Group Number 142

Paulina Sobocińska (5457033)

Matei Tudor Ivan (5537371)

Junwon Yoon (5024722)

Basic Features

Shading using Phong Illumination Model

Sources:

i. Lecture Notes from Lecture 04 - shading
ii. https://en.wikipedia.org/wiki/Phong_reflection_model

Description:

shading.cpp file

Inside the computeShading() function, first I calculate the light and camera vector. Then I calculate the diffuse term of the Phong model using the formula given in the lectures. When the dot product of the light vector and normal is <= 0, I set the diffuse term to 0. The specular term is also implemented using the formula given in the lectures, here I check if the dot product between the reflection vector and the camera vector is <=0, and if yes – I set the specular term to 0. Then I return the sum of the specular and diffuse term.

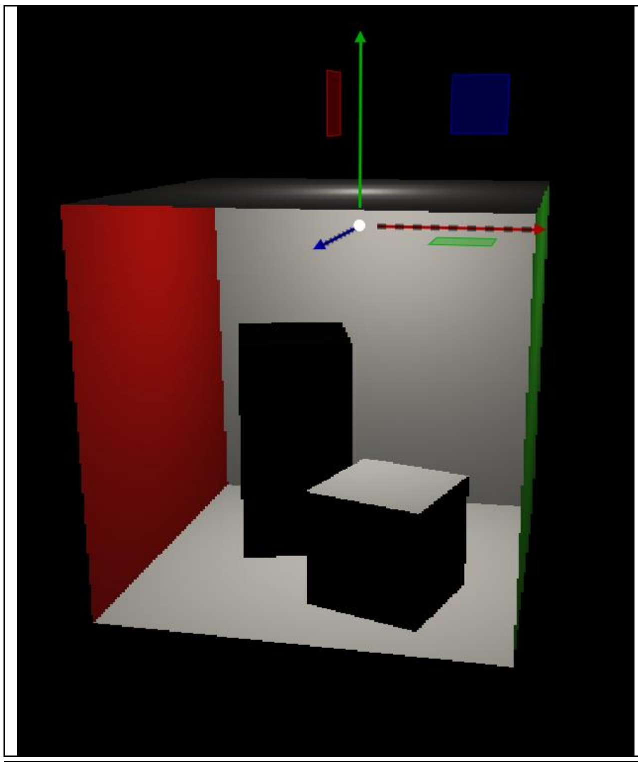
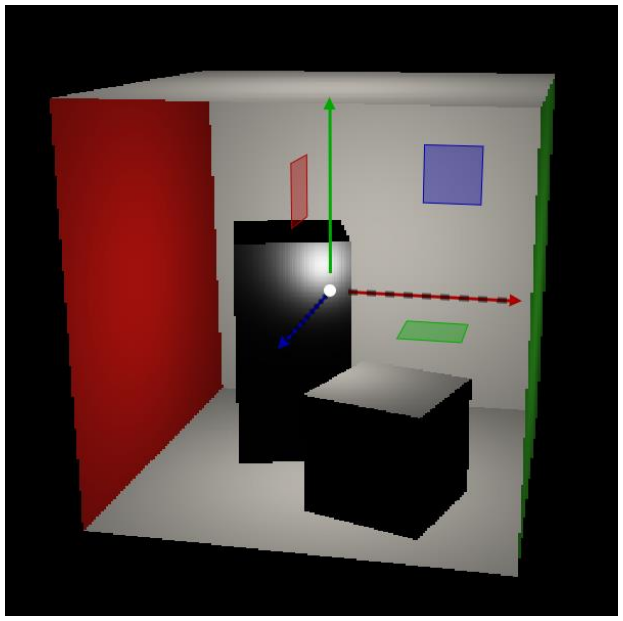
light.cpp file

Inside the computeLightContribution() function, I call the computeShading function and add the result to the final vector returned by computeLightContribution() (i.e., for every given point I just compute what the shaded color would be). The returned vector is initialized to 0, and after computing the shading, I check if any of the values are bigger than 1, and if yes, I set them to 1.

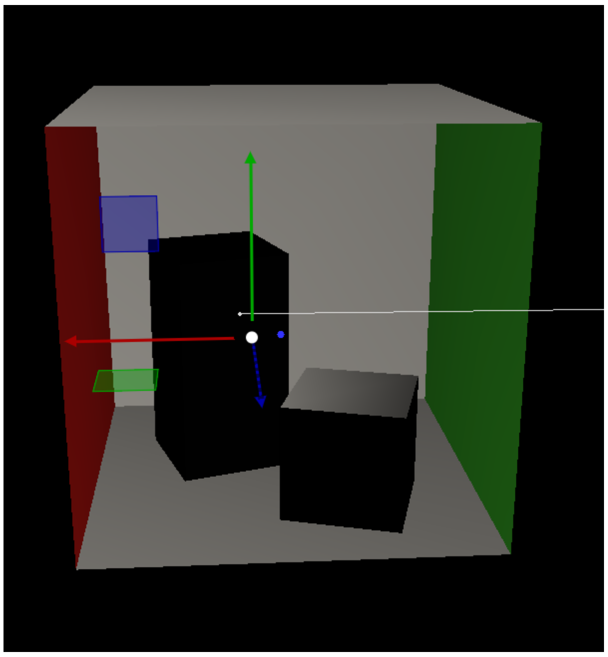
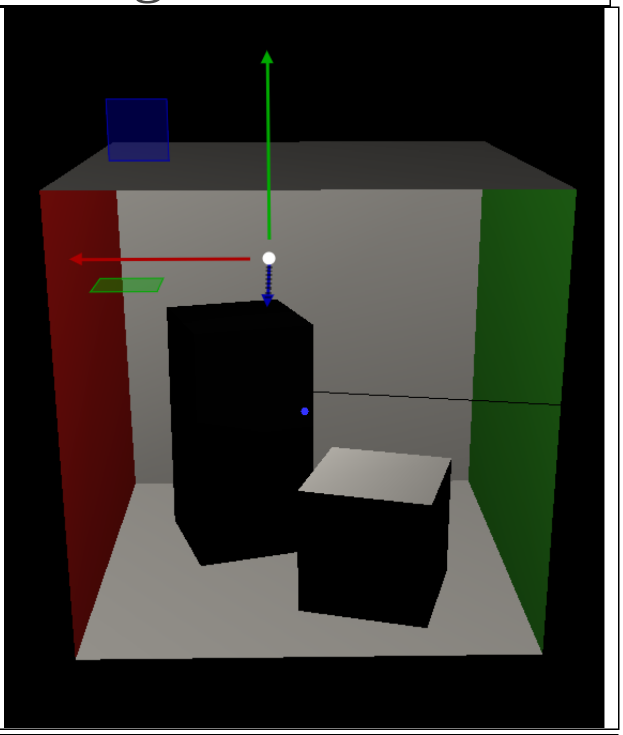
Visual debug

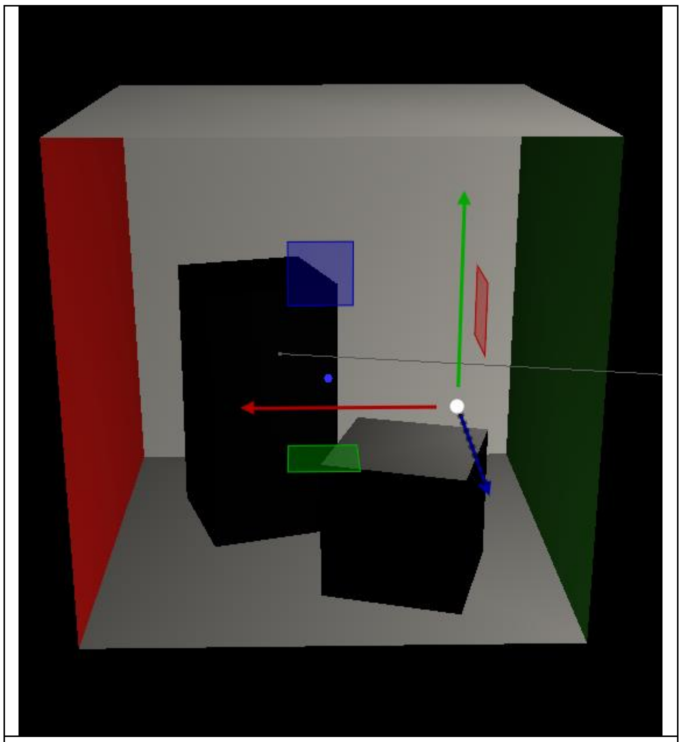
In render.cpp in getFinalColor, after calculating the final color vector, I just draw the given ray in the calculated color, so I can check what the color at this point will be.

Rendered Images

	
Shading when light is in default position	Shading with light in front of the mirror

Visual Debug

	
Light source near the ray – the color is white	Light source far away from the ray – the color is black


Light source a little bit far away – the color is grey

Recursive ray-tracers

Sources:

i. https://math.stackexchange.com/questions/13261/how-to-get-a-reflection-vector

Description:

shading.cpp file

Inside the computeReflectionRay(), I calculate what the reflected ray will be, relying mostly on the reflection vector formula from my source.

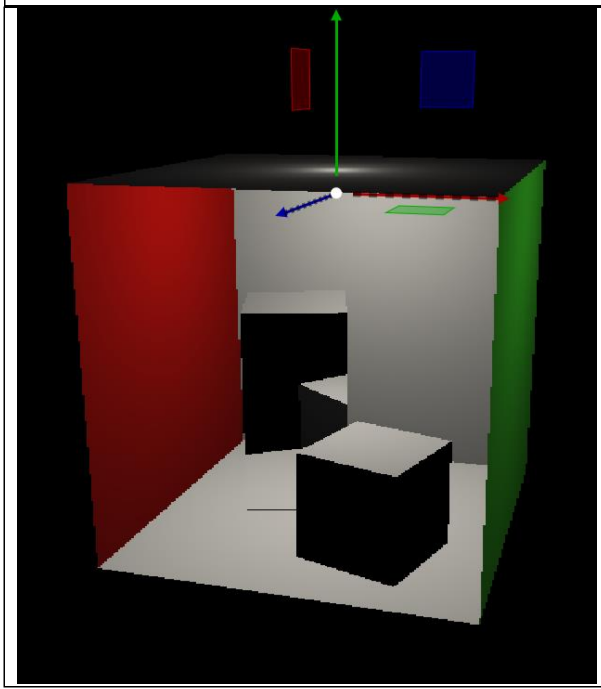
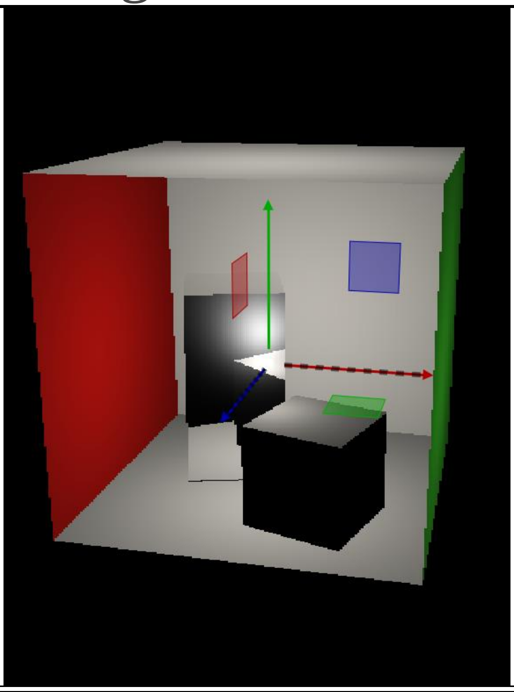
render.cpp file

Inside the getFinalColor() method, if the recursive ray tracing is enabled, rayDepth < 2 and hitInfo.material.ks is not black, I compute the reflected ray by calling computeReflectionRay() function. Then, still inside the if statement and still inside getFinalColor(), I call getFinalColor() with the reflected ray, and add the result to the returned vector. Since I check the rayDepth in the if statement, this is my recursion base case – when I call getFinalColor() recursively, I just increment rayDepth by 1.

