.cpp

2.4 Glossy reflections

2.4.1 Description

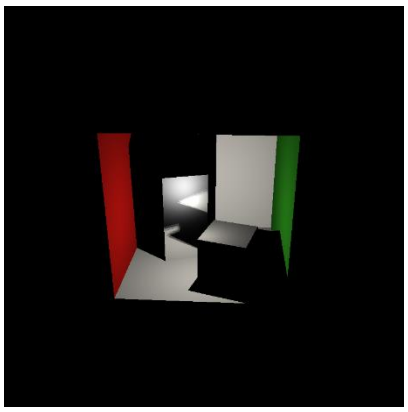
Glossy reflection is a specific reflection which can be illustrated by the picture below:

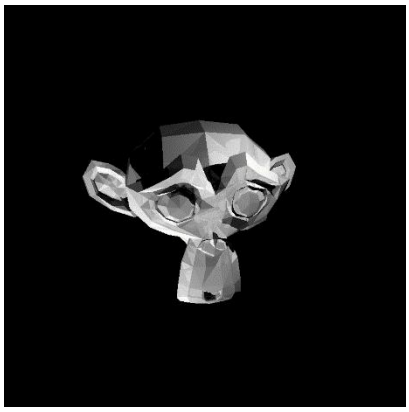


(picture from: https://www.researchgate.net/figure/An-illustration-of-reflection-from-a-glossy-surface_fig1_350121411)

This feature is implemented by building new x, y axis, changing the direction to get the final color by using a normally distributed random number generator and then repeat this process for many times in order to get the final color which is the sum of the color calculated in each epoch and then divide it by the sampleTime.

2.4.2 Rendered Image



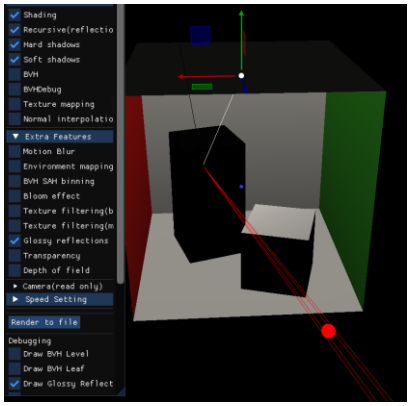


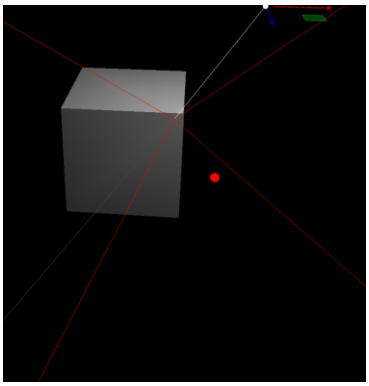
(the picture rendered for glossy reflection)

2.4.3 Visual Debug

I choose to get a red sphere in the center of the rays to indicate the direction of happening glossy reflection.

Need to open glossy reflection in Extra Features and the drawGlossyReflection in debug.





(The screen shot when opening visual debug of glossy reflection)

2.4.4 Location

This method is implemented in **getFinalColor** in **render.cpp.**

2.4.5 Reference

1. https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwiA0q6HxY_7AhWzxgIHHXmYCp8QFnoECBkQAQ&url=https%3A%2F%2Fwww.oreilly.com%2Flibrary%2Fview%2Fray-tracing-from%2F9781498774703%2FK00474_C025.xhtml&usg=AOvVaw0ZVOnEDn7Io75pgQufwo2q

2. https://gamedev.stackexchange.com/questions/105807/ray-tracing-glossy-reflection

3. https://www.bilibili.com/video/BV18E41117hB/?spm_id_from=333.337.search-card.all.click&vd_source=b16e8efcf0621ac05863d68d4f22107a

