Group 170
Bosma, Pepijn
Iftode, George
van Pelt, Mitchell

Table of contents

Table of contents **2**

Basic features **3**
- Shading using Phong Illumination Model 3
- Recursive ray-tracer 5
- Hard shadows 7
- Area lights 9
- Acceleration data-structure generation 13
- Acceleration data-structure traversal 17
- Normal interpolation with barycentric coordinates 20
- Texture mapping 22

Extra features **23**
- Texture filtering: Bilinear interpolation 23
- Cast multiple rays per pixel with Irregular Sampling or Antialiasing 25
- Glossy reflections 27
- Depth of field 31

Performance test **35**

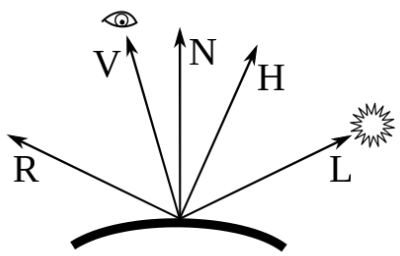
References **36**
- Shading using Phong Illumination Model 36
- Area Lights 36
- Normal interpolation with barycentric coordinates 36
- Texture filtering: Bilinear interpolation 36
- Glossy reflections 36
- Depth of field 36
- Irregular sampling 36

Basic features

Shading using Phong Illumination Model

computeShading(...):
The point where the ray hits is calculated using the ray provided. The normal vector (N) is calculated with the method *intersect(...)* in “bounding_volume_hierarchy.cpp” using the cross product between edges:

$$(v1 - v0) \times (v2 - v0)$$



The vectors V and L are both calculated using this formula:

$$\vec{AB} = \vec{b} - \vec{a}.$$

R is more complex. First the projection of L on N is calculated:

$$projection\ of\ \vec{L}\ on\ \vec{N} = \frac{\vec{L} \bullet \vec{N}}{\left|\vec{N}\right|^2} \cdot \vec{N}$$

Since all direction vectors are normalized, the bottom term cancels out nicely. Now we need a vector from L to the projection, this will be equal to:

$$\vec{L} \bullet \vec{N} \cdot \vec{N} - \vec{L}$$

To get R, we travel to L once, and twice to this this projection vector:

$$\vec{L} + \vec{L} \bullet \vec{N} \cdot \vec{N} - \vec{L} + \vec{L} \bullet \vec{N} \cdot \vec{N} - \vec{L}.$$

Some terms can be grouped together and some cancel out giving us:

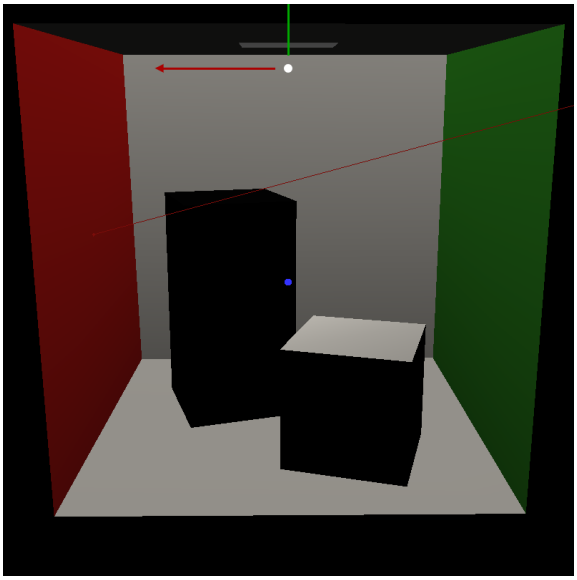
$$2 \cdot (\vec{L} \bullet \vec{N} \cdot \vec{N} - \vec{L}) + \vec{L}$$

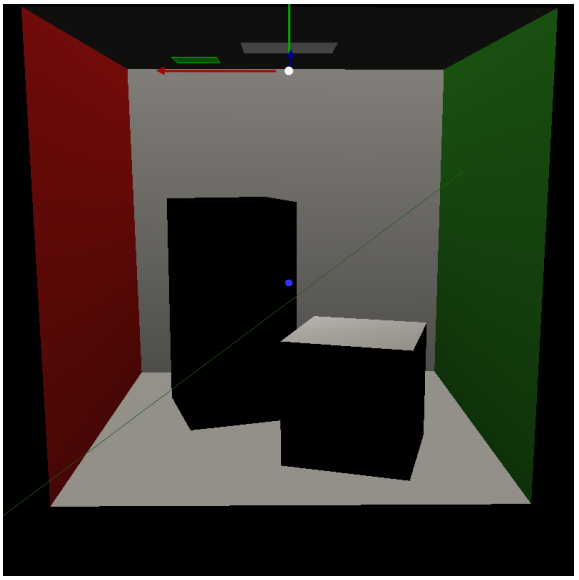
If the light is behind the surface, it should not light up the other side. Checking if the dot product between N and the ray.direction is greater than or equal to zero filters this out.

Now the formulas from the lecture can be filled in. The diffuse (D) term will only be added if $\vec{N} \bullet \vec{L} \geq 0$. The specular (S) term will only be added if $\vec{R} \bullet \vec{V} \geq 0$. This filters out lights that come from the back of a plane.

computeLightContribution(...):
First check if shading is enabled. When it’s not, return the diffuse color. If it is, all lights are checked using the for loop provided in the comments. When a point light is found, other cases will be discussed in the corresponding features, the color is calculated with *computeShading(...)* and multiplied by *testVisibilityLightSample(...)* to check for visibility. The last step is to add it to totalColor.