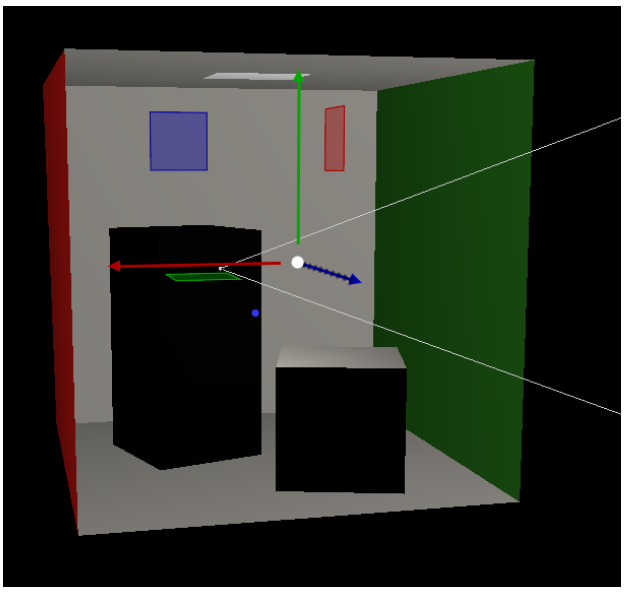
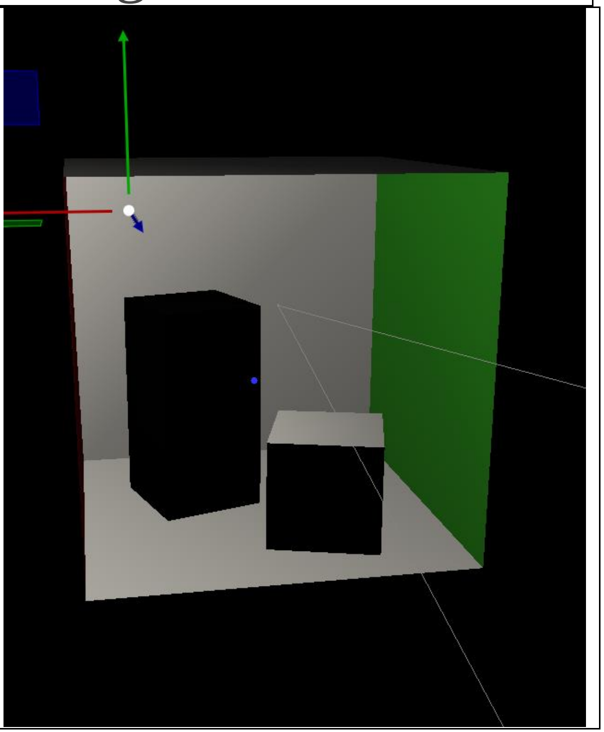
Visual debug

Also in render.cpp in getFinalColor(), after calculating the final color, if the recursive ray tracing is enabled, I draw the reflected ray in the final color.

Rendered Images

	
Shading + recursive ray tracing	Shading + recursive ray tracing

Visual Debug

Drawing the reflected ray	Drawing the reflected ray

Hard shadow

Sources:

i. Lecture Notes from Lecture 07 – shadows
ii. https://artisticrender.com/how-the-light-path-node-works-in-blender/

Description

light.cpp file

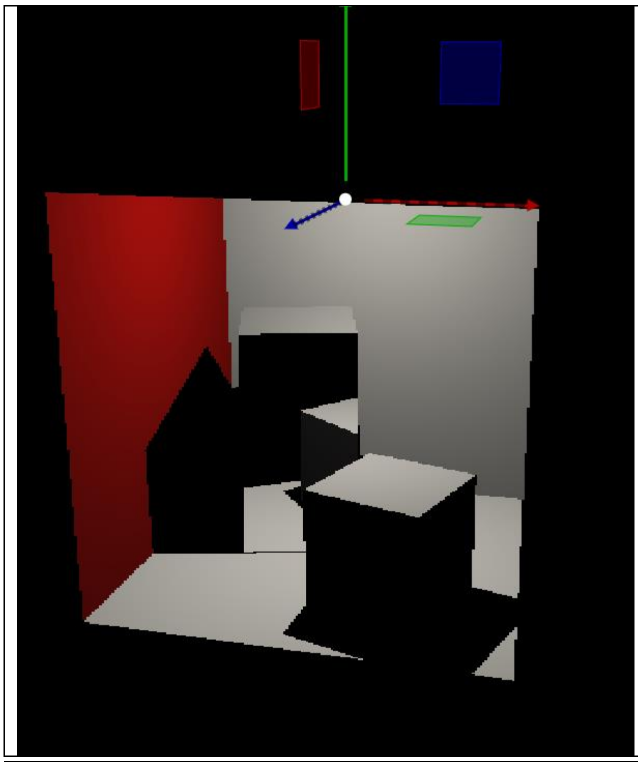
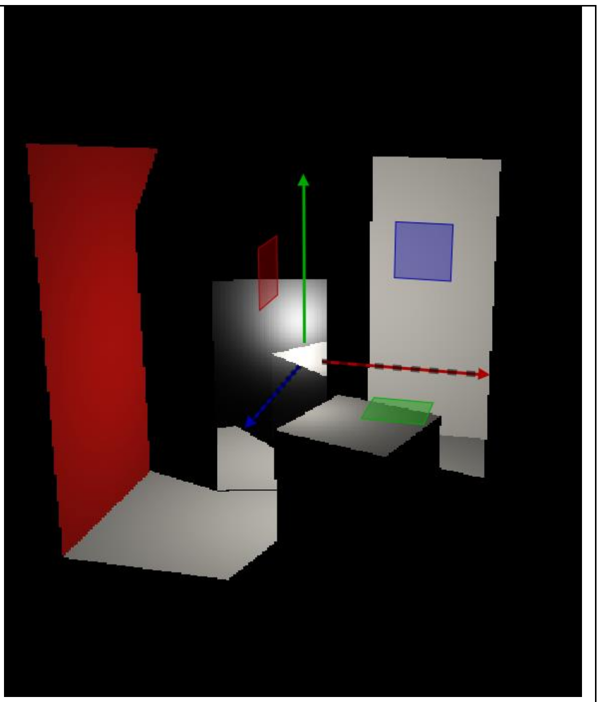
Inside the testVisibilityLightSample() function first I check if the feature is enabled, and if not, I return 1, so every point is "visible". Then I compute the shadow ray, the origin is "t + origin * direction – offset * direction". The direction is normalized light position – origin, and t is the length of light position – origin. To check if the shadow ray hits something before it gets to the light source, I use the bvh.intersect method. If it returns true, so the shadow ray hit something, I return 0 (this point is not visible). If it does not hit anything, I return 1.

Then I call this function inside computeLightContribution() function. I made an if statement that checks if testVisibiltyLightSample returns 1, and if yes, then I compute the shading and add it to the returned vector. Since I already checked if the hard shadows are enabled in the testVisibilityLightSample function, I don't have to check it again in computeLightContribution.

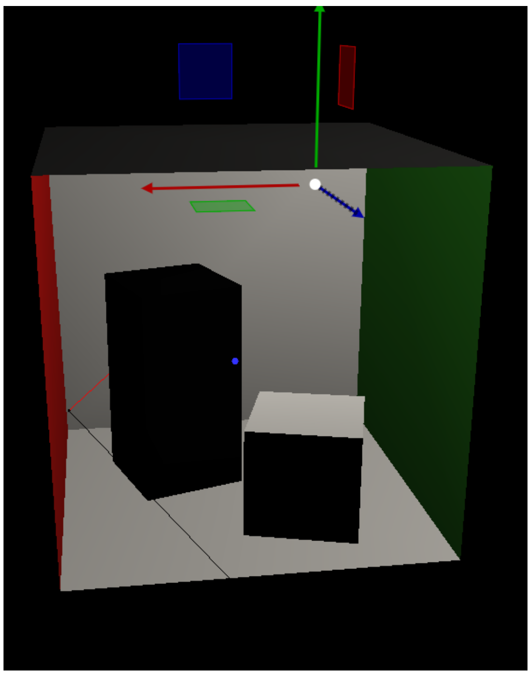
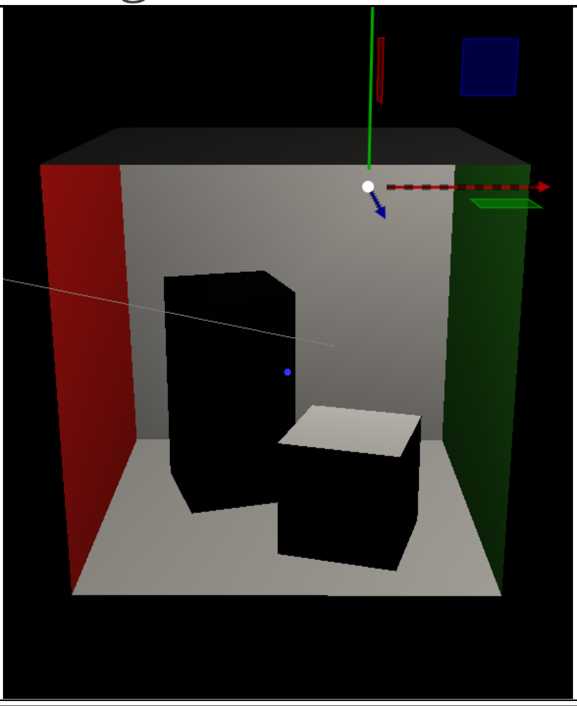
Visual debug

In light.cpp in computeLightContribution, if hardShadows are enabled I compute the shadow ray for the given ray. I check if it hits something by using bvh.intersect, and if it returns true, so the shadow ray hit something other than light, I draw it in red color.