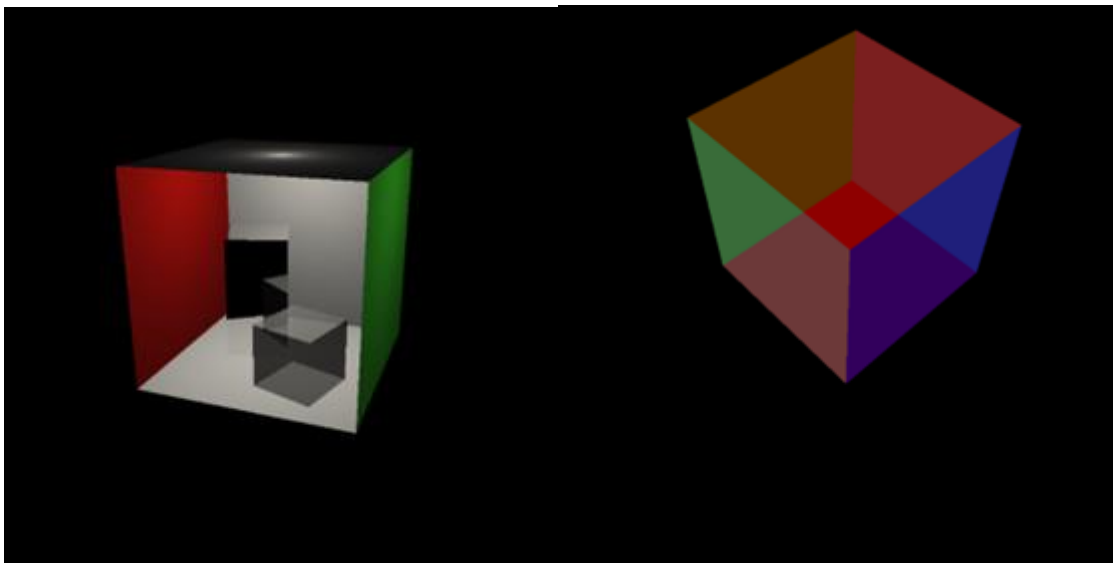
4. https://stackoverflow.com/questions/16153589/generating-a-uniform-random-integer-in-c

2.5 Transparency

2.5.1 Description

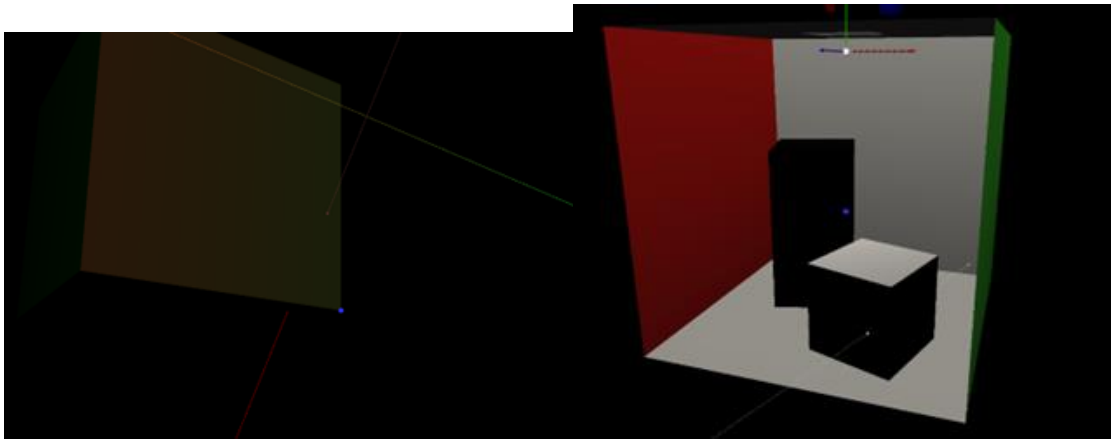
Transparency is implemented by using alpha blending as shown in the lectures. If a ray hits a transparent surface then a new ray is computed starting from the intersection point and used to calculate the color of the surfaces behind the first one until the max depth of 3 is reached. At that point the color is calculated using the formula of transparency * color + (1 – transparency) * colorBehind. The visual debug of transparency is drawing the rays and intersection points used in determining the colors of the surfaces behind the first.

2.5.2 Rendered Image



Cornell box with reflection and transparent cube Cube with segment lights

2.5.3 Visual Debug



The rays and intersection points used for determining color in the same scenes as above

2.5.4 Location

This is implemented inside the getFinalColor method in render.cpp.

3.1 Performace Test

	Cornell Box	Monkey	Dragon
Num triangles	32	967	87130
Time to create	0.5438 milliseconds	1.3296 milliseconds	154.601 milliseconds
Time to render	65.7986 milliseconds	160.529 milliseconds	463.624 milliseconds
BVH levels	3	5	9

Max tris p/ leaf node	4	31	171