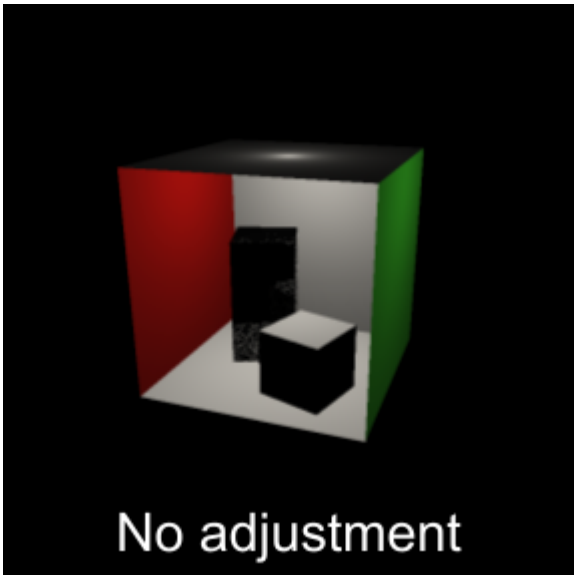
Visual debug:

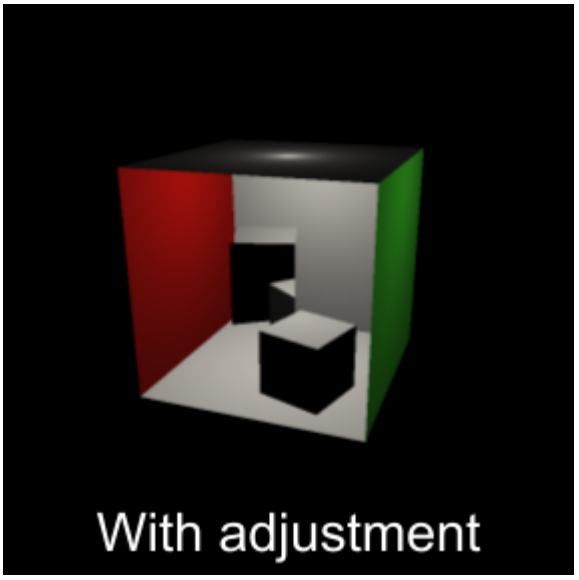




Recursive ray-tracer

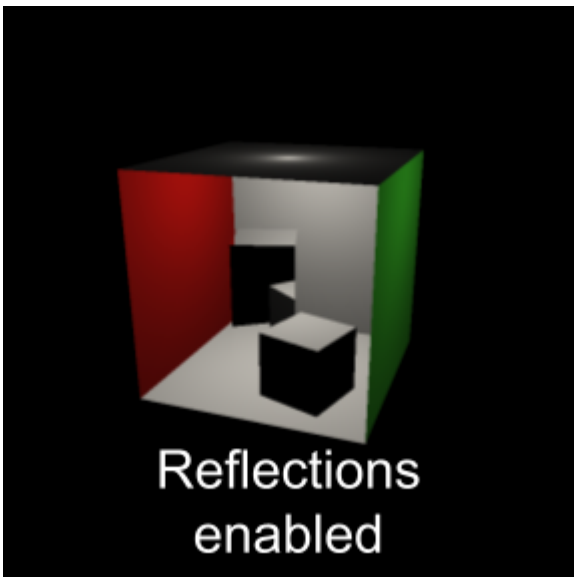
computeReflectionRay(...):
If the specular constant of the material is black, the function returns a vector centered at the origin with t = -1. With this we can filter out these reflections in *getFinalColor(...)*.
The direction of the reflected ray is the opposite of the vector R calculated in *computeShading(...)*. Therefore the calculation is the same, with only the L vector being the ray.direction inverted.
The origin of the reflected ray is the endpoint of the ray provided. To stop the reflected ray from intersecting immediately with its own reflective surface, it is translated a minimal amount in its own direction.



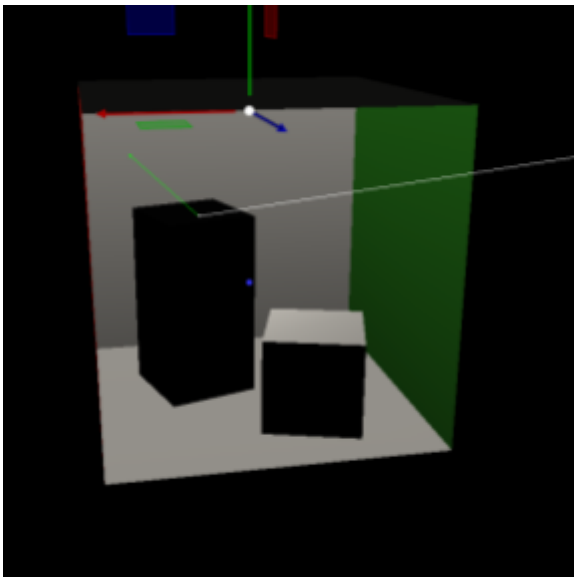


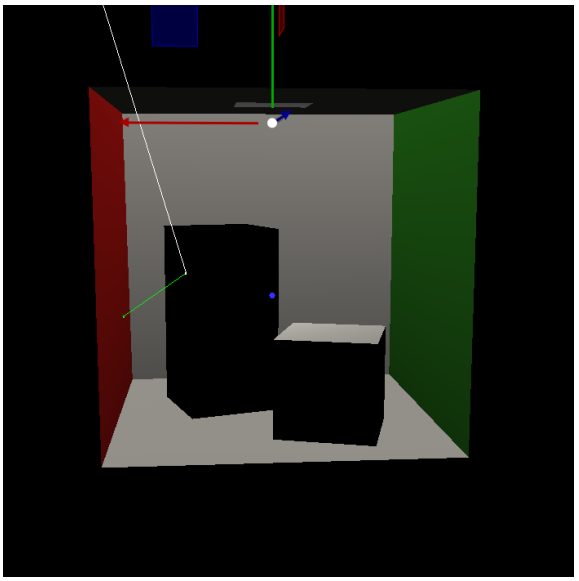
Final step is to set the t-value to the float max.

getFinalColor(...):
If recursive reflections are enabled, calculate the reflected ray. First calculate if the reflected ray hits another surface. This is done before the visual debug to get the t-value of the reflected ray right. Then check if the t-value is valid (bigger than or equal to zero). If so, it adds the light contribution of the reflected ray to the Lo of the original ray.