Rendered Images

	
Shading + recursive ray tracing + hard shadows	Shading + recursive ray tracing + hard shadows

Visual Debug

	
Drawing a shadow ray – this point is not visible	This point is visible

Area lights

Sources:

I. Fundamental of Computer Graphics, 4th Edition, p. 299 - 308
II. Lecture Notes from Lecture 11 - Acceleration Data Structures

Description:

Visual Debug Button: Debug Soft Shadow

The visual debug only works if the soft shadow and shading is enabled!

main.cpp file

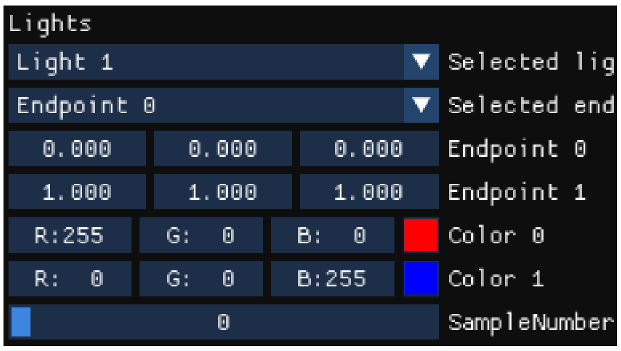
We create 2 new sliders, named ‘SampleNumber’, that can take an integer value between 0 to 1000. They are placed under the categories of segment light and parallelogram light. The input value is passed as an integer value ‘numberofarealights’

light.cpp file

Before any methods, we receive the integer value ‘numberofarealights’ as a form of extern int, so that we could use it from different classes

And then we define two interpolation methods ‘linearInterpolation’, and ‘bilinearInterpolation’, which takes the position vectors and light vector to perform linear interpolation for segment lights, and bilinear interpolation for parallelogram lights

Then we define two computing methods, ‘computeSegmentLight’, and ‘computeParallelogramLight’, which does the computation of the averaged color, as well as the visual debug in one place. First, about the sampling, we call the sample methods (which the explanations will be given below) multiple times to sample a light from a random position. On every sampling method call, the color and the position will be updated, so we sum every color, and divide it with the number of the samples to get an average. The return value will be this average value. The number of samplings would be done is given by the user, by the slider ‘SampleNumber’. You can find it at the bottom of the lights section in the UI, shown on the image below

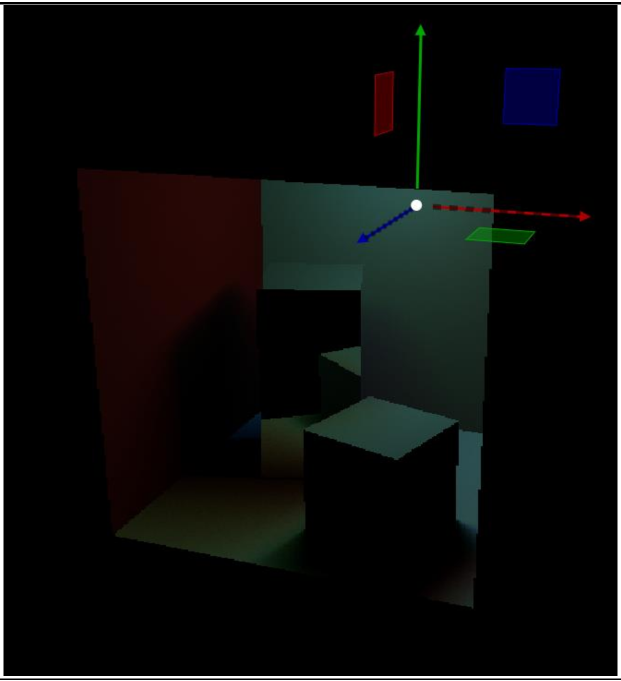
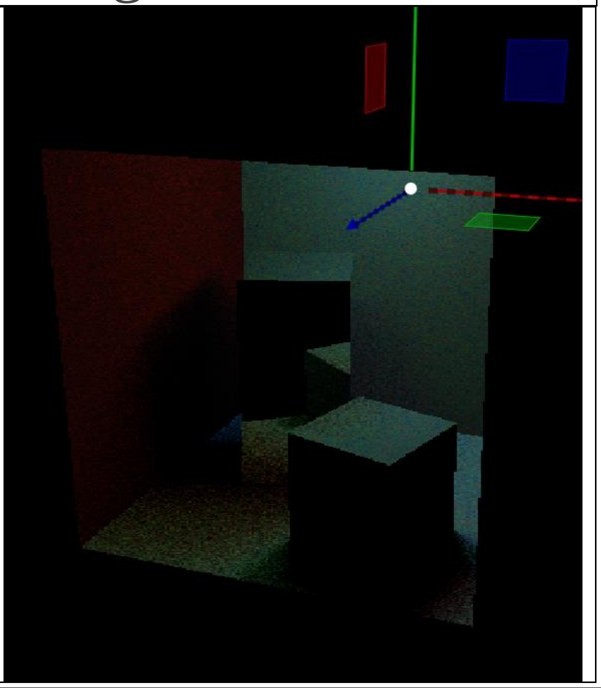
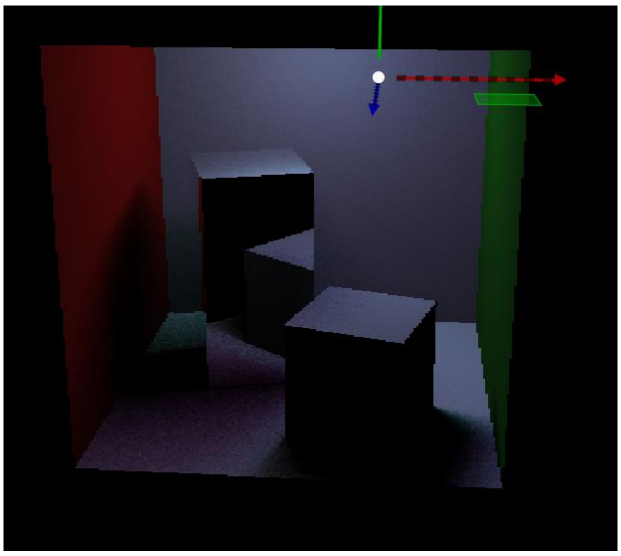
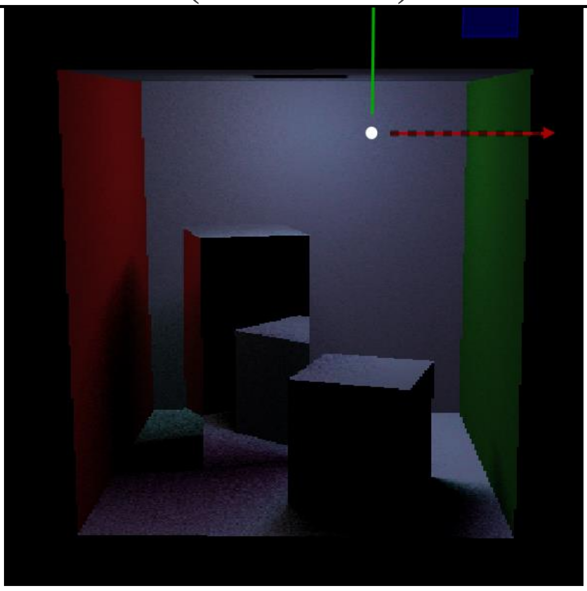


```
Lights
Light 1                                   ▼  Selected lig
Endpoint 0                                ▼  Selected end
   0.000        0.000        0.000           Endpoint 0
   1.000        1.000        1.000           Endpoint 1
  R:255       G:   0      B:   0     🟥      Color 0
  R:   0      G:   0      B:255      🟦      Color 1
                    0                        SampleNumber
```

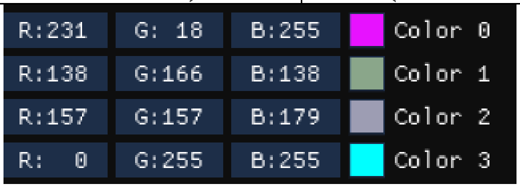
With the updated position, we change the t value of the shadow ray we’ve defined earlier. We shoot the rays in the selected position; in a way we can see the multiple rays from the area light reaching the position.

In the sampling methods ‘sampleParallelogramLight’ and ‘sampleSegmentLight’, we randomly sample points from the area light and update its position and color. We do this by using ‘static_cast<float>’. The reason we chose random sampling over uniform sampling is because random sampling will show small wiggles, especially in when there are small number of shadow rays. This will help us see dramatic differences when there are different numbers of samples.

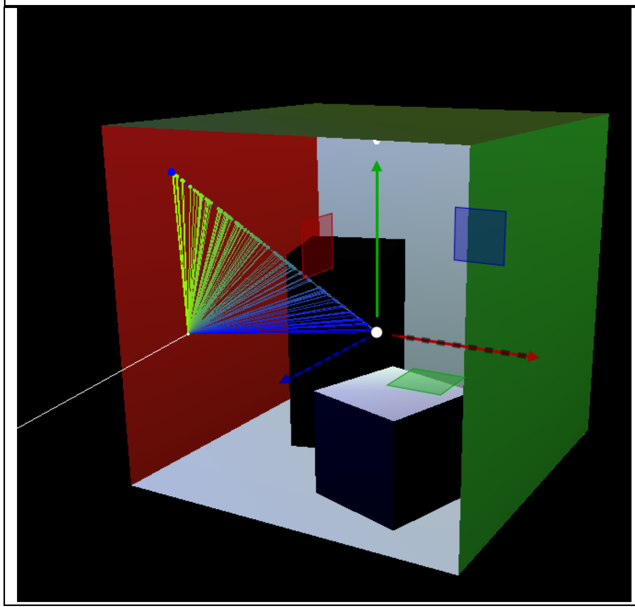
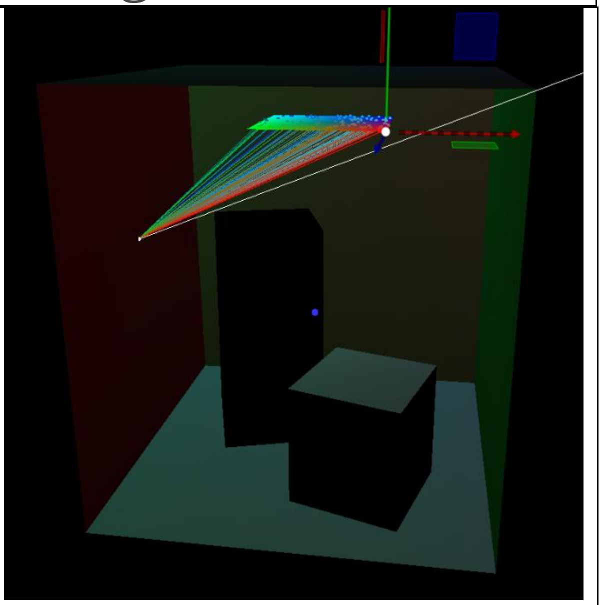
Lastly, we add the computed value we’ve calculated in the method ‘computeLightContribution’. We call the functions ‘computeSegmentLight’ and ‘computeParallelogramLight’ and add the results to the return vector.

Rendered Images	
	
Soft shadows with 1000 shadow rays (Default color)	Soft shadows with 12 shadow rays (Default color)
	

Soft shadows with 100 shadow rays (Different color combination)	Soft shadows with 50 shadow rays (Different color combination)



R:231	G: 18	B:255	🟪	Color 0
R:138	G:166	B:138	🟩	Color 1
R:157	G:157	B:179	⬜	Color 2
R: 0	G:255	B:255	🟦	Color 3

Visual Debug	
	
Visual debug on segment light	Visual debug on parallelogram light

Acceleration data-structure generation

Sources:

i. Fundamental of Computer Graphics, 4th Edition, p. 299 – 308
ii. Lecture Notes from Lecture 9 - Acceleration Data Structures

Description:

bounding_volumne_hierarchy.h file

We define two structs, tmindex and Node

Struct ‘tmindex’ contains the indices for the triangle and mesh, namely

- ‘trindex’, an integer value that contains the index for triangle
- ‘mindex’, an integer value that contains the index for mesh

Struct ‘Node’ contains the parameters for the constructor, namely

- ‘indices’, which is a vector of trindex, the struct we defined earlier
- (Here we made every node have the indices of the meshes and triangles they contain, because we created a complete and full binary tree to keep the nodes; this binary tree is useful because we can easily access the children of a node, and we would insert nodes at their correct index, for the tree to actually be binary, full and complete; it is then easier to work with the nodes; this does not affect in any way the bvh, it works as intended, and it has fast creation and rendering times; also by keeping indices, they do not get mixed up when sorting centroids to then slice, and it)
- ‘aabb’, an Axis-Aligned Box of the node
- ‘leaf’, a boolean that shows whether the node is leaf or not (each node is a considered a leaf at first, after the split function ends, we set the Boolean to false, depending on whether it is a leaf or not)

bounding_volumne_hierarchy.cpp file

Main method: BoundingVolumeHierarchy

To begin with, we define two parameters

- A root node, which will contain all the triangles and meshes and an AABB which contains everything
- ‘alternatingMedian’, an integer value used to decide which axis we’re going to slice our AABB

Then we initialize a dummy binary tree, using the method ‘fakeBinaryTree’, each dummy node being defined as having specific values for its AABB. We set the maximum depth to be 16 to avoid excessive computation. We make a nested for loop that goes over all the indices of meshes and triangles. Inside the loop, we obtain the bounds of the triangles using ‘mmin’ and ‘mmax’ method defined inside the file and initialize a tmindex struct with their fields given as the indices of the loop and push it to the root tree. With the minimum and maximum bounds computed, we initialize a new AABB, and put it as the field for the root node. Here we pass the parameters AABB, node, and the vector of meshes, and some initial values as the parameter to our helper method ‘slice’

Helper Method: slice

The helper method ‘slice’ takes 6 parameters as its input parameter

- ‘aabb’. the Axis-Aligned Bounding Box
- ‘alternatingMedian’, an integer value used to decide which axis we’re going to slice our AABB
- ‘node’, our current node
- ‘level’, integer value that stores the current level inside the tree
- ‘nodeInd’, the index of our current node inside the tree

We create a centroids vector, that will be filled with the centroids of triangles. We also insert the node that we are working with in the nodes tree at the index given as parameter, nodeInd.

At this point we set up the conditions for the recursion. More specifically, when to stop the recursion:

1. If the size of the indices vector is 0, 1 or 2

2. If the size of the indices vector is more than 2, but the level is 15

Else, if the size of the indices vector is larger than 2, but the level has not reached 15 yet, we continue the recursion. Inside the else clause, we fill the ‘centroids’ vector that will store the vertex of the centroid, as well as the index of the corresponding triangle and mesh, using struct tmindex. Then, using a for-loop, we iterate over all the triangles in the current node, calculate the centroids for each triangle, and push them to the ‘centroids’ vector we created just before. The way to calculate the centroid is simply adding the coordinate values (x, y, z) for each vertex, and dividing them by 3. We then decide which axis we want to slice the scene, using the ‘alternatingMedian’ integer. We check the remainder of ‘alternatingMedian’ divided by 3, and

- If remainder = 0: x
- If remainder = 1: y
- If remainder = 2: z

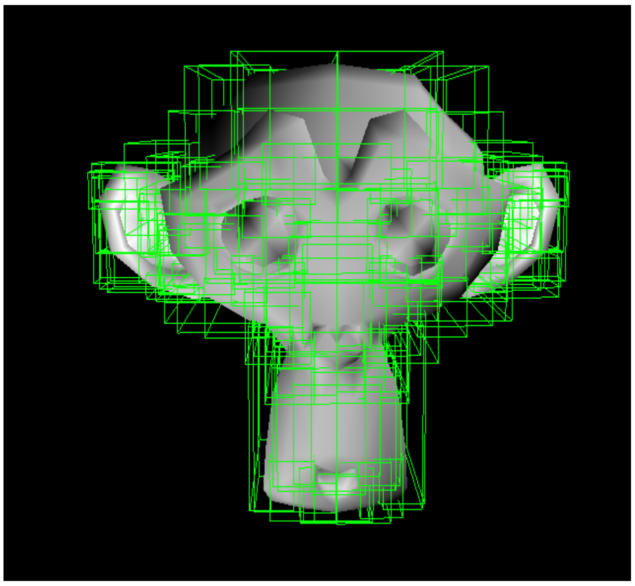
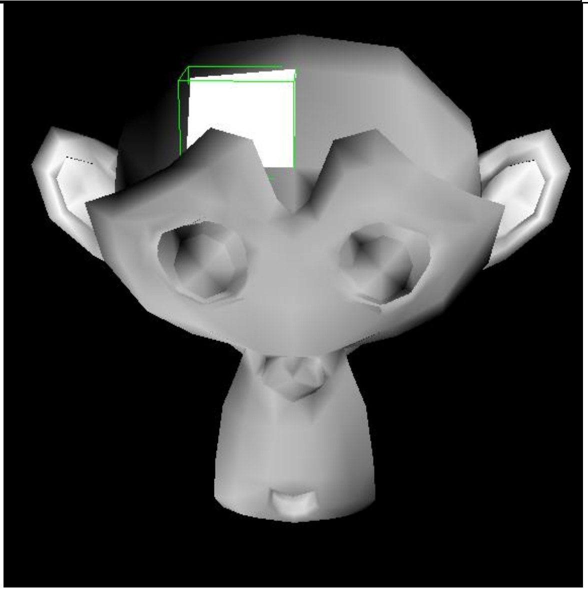
will be the axis chosen

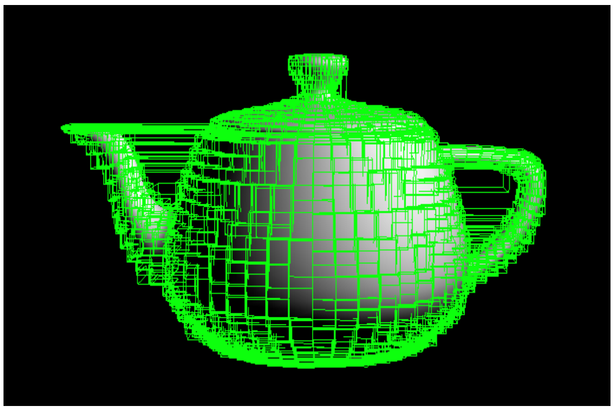
We will increment our ‘alternatingMedian’ by 1 if it has remainder 0 or 1, and then subtract 2 if it has remainder 2, so that the axes would take turns for slicing the scene. By the axis chosen, we will sort the triangles inside the vector ‘centroids’. For example, if x was chosen, we will sort the triangles depending on the x value. The sorting algorithm we’ve chosen is quick select which has a time complexity of O(n log n).

Now, we create two new nodes, namely ‘c1’ and ‘c2’, which will be the node representation of the two sliced scenes. The first half of the centroid will be pushed into ‘c1’, and the other half to ‘c2’. We make use of the ‘mmax’ and ‘mmin’ functions again to get the bounds for c1 and c2 and create a new Axis-Aligned bounding boxes for each. Now we make the recursion call, once for ‘c1’, and once for ‘c2’. The ‘level’ will be incremented by 1 for both nodes, while the value ‘nodeInd’ will be (nodeInd*2 + 1) for ‘c1’ and (nodeInd*2+2) for ‘c2’.

To get the number of levels, we check for each iteration of the slice method if the ‘level’ parameter has surpassed m_numLevels (m_numLevels updates, if needed, with each iteration).

The number of leaves is incremented each time the stoppage condition is fulfilled (indices.size <= 2 or indices.size > 2 and level >= 15)

Visual Debug	
	

Monkey with level 8	Monkey with 494th leaf
	
Teapot with level 11	Teapot with 2721st leaf

Acceleration data-structure traversal

Sources:

iii. Fundamental of Computer Graphics, 4th Edition, p. 299 – 308
iv. Lecture Notes from Lecture 9 - Acceleration Data Structures

Description:

Visual Debug button: Debug BVH Traversal, Debug BVH Traversal Not Visit

The visual debug only works if BVH is is enabled!

Both visual debugs can be seen separately via checkboxes and sliders

bounding_volumne_hierarchy.cpp file

Main method: intersect

To begin with, we make a list that would contain all the AxisAlignedBox the ray intersects, and a list that contains the AxisAlignedBox in which the ray only hits one side of its child, and then makes the initial call

Helper method: intersect1

First, we define the boolean ‘hit’, which will be our final return value. And then we create two rays, ‘temp’ and ‘temp2’ copied from the input ray, since if we call the intersectRayWithShape method, it’ll change the t values of the ray. We input the AxisAlignedBox and ray ‘temp’ to compute intersectRayWithShape, and if the result is false, it means the ray doesn’t hit the AxisAlignedBox, so we return false. If they do hit, push this AxisAlignedBox to the list.

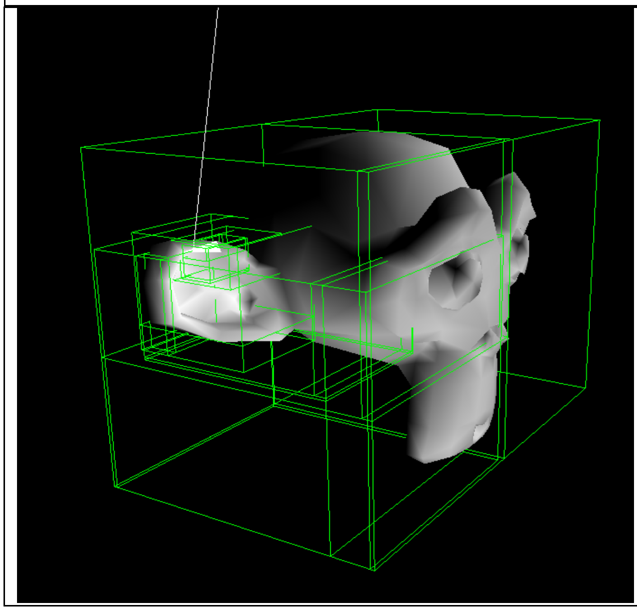
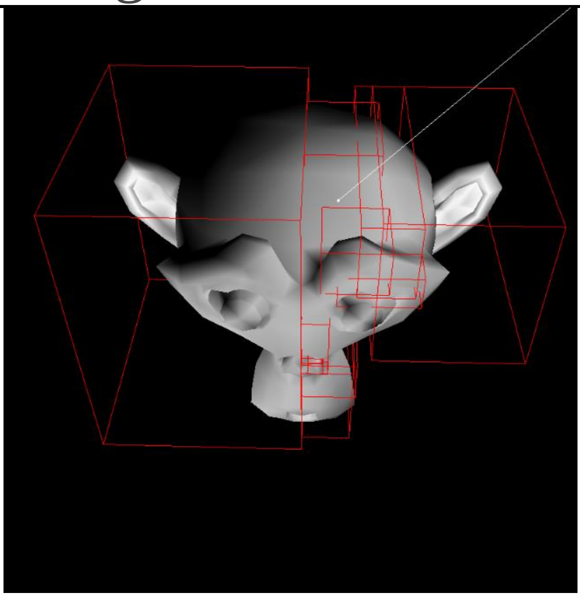
Now if the node that we have is a leaf, we go over all the triangles of all the meshes, and if the ray hits the triangle, update its hitInfo, change our return boolean value ‘hit’ to true. This is the part where we also draw the triangle and AxisAlignedBox for visual debugs. We use the calculated x, y, z vertices to draw a triangle, and we draw every AxisAlignedBox from the list that we kept. This will draw the triangle that we intersected, and the AxisAlignedBoxes that the ray has intersected along the way.