Visual debug:





Hard shadows

testVisibilityLightSample(...):
A point can be occluded for two reasons:

1) The ray from the point to the light (lightray) intersects with the plane the point lies on from the opposite side than the camera is looking at. The side on which a ray hits can be calculated using the dot product between the ray and a normal on the plane. If the ray hits on the side the normal is pointing away from, the dot product will be positive, otherwise negative.

$$s1 = n \cdot PC$$
$$s2 = n \cdot PL$$

With n being a normal on the plane the point lies on, PC the unit vector of the direction from the point to the camera, PL the unit vector of the direction from the point to the lightsource, and $s1$ and $s2$ representing the side from which the rays hit the plane. If $s1$ and $s2$ are both positive or both negative the rays hit on the same side, if not, the rays hit on different sides and thus the point is in shadow.

2) Another object is blocking the lightray and the lightray thus can't reach the light. To determine if the lightray is blocked by something, it can be tested if it intersects with anything before it reaches the lightsource. To achieve this, the lightray l is defined as

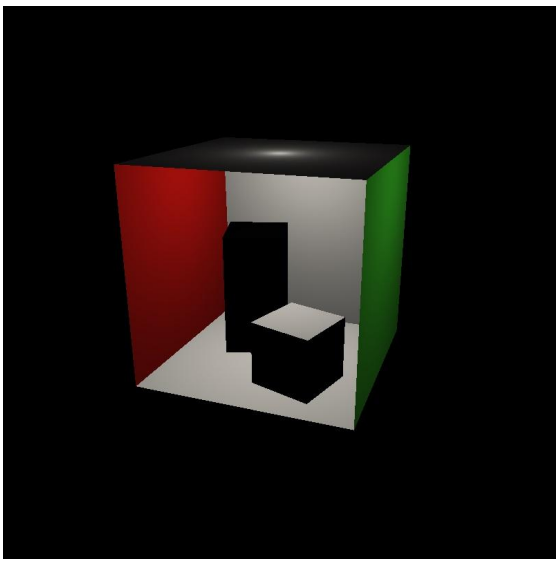
$$l = p + tPL$$

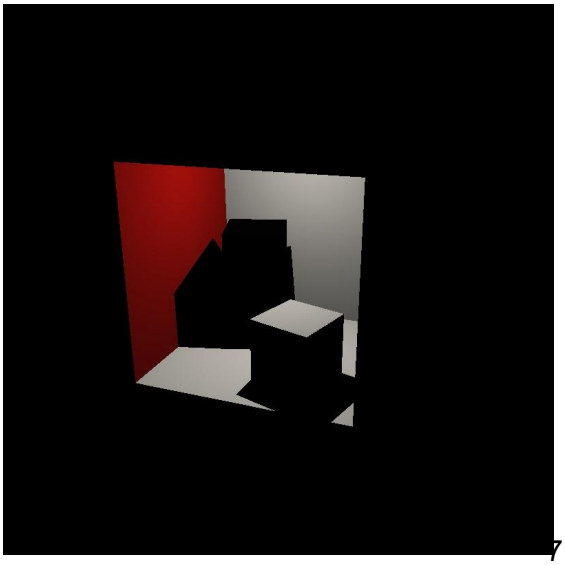
With p being the point that is tested for occlusion and t the t-value representing the length of the ray (initially set to infinity). t can be calculated using the *bvh.intersect(...)* function. After the intersect function is used on the ray, the t-value will be updated to represent the distance between the point and the first object it intersects with. If the lightray intersects with anything before it reaches the lightsource, the light is blocked by this object and the point will be occluded (O). I.E. the t-value is smaller than the distance between the point and lightsource.

$$t < d(P, L) \;\; -> \;\; O$$

Note: in the implementation p is moved closer to the lightsource by a slight amount to avoid the ray intersecting with p itself due to floating point inaccuracies.

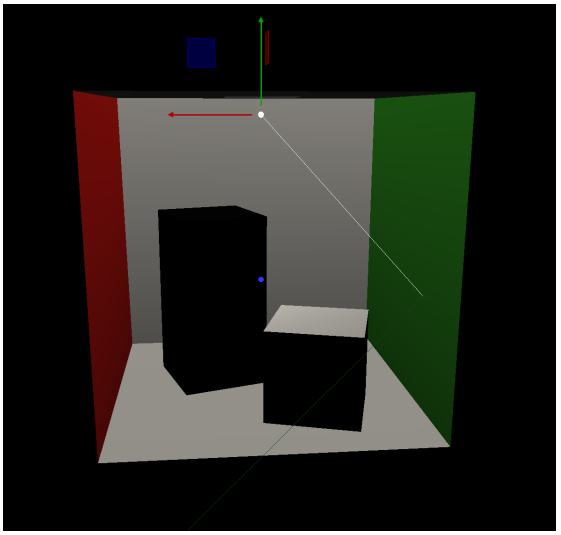
Rendered images:

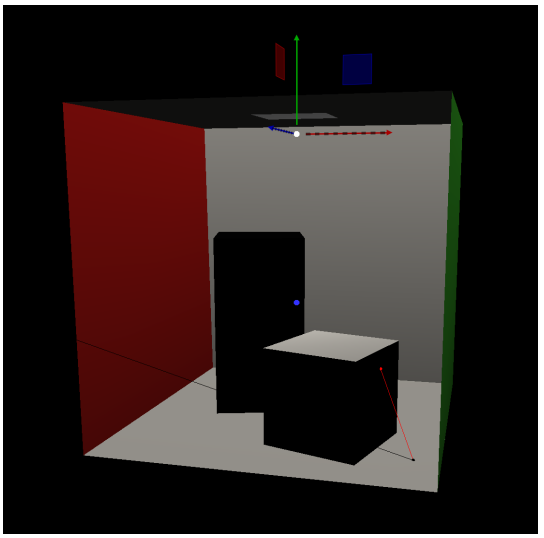




Left: Hard Shadows disabled (shading enabled). Right: Hard Shadows (and shading) enabled.

Visual debug:





Left: lightray that reaches the light drawn in the color of the light. Right: blocked and thus occluded lightray drawn in red.

Area lights

To sample points from the segment and parallelogram lights the Monte Carlo method was used. With this method, a random point is chosen from an uniform distribution and used for calculations. When the sampling is repeated many times, the resulting averages will converge to the values of the entire lightsource. The advantage of this method is that the taking of an individual sample is computationally inexpensive and not many samples have to be taken for an accurate color. Furthermore, each sample is independent from the previous samples, thus preventing any patterns from forming.

sampleSegmentLight(...):
A point on the segment light can be calculated as follows:

$$p = v0 + r(v1 - v0)$$

With $v0, 1$ being the start and endpoint of the segment and r a scalar between 0 and 1. In the Monte Carlo method r must be a random number that is different in each iteration. To pick such a number the *rand(...)* function in the C Standard General Utilities Library was used.
The color c of the sampled point can be calculated using linear interpolation with the same random scalar r and the provided endpoint colors $c0, 1$:

$$c = (1 - r)c0 + rc1$$

sampleParallelogramLight(...):
The sampling of the parallelogram light uses the same method as the segment, only more extensive since a parallelogram is a two dimensional object rather than just the 1D for the segment light. A point on the segment can thus be calculated with:

$$p = v0 + r1(e1) + r2(e2)$$

Here the adjacent edges $e1, 2$ of $v0$ are already provided in the parallelogram structure. Two random scalars $r1, 2$ are used to obtain random coordinates of both dimensions.
To calculated the corresponding color of the sampled point bilinear interpolation is used: first the two colors on the 1D opposite edges are calculated using linear interpolation:

$$cA = (1 - r1)c0 + (r1)c1$$
$$cB = (1 - r1)c2 + (r1)c3$$

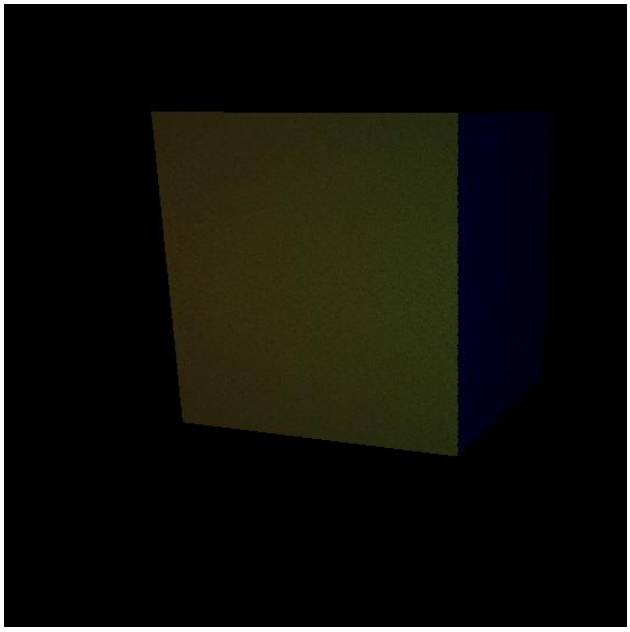
To complete the bilinear interpolation process, the resulting colors cA, B are again interpolated with each other in the other dimension:

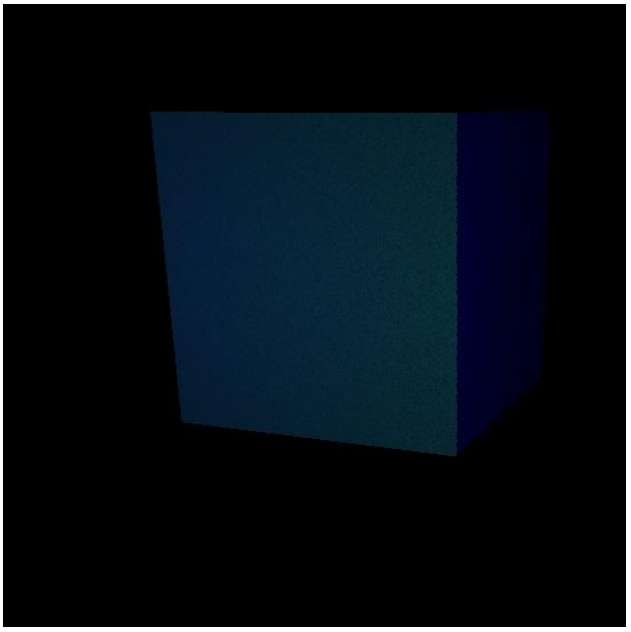
$$c = (1 - r2)cA + (r2)cB$$

computeLightContribution(...):
This method loops over all the lights in the scene. To ensure the *rand(...)* function indeed provides random numbers, it must be initialized with a seed using the complementary *srand(...)* function. This is done before the loop over the lights starts. It is common to use the current time in seconds as a seed, but after it was observed that multiple consecutive frame renderings used the same seed and thus produced the same set of random numbers, it was changed to the current time in milliseconds.
For the Monte Carlo method to work the sampling must be repeated many times. After a color sample is taken the *testVisibilityLightSample(...)* is used to test if it contributes to the lighting. Per light the average of all the samples that are visible is taken to compute the final color of each segment and parallelogram light. In the implementation 20 samples are taken