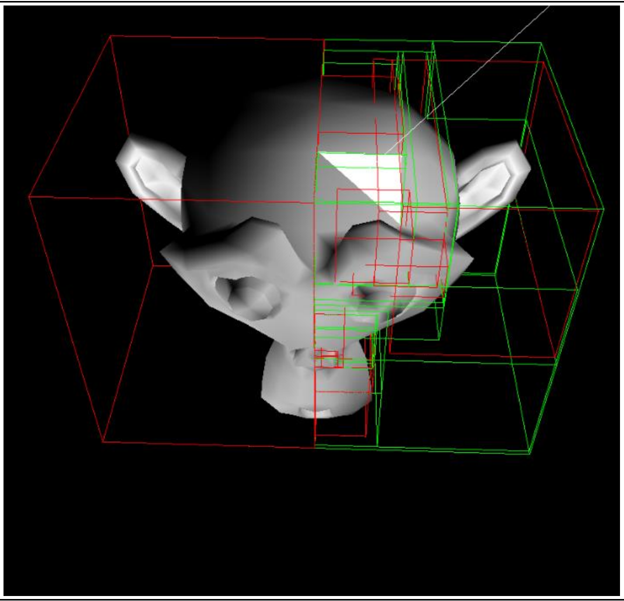
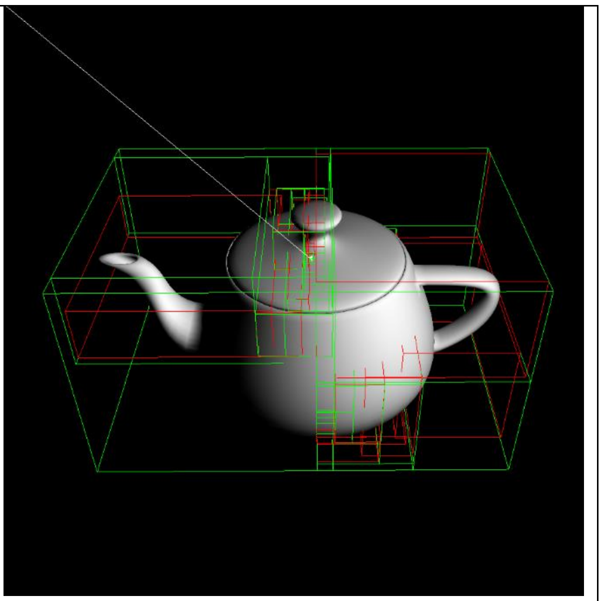
Else if the node that we have is not a leaf, we again set the rays from ‘temp’ up to‘temp6’ copied from the original input ray. For each ‘intersectRayWithShape’ we use one temp rays each. Again, the reason we set the temporary rays is to avoid the ‘intersectRayWithShape’ changing the t values of the ray.

First, we check the ray hits only one child. That is, ray goes through left child’s AxisAlignedBox, BUT doesn’t go through right child’s AxisAlignedBox, or the other way around. We push all the AxisAlignedBoxes that didn’t hit the ray, into the vector ‘notlist’. This list of AxisAlignedBox is drawn when the ray hits the triangle, thus drawing all the intersected but not visited nodes.

We also check if the rays intersect on the left child box, OR right child box. If they do, we make recursive calls on each child nodes, and do a or operation with the output boolean ‘hit’. Now we return the final return boolean ‘hit’. The visual debugs are shown below.

For visual debug, Debug BVH Traversal button will draw all the AxisAlignedBox that the ray has intersected in color gree. Debug BVH Traversal No visit button will draw all the intersected but not visited AxisAlignedBox in color red.

Visual Debug	
	
Visual debug on Monkey (Show intersected AABB)	Visual debug on Monkey (Show intersected but not visited AABB)

	
Visual debug on Monkey (Show both visual debugs)	Visual debug on Teapot (Show both visual debugs)

Normal interpolation with barycentric coordinates

Sources:

i. https://mathworld.wolfram.com/BarycentricCoordinates.html
ii. https://en.wikipedia.org/wiki/Barycentric_coordinate_system

Description

interpolate.cpp file

Inside computeBarycentricCoord() function, first I calculate the whole area of the given triangle (made of the given vertices). I used formulas from my sources to calculate the area. Then I repeat the process, but for the smaller triangles made of two of the given vertices and the given point. To get the barycentric coordinates, I just divide the smaller triangle's area over the big triangle's area. I do it for two triangles, and since barycentric coordinates all together have to be equal 1, the third barycentric coordinate is just (1 – coord1 – coord2). It is stated that the given point is already inside the triangle, so I don't check if the computed barycentric coordinates are smaller than 1.

Inside interpolateNormal() function, I just multiply each given normal by the corresponding barycentric coordinate, which results in one interpolated vector.

bounding_volume_hierarchy.cpp

In the intersect function, I saved the triangle vertices in my local variables. Then I made an if statement that checks if there was a hit and if the interpolation is enabled. If yes, I compute the intersection point using the given ray, then I compute barycentric coordinates by calling computeBarycentricCoord() with the vertices' positions and the intersection point. I compute the interpolated normal by calling interpolateNormal with the given vertices' normals and computed barycentric coordinates, and then normalizing it. Then I created an "interpolated ray" where the origin is the intersection point and the direction is the interpolated normal. I draw this ray in green, this is my interpolated normal. For every given vertex, I also draw their normal in blue.

Texture mapping

Sources:

i. Lecture Notes from Lecture 6 – Textures

Description

interpolate.cpp file

Inside interpolateTexCoord() I multiply each given coordinate by the corresponding barycentric coordinate.

texture.cpp file

Inside acquireTexel(), first I calculate the corresponding x and y coordinate of the pixel. The x coordinate is texCoord.x * image.width, and the y coordinate is (1 – texCoord.y) * image.height. Then, I compute the index of the pixel by multiplying x * image.height + y. Then I return the pixel at the calculated index.

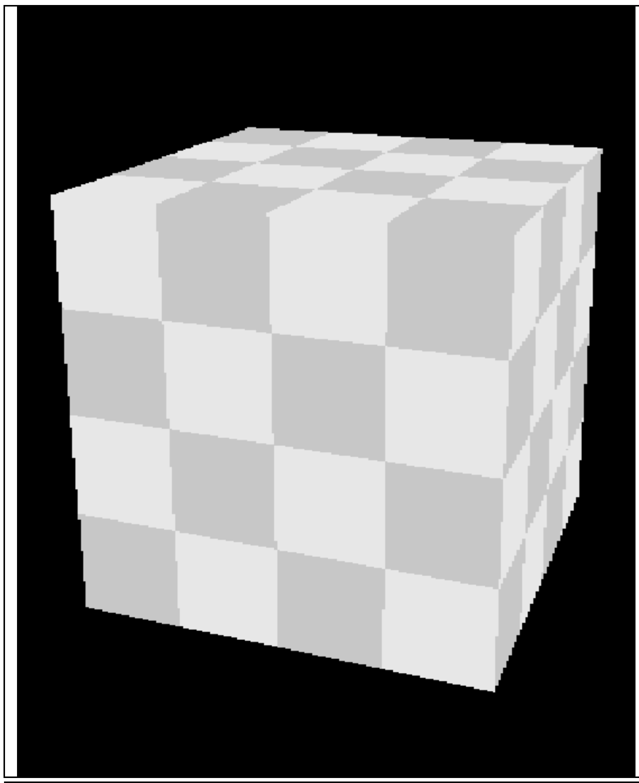
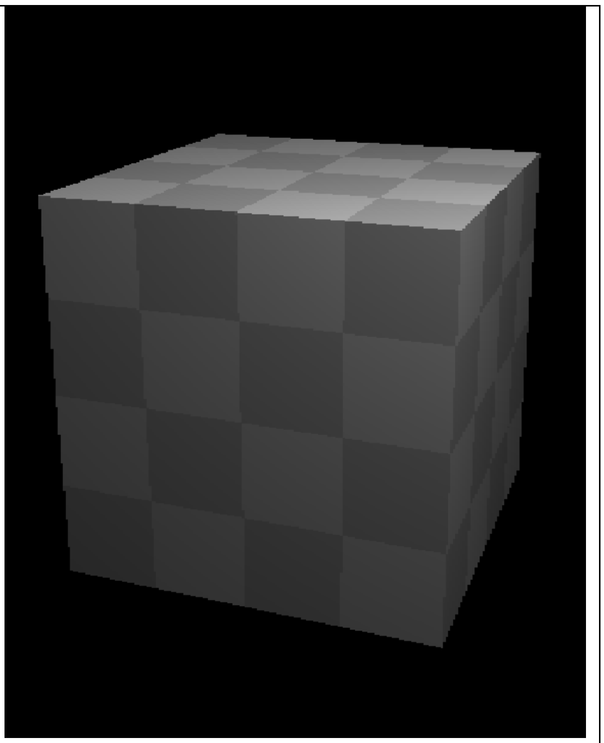
light.cpp file

Inside computeLightContribution(), if the shading is disabled, I created an if statement (I used the if struct that was given in mesh.h in struct Material - if (material.kdTexture)) and I change the hitInfo.material.kd by calling acquireTexel().

shading.cpp file

Since in light.cpp I only change the texture if the shading is disabled, in computeShading() I added an if statement if (material.kdTexture && features.enableTextureMapping), and again I change the hitInfo.material.kd by calling acquireTexel().

Rendered Images

	
Only texture enabled	Texture + shading

Extra Features

Bloom filter on the final image

Sources:

i. Lecture Notes from Lecture 2 – Image and Linear Algebra

Description:

Visual Debug: Show Bloom layer only

The visual debug only works if bloom filter is enabled!

main.cpp file

In main.cpp file, we make two sliders named ‘Bloom Threshold’ and ‘Bloom Distance’. They take inputs from the user, who is responsible for setting the variables. The threshold accepts a value between 0 to 1 in the form of float, and the distance accepts a value between 1 to 100 in the form of int. This input could be used in the form of extern in other classes.

We’ve also made a checkbox named ‘Show Bloom Layer Only’, so that we could separate the bloom layer from the original image.

CmakeLists.txt

We added the directory of bloom files under ‘add_library’ to execute them

render.cpp file

First of all, we include the bloom.h file in order to make usage of the functions.

In render.cpp file, we take the user inputs from the slider we had in main.cpp as the form of extern int and extern float. We define them as ‘threshold’ and ‘dist’, which are thresholds and distances for the bloom filter, respectively. If the bloom effect is enabled, we pass the ‘scene’, ‘threshold’, ‘distance’, ‘windowResolution’, and ‘features’ as the parameters of the function ‘bloom’, defined at in the bloom.h file.

screen.h file

We initialize the function getPixel, in order to retrieve a pixel when given a certain point of the screen.

screen.cpp file

In screen.cpp file, we define the ‘getPixel’ function, in which it takes a two integer values, x and y, and returns the index of the corresponding pixel inside the texture data. The formula to calculate the index was copied from the method ‘setPixel’, which is defined in the same class.

bloom.h file

We include the ‘screen.h’ file in this bloom.h file to make usage of screen.

We define the function ‘bloom’ that takes a Screen ‘screen’, float ‘threshold’, int ‘dist’, glm::ivec2 ‘windowResolution’, Feature ‘features’ as its input.

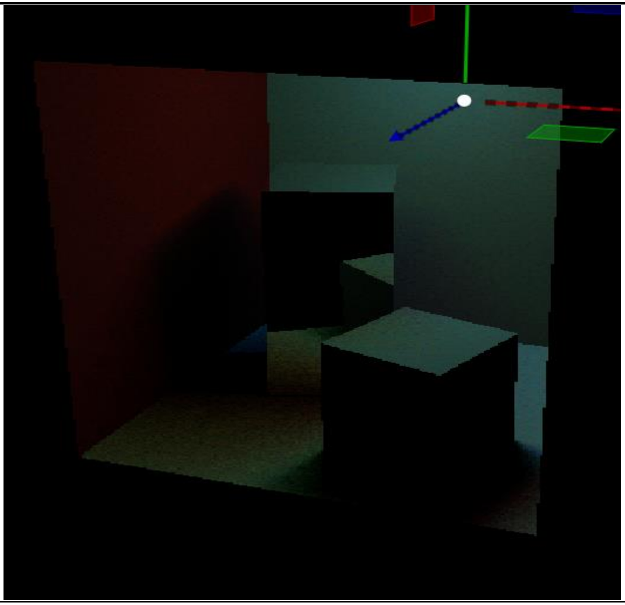
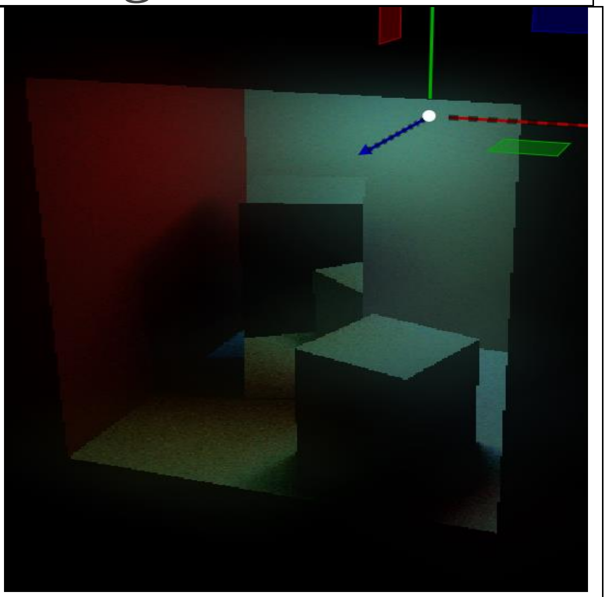
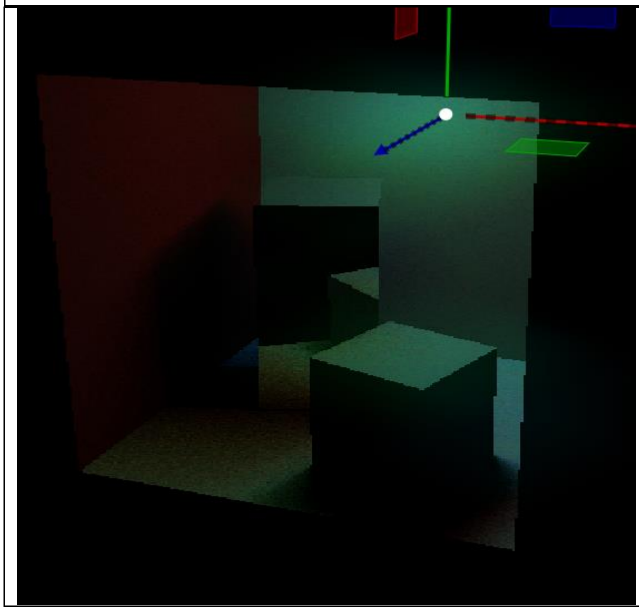
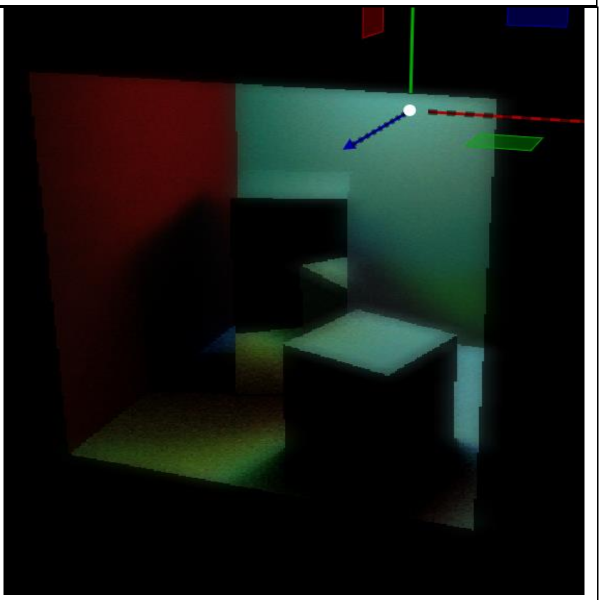
bloom.cpp file

First go over the entire window, using a nested for loop, to go over one and each pixel inside the scene. If their values are smaller than the given threshold, set them to zero. Then push all the pixel into a vector we’ve created, named ‘thresfilter’.

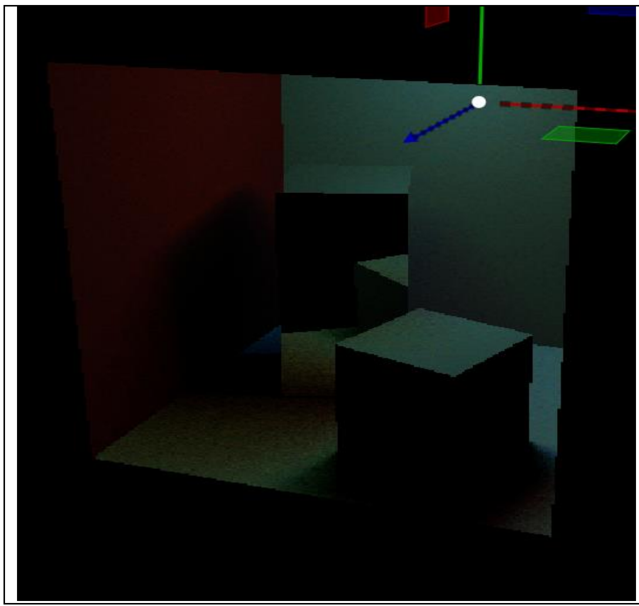
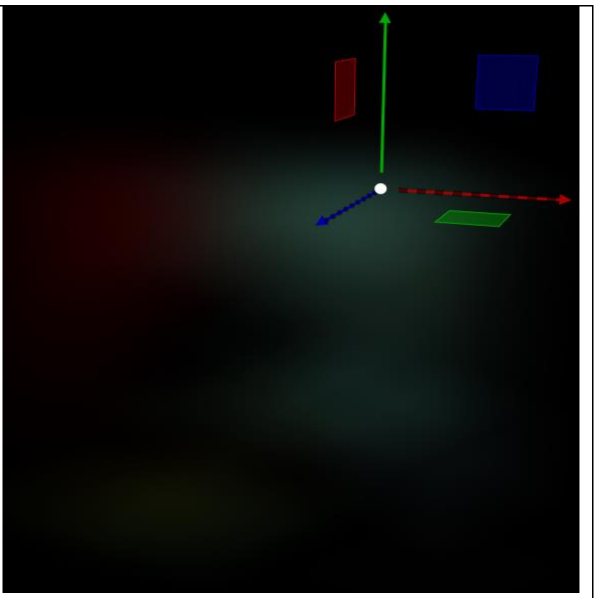
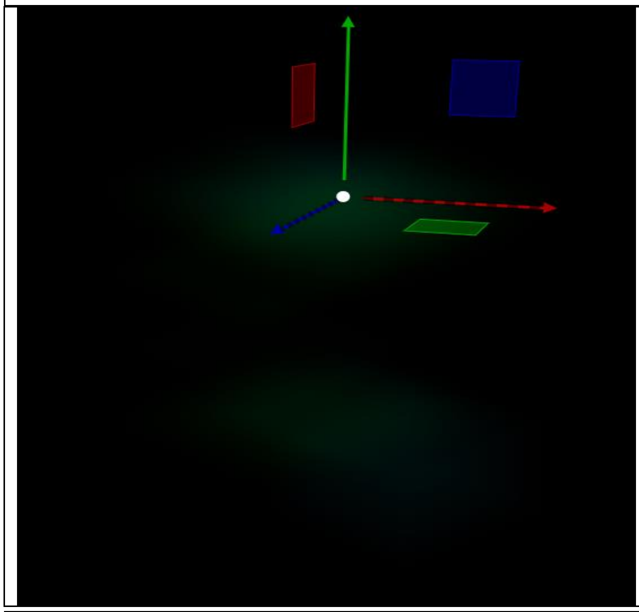
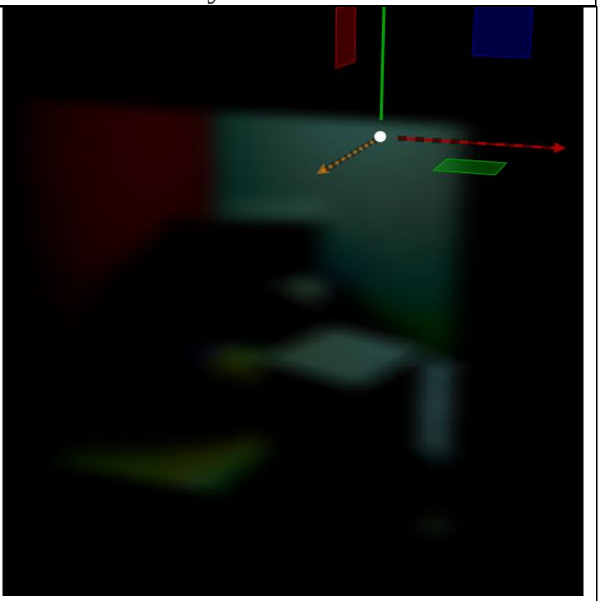
Next, we again go over the entire window using the nested loop. We calculate the new position of the filtered pixels above, using distance. Using the ‘dist’ variable, we define the boundaries of x and y that we want to average. We make another nested loop inside to calculate the average value of the values inside the vector we calculated early. After adding the averaged value to the original image, we return the new value for each and every pixel.

For the visual debug, we've created a checkbox that shows up when we enable 'Bloom effect', named 'Show Bloom Layer Only'. If we click on this, the screen will show only the blurred effect, excluding the original image.