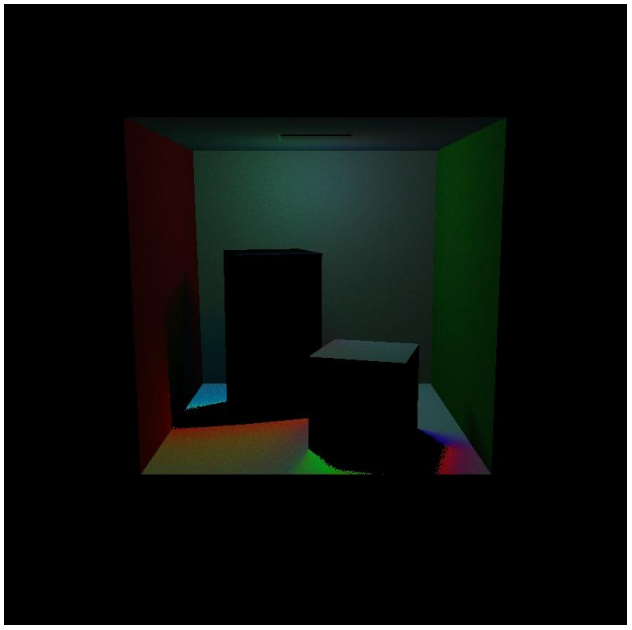
for each segment light and 100 for a parallelogram. These numbers are relatively low and were chosen for practical computation speed, thus allowing the soft shadows to work on live ray tracing mode. Note that these numbers can be increased to provide a more accurate and less noisy rendering of the soft shadows.

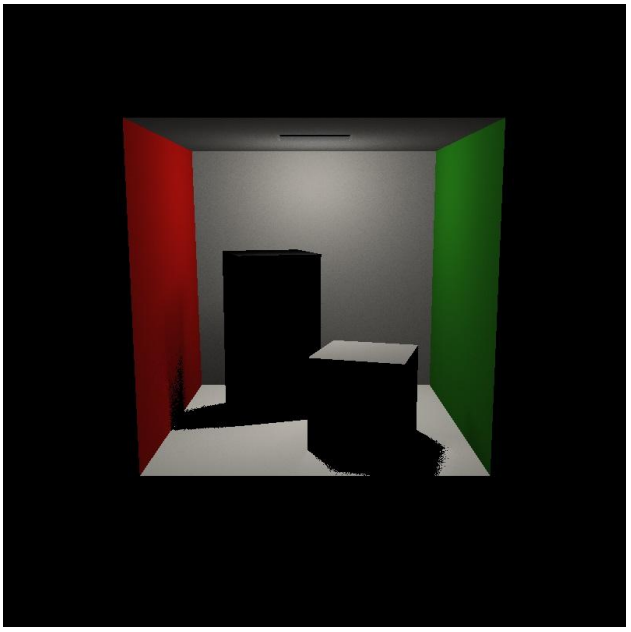
Rendered images:





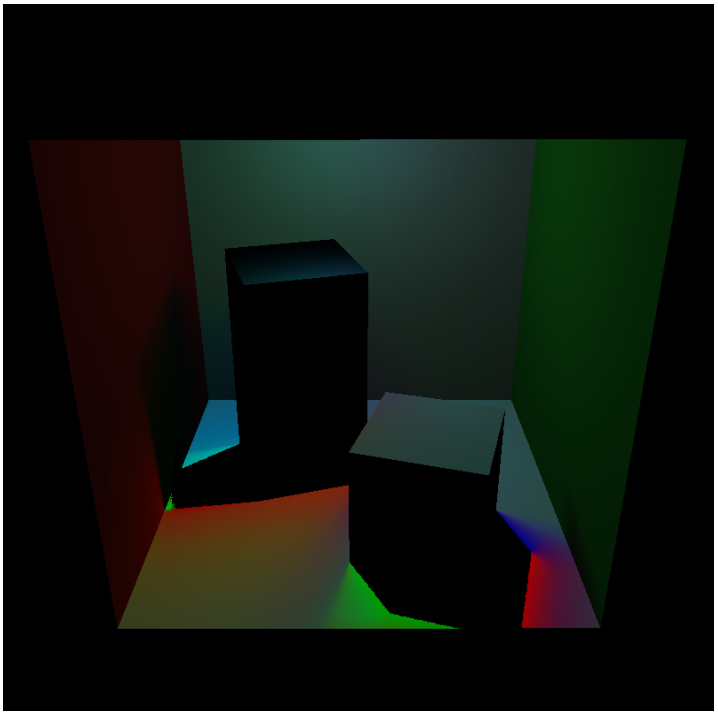
Left: segment light with red and green. Right: segment light with blue and green. (A slight color fading is visible).



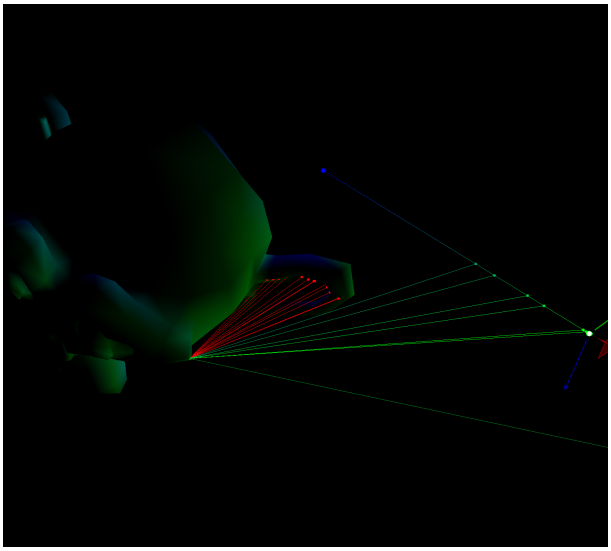


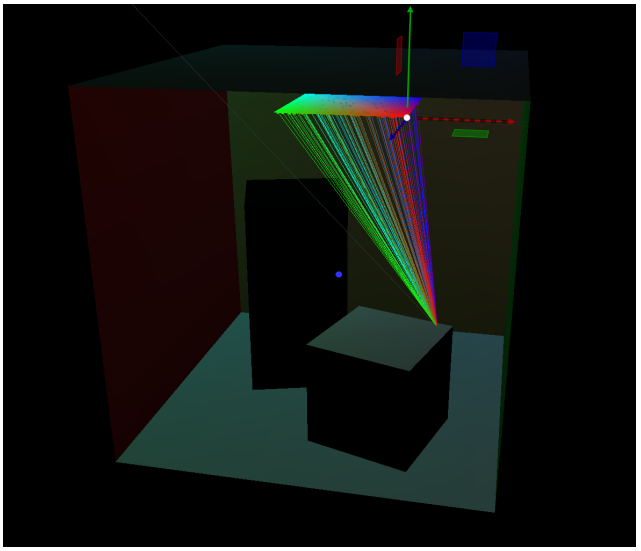
Left: parallelogram light with red, blue, green and cyan. Right: parallelogram light with only white to illustrate the soft shadows. Note again that these shadows are a bit ‘dotty’ here. To illustrate the full potential of the ray tracer, the bottom image was rendered with a

parallelogram light sample rate of 10000. Note that now there is almost no noticeable noise on the surfaces and a very realistic fade between the colors on the floor.

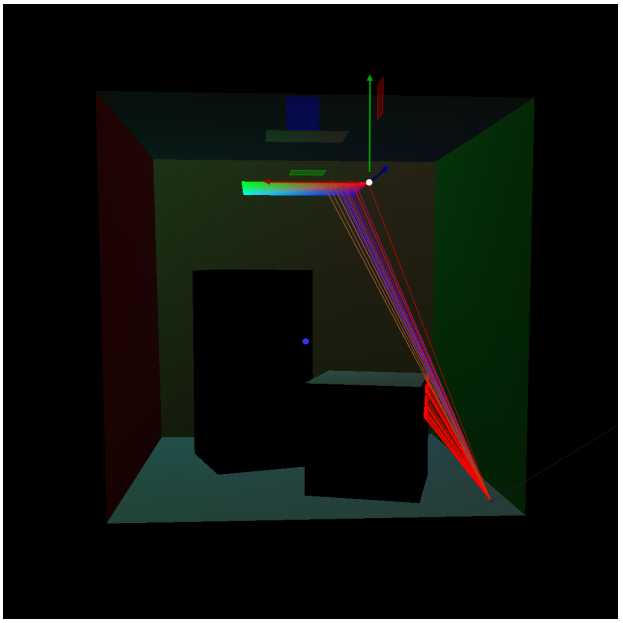


Visual debug:





Left: segment light with half of the sample rays being blocked. Right: parallelogram light with all the sample rays reaching the light. Bottom: parallelogram with large part of sample rays being blocked by the cube.



Acceleration data-structure generation

Before we talk about the generation we must first understand the *structure* of it.

In a typical BST fashion the BVH structure we use is a *std::vector* which has the property that if we know the index of a node we can deduce the indices of it's 2 children. More specifically if we know *root* then

2 * root + 1 is the *left* child ***and*** 2 * root + 2 is the *right* child

Each element of the BVH vector is of structure *Node:*

```
struct Node {
    AxisAlignedBox box;
    bool leaf;
    std::vector<std::pair<int, int>> indx;
};
```

- box - is the *Axis Aligned Bounding Box* for the current Node
- leaf - is a boolean which specifies if the current node is a leaf or not
- indx - is a vector containing pairs of 2 forms:
    - If the node is a **leaf** then it contains all the triangles is has inside as a pair of 2 indices (*index of the mesh*, *index of the triangle inside the mesh*)
    - If the node is **not a leaf node** it just contains the indices for it's 2 children on the first position in the vector

Now onto the actual generation: in order to begin the creation of the structure we need to do 2 things: *first* get the list of all the triangles inside the scene because we want to be able to split them in AABB's, and *second* properly size the BVH vector to make sure everything fits inside. This is done inside the constructor.

For getting the triangles there is a method in *bounding_volumes_hierarchy.cpp* - *getTriangles()* which takes in the vector in which we want to add the triangles to and the vector of Meshes. We store them as earlier with a pair of 2 indices.

For sizing the BVH vector of Nodes we know that a **full BST** has the property that the number of levels is equal to the *exponent* of the smallest power of 2 larger than the number of leaves which in this case coincides with the number of triangles. Besides this since the BVH tree could have a very large size we capped it at 20 levels of depth maximum since it still improves the performance by a lot and. So since we can get the depth level easily for calculating the size we allocate 2 ^ (depth+1) - 1 as size since if you sum the number of nodes on each level (2^0 + 2^1 + … + 2^depth) you get this number. Since not every node can have a triangle inside or children from the splitting some of them can be *empty* so we allocate the full size in order to keep our properties and make sure *nothing* goes wrong with memory *:(* .