Rendered Image

	
Original Image	Threshold = 0.1, Distance = 50
	
Threshold = 0.25, Distance = 50	Threshold = 0.1, Distance = 10

Visual Debug

	
Original Image	Threshold = 0.1, Distance = 50 Only Show Bloom
	
Threshold = 0.25, Distance = 50 Only Show Bloom	Threshold = 0.1, Distance = 10 Only Show Bloom

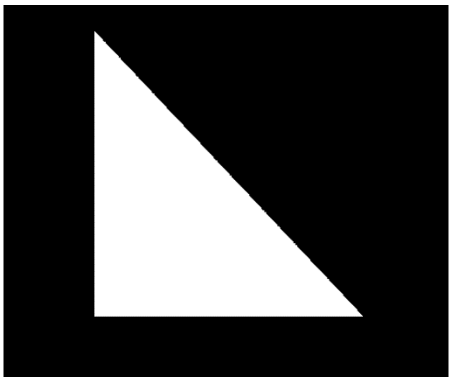
Cast multiple rays per pixel with irregular sampling

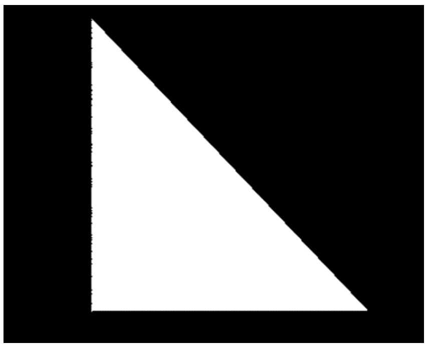
Sources :

i) Fundamentals of Computer Graphics, Fourth Edition (chapter 13.4.1)

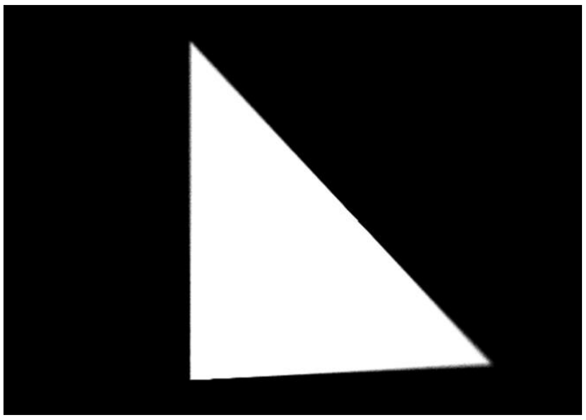
The multiple rays task is implemented in render.cpp. The method in which this task is done is renderRayTracing. It is a variation of the normal way in which pixels are set, and it's purpose is to smooth out the ray traced scenes. For this to be done, the corresponding flag "Multiple rays per pixel" has to be checked. Then, the user can set the number of rays per pixel, up to 10 (actually it will be the square of the chosen number). After the new rays have been generated, using the hybrid strategy described in the book, they will all be averaged for a pixel. This process then happens for each pixel we have.

Rendered Image





Multiple rays per pixel enabled vs not enabled. By looking closely, we can see that in the left picture, the sides are not as jagged as they are in the picture on the right.



Multiple rays per pixel and depth of field working together

Visual Debug

The idea for the visual debug is to draw the each sampled ray, for a certain pixel. This would happen after checking the multipleRaysPerPixel flag and drawing a ray, although I was not able to get it to work. I was able to draw some rays for a certain pixel on the screen (of my choosing), although I could not see them, only the pixels they would intersect on the screen.

Glossy reflections

Sources:

i. Fundamental of Computer Graphics, 4th Edition, p. 333
ii. https://en.wikipedia.org/wiki/Gloss_(optics)

Description:

Visual Debug Button: Debug Glossy

The visual debug for glossy reflection only works when glossy reflection and recursive ray tracing are enabled

main.cpp file

In main.cpp file, we make a slider named ‘Number of Ray’, that takes a user input in the form of integer. This slider takes an integer between 0 to 100, and stores it in a int variable ‘numray’.

CmakeLists.txt

We added the directory of glossy files under ‘add_library’ to execute them

render.cpp file

In render.cpp file, we first receive the user input in slider – ‘numray’ – in the form of extern int. And then, inside the ‘getFinalColor’ method, we create an if statement where if the glossy feature is enabled, we add the return value of method ‘getGlossy’, which is defined in glossy.cpp. If the glossy feature is disabled, we compute the normal recursive ray tracing by making a recursive call of ‘getFinalColor’ function.

glossy.h file

We define the function ‘getGlossy’ that takes a Screen ‘screen’, BvhInterface ‘bvh’, Features ‘features’, Hitinfo ‘hitInfo’, int ‘rayDepth’, and int ‘numray’.

glossy.cpp file

Initially, we define two zero vectors ‘sum’ and ‘S’, the material value for specularity ‘Ks’, float ‘glossiness’, Ray ‘par’, copy of the original ray (with an epsilon value thrown in the equation, to resolve floating errors), and ray ‘reflected’, calculated by the method ‘computeReflectedRay’ in shading.cpp file. We also set the ‘glossiness’ to be the inverse of shininess. If a material has a higher shininess value, it means it is more close to an ideal mirror, and if it has a smaller shininess value, the less it will be closer to an ideal mirror.

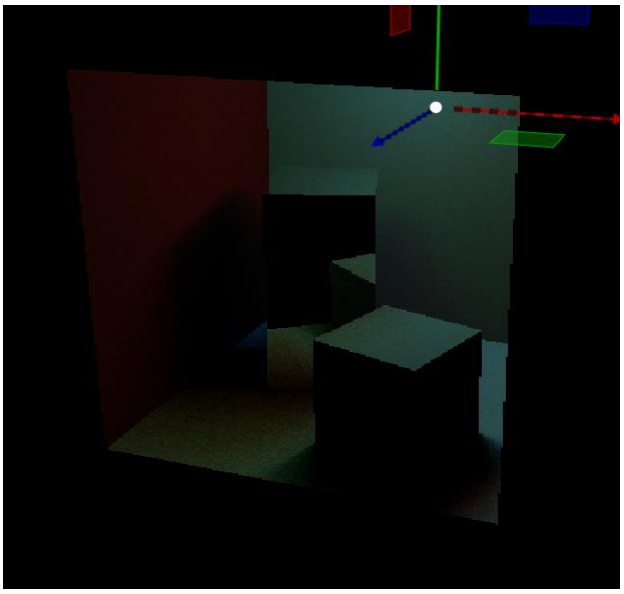
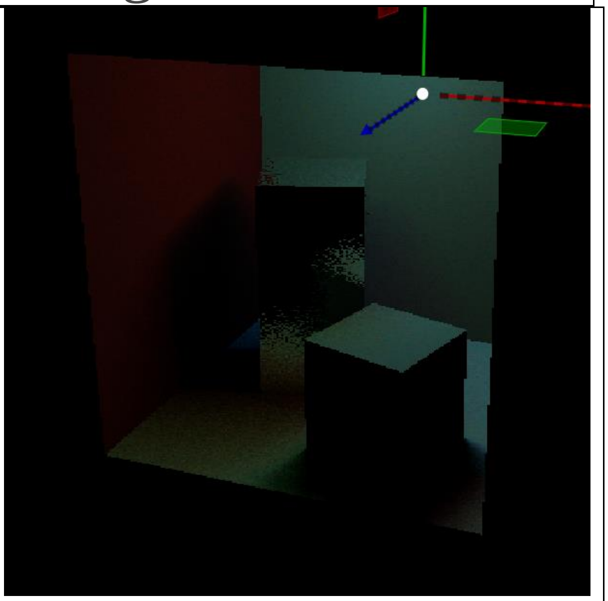
Now we create a for loop that iterates the number of rays (input ‘numray’), since we want to add up the values and then get the average. Inside the loop, we initialize two random floats between 0 and 1, and take their absolute values, named ‘xrand’ and ‘yrand’. We use the ‘glosiness’ value and these random variables to define a sample ray – ‘tempray’.

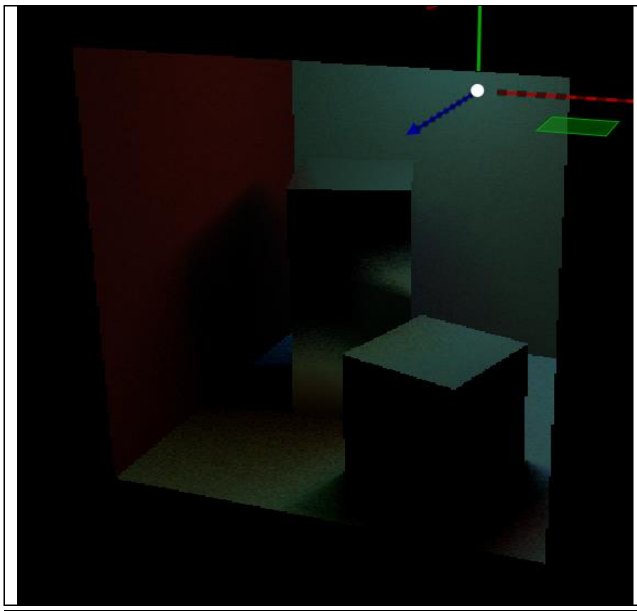
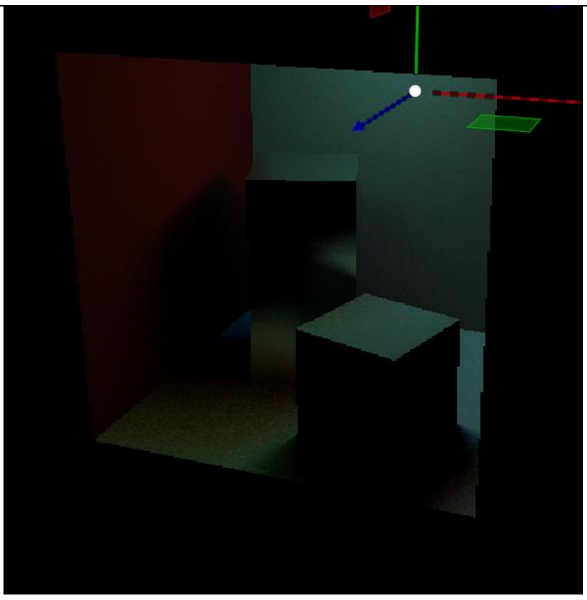
Next we define the three vectors