Now that we have everything ready we call the *build()* function which recursively builds our BST having as parameters the following:

- the vector as reference (because it's easier to work with)
- the vector of triangle pairs that the current node has inside
- the index of the node we're inside (we regard as the *root* for the subtree)
- the depth

Before doing anything else we must calculate the AABB for the node we're in and this is done by the *getMeshesAABB()* function which uses 2 sub functions: *getTriangleBox()* which takes the vertices for each triangle and builds it's specific box, and *combine()* which combines 2 AABB's into 1 that contains them both. This method iterates through every triangle, gets its box and sums it up to the box of the node.

The next step of our building process is ***checking*** whether we hit our *depth threshold* or if we only have a *single triangle* inside our node. If that happens we mark our node as a *leaf*, add the triangle/s to the vector inside the node, and return (we also increase the number of leaves and add it to a vector of leaves for debug purposes).

If the node is **not a leaf** then we add the indices for it's children to the indices vector and begin the **splitting process**. For the splitting process we have a hacky way of doing it. Since the order is based on the *depth* (we first split across x then y then z) we can check what axis we need to split by using a modulus operation on the depth by 3. If the *remainder* is *0* we split by x (since x is the first axis and the root has depth 0), if it's *1* we split by y, and if it's *2* we split by z. This can be combined with an array of 3 glm::vec3's with the following format. The *array[0]* contains a vec3 with the x element 1 and the others 0, the *array[1]* contains a vec3 with the y element 1 and the rest 0, and the *array[2]* contains a vec3 with the z element 1 and the rest 0. Now we can use this information to easily process the triangles centroids. For the centroids we have a vector of pairs of (float, int) which stores:

(the coordinate of the triangle's centroid on the axis we're interested in, the index)

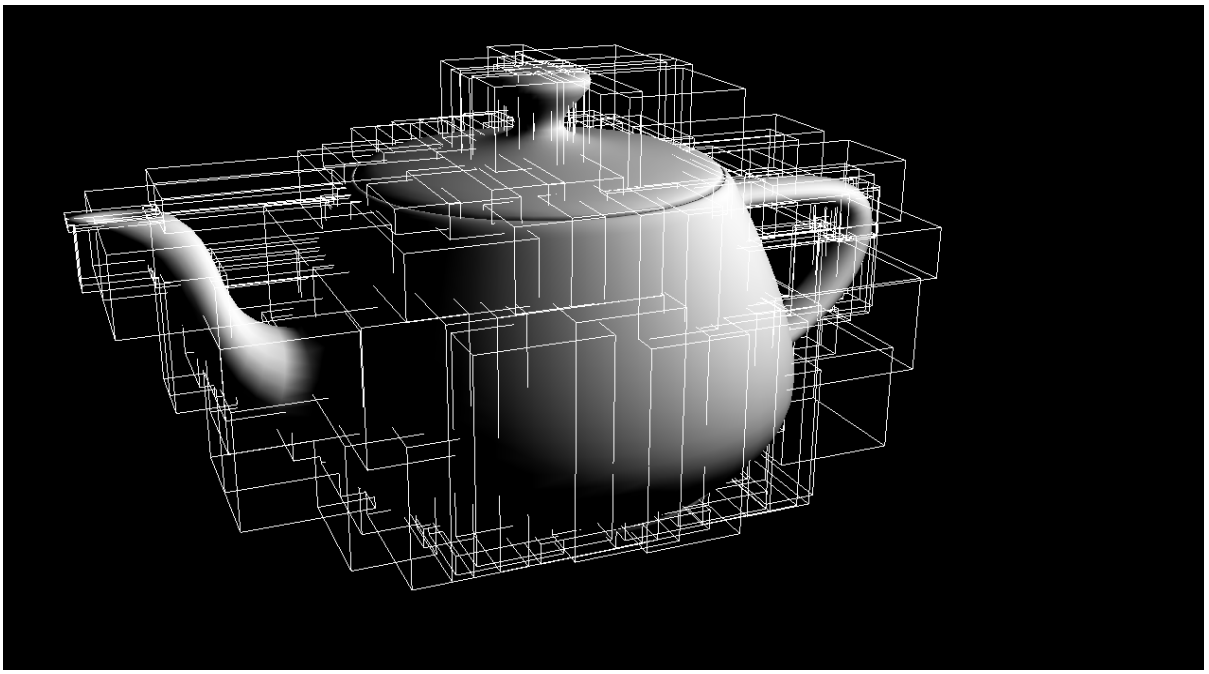
Now to build our **centroids vector** we iterate through the triangles, calculate the centroid, and now for the *smart part*, we get the dot product of the *centroid* and the element in the array of smart glm::vec3's (1,0,0) in the position determined by the depth's module by 3. This gives us a single float which we can use to sort the vector of centroids simply with an *std::sort* and split it in the middle. This is a simpler way of approaching this problem than creating 3 different custom sorting comparators with more and much harder to understand code.

Having sorted this and split the triangles into 2 parts we can go and call our *build()* function again on the 2 children.