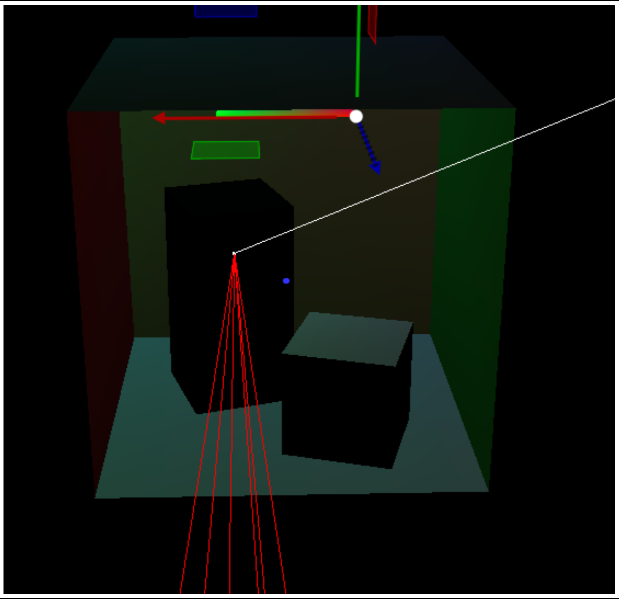
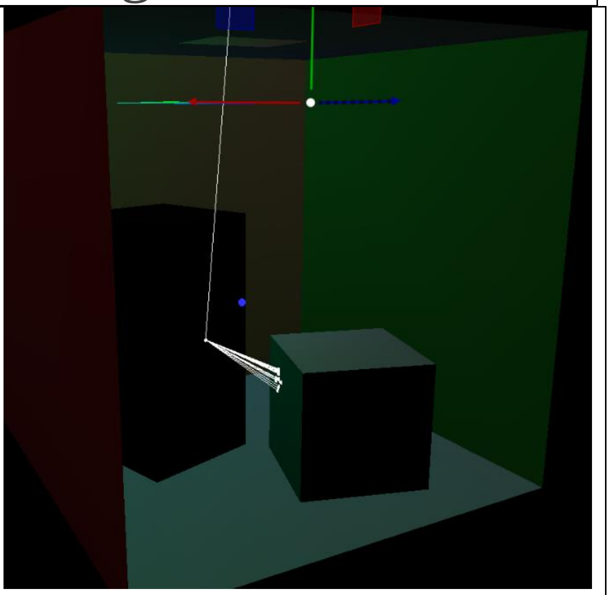
- x : normalized cross vector between ‘reflected’ ray direction and ray direction
- y : ‘reflected’ ray direction
- z : normalized cross vector between ‘y’ and ‘reflected’ ray direction

We set the direction of ray ‘par’ to be the product of ‘tempray’ and x, y, z values given above, respective to its position. We get the color of that ray, multiply with the specularity, and add it to our vector ‘sum’. Lastly, we divide the vector sum with the number of rays we’ve sampled, to retrieve the average value of every samples we’ve calculated.

In the visual debugs, when only the recursive ray tracing is enabled, we can see that the ray reflects in a single form, which is what we could expect from a ideal mirror. If recursive ray tracing and glossy reflections are enabled, there will be no visual debug rays shown. If recursive ray tracing, glossy reflection, and the visual debug button for glossy reflection is enabled, and the ray is shot on the mirror, the ray will reflect and diverse in the many separated rays, which is what we expect from the glossy reflection

Rendered Image	
	
Original Image	Number of rays = 1

	
Number of rays = 30	Number of rays = 100

Visual Debug	
	
Light ray shot on the mirror without intersection	Light ray shot on the mirror with intersection

Transparency

Sources:

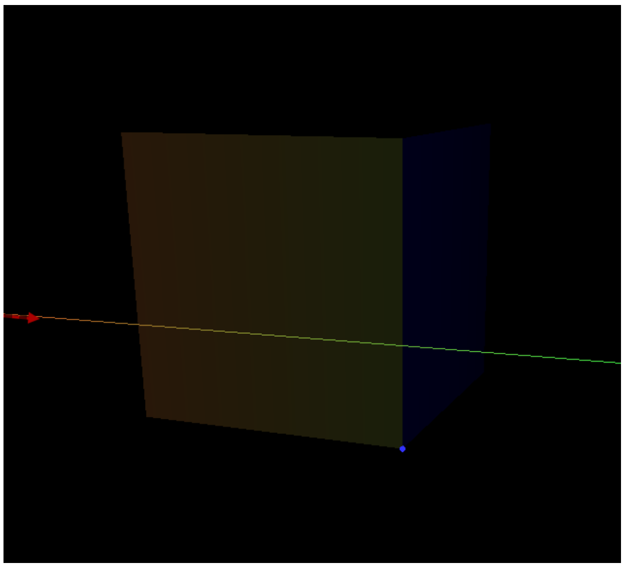
i. https://en.wikipedia.org/wiki/Alpha_compositing

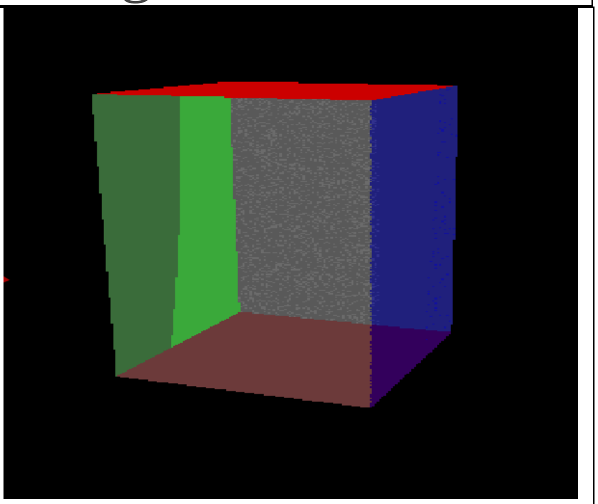
Description

render.cpp file

I implemented this function using a very straightforward formula from my source. Inside getFinalColor, before I return the final color, I check if transparency is enabled. If yes, I compute a ray that starts at the intersection point of the original ray, and goes in the same direction. Using the formula from my source, I just return a new vector, which is transparency * color + (1 – transparency) * getFinalColor called with my new ray. As a recursion base case, I just check if rayDepth < 5 and while calling getFinalColor, I increment it by 1. In this way, I perform the alpha blending process, where I combine the color of the background and the color of the object with specific weights. Obviously the weight in this case is my transparency variable, which defines how transparent the object will be. As a visual debug, I just draw the new computed ray.

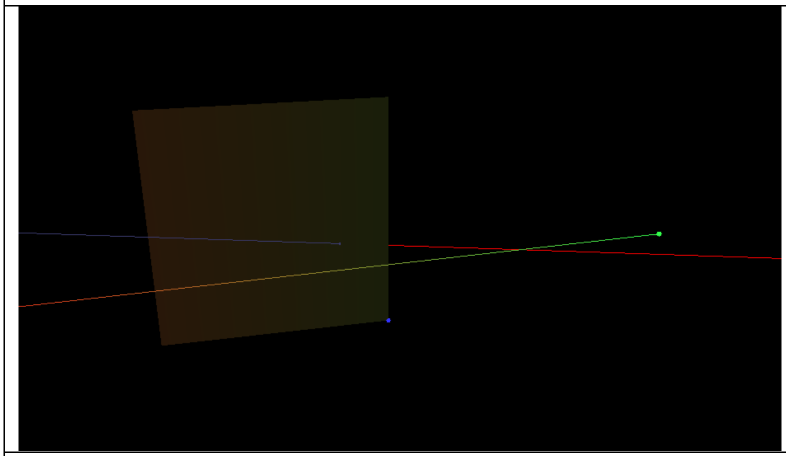
Rendered Image





Original cube

Transparency enabled



Visual debug

Depth of field

Sources :

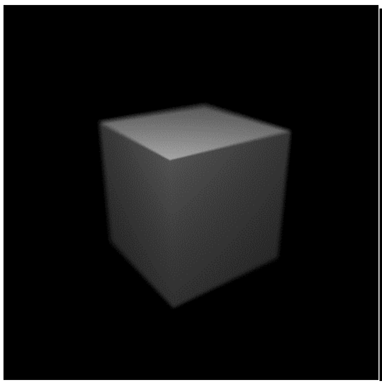
i) https://pathtracing.home.blog/depth-of-field/

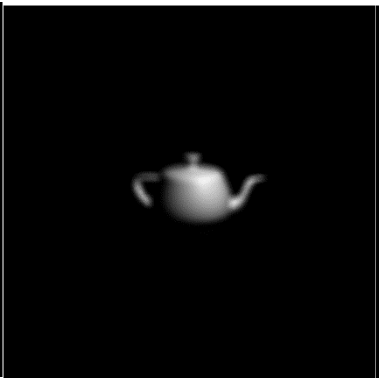
The depth of field task is implemented in render.cpp. The method in which this task is done is depthOfField, and it is called in renderRayTracing. It takes as parameters the aperture, focal length and number of samples (along with the scene, bvh, features and ray), which can be changed by moving the slider when running the application. For each pixel, a camera ray is created, which is then used in the depthOfField function. Then, a number (which is equal to the number of samples) of secondary rays are computed. A secondary ray is computed in the following way :

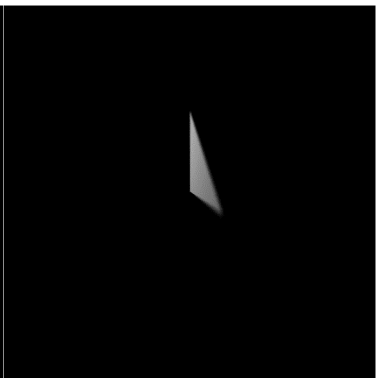
1) A convergence point is calculated using the ray origin, the ray direction and the focal length;
2) The ray origin is shifted using the aperture variable and a random number generator (a random number for each axis), in order to get the blur effect;
3) The new ray direction is calculated for the shifted point;
4) Repeat until number of samples is satisfied and then average all of the rays for a pixel.

For the debug, I replicated this program in the getFinalColor method. After a ray is shot through a the pixel of a triangle and the depth of field flag is checked, you can see how the random secondary rays are generated. You can move the sliders for the aperture, focal length and number of samples to see how they affect the generated rays.

Rendered Image







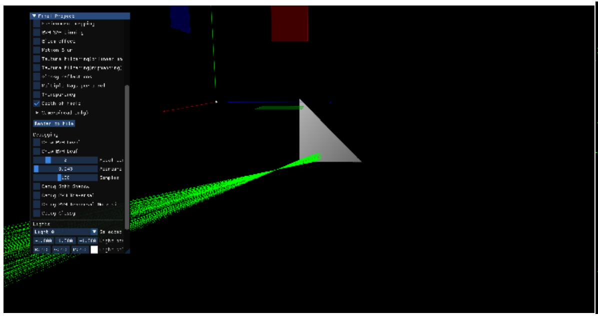
Focal – 2, Aperture – 0.044, Samples - 94

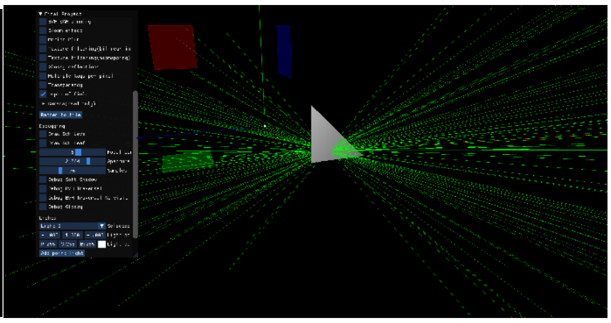
Focal – 3, Aperture – 0.084, Samples – 100

Focal – 5, Aperture – 0.445, Samples – 78

For example, in picture 3, the lower right corner is blurry because the focal point for those pixels is far, whereas the lower left corner is clear because the focal point is right next to it.

Visual Debug





Focal – 2, Aperture – 0.043, Samples - 100

The area defined by the rays would be seen as blurry, because of how far the focal point is

Focal – 6, Aperture – 2.264, Samples - 76

As the aperture is increased, the rays become more dispersed, and even though the focal point is quite close to the triangle, the rendered image will be very blurry

Performance test

	Cornell box	Monkey	Dragon
Number of triangles	32	968	87130
Time to create	7.1ms	10ms	22.2sec
Time to render	300ms	700ms	16.8sec
BVH levels	3	8	14
Max triangles per leaf node	2	2	3