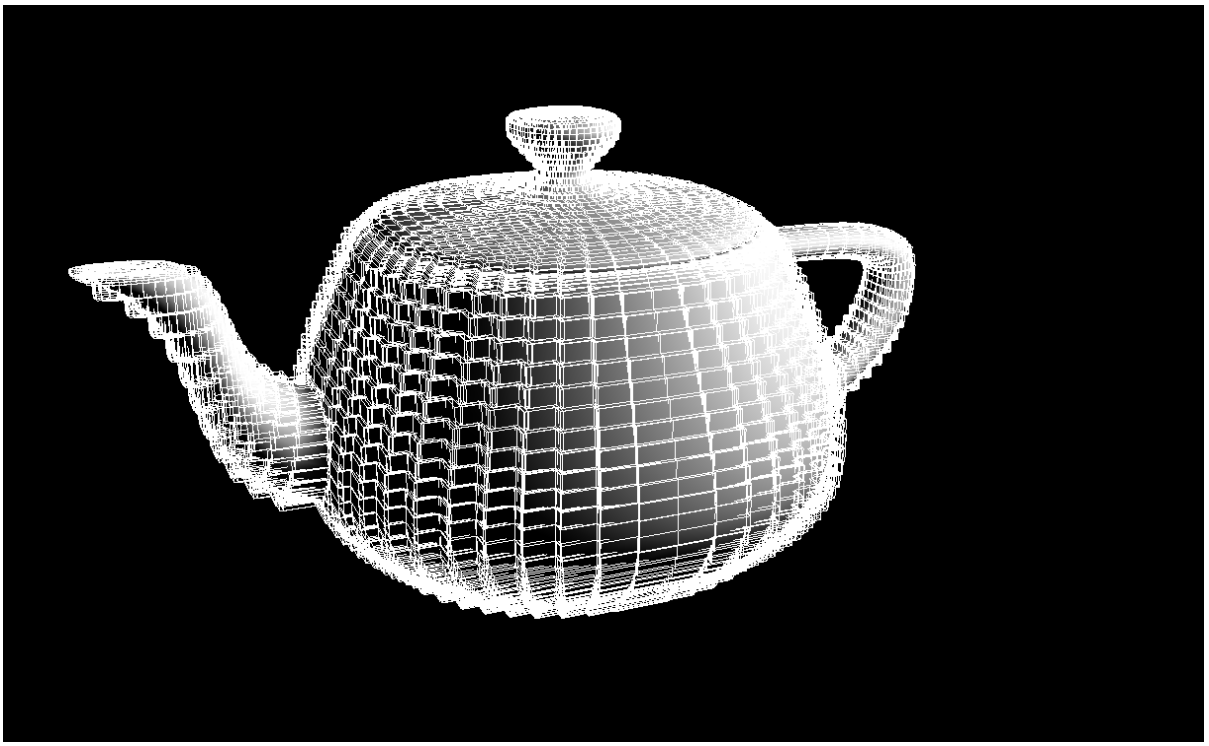
Unfortunately, the support on **spheres** is yet to be *implemented* due to it being an optional part and to time constraints and other course work but it would be the most important thing to do as an *improvement* of the whole quality of the *renderer*.

Visual debug and functionality:

Accessible from the GUI

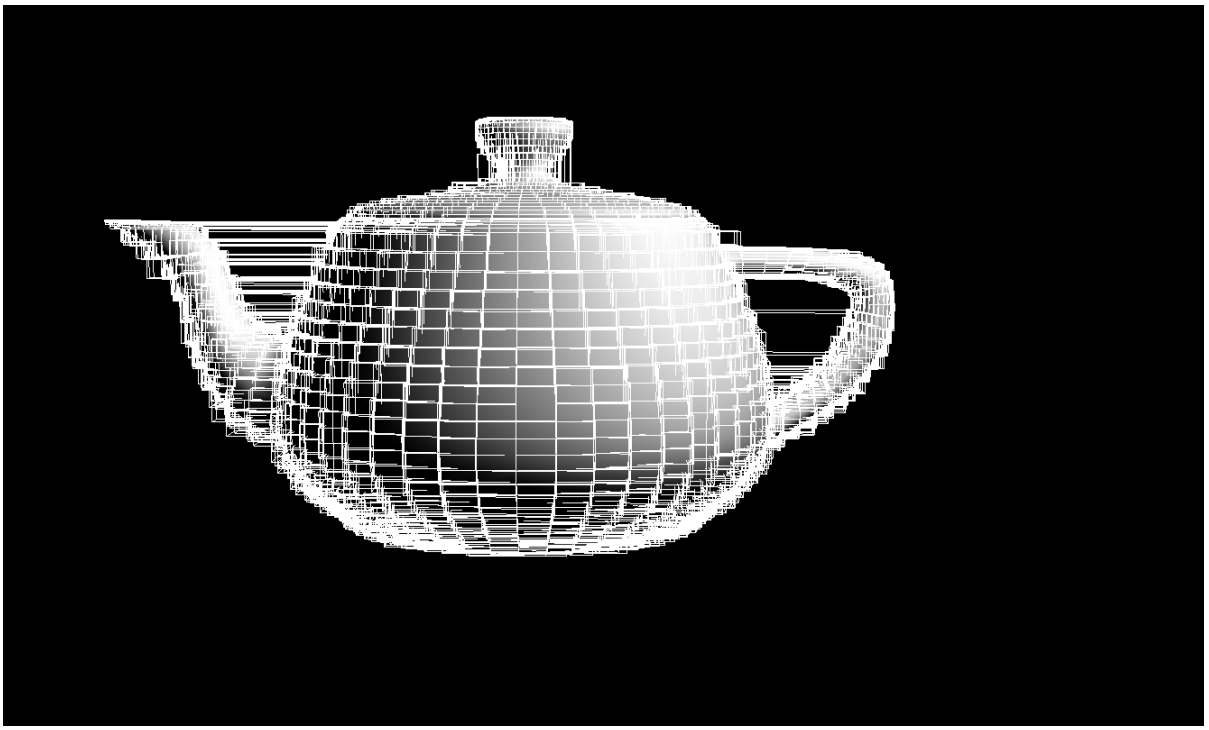


Here is a *random* level that is not the *root* (one box, the biggest), nor the *leaves* level (each box contains its own triangle). Each wireframe box contains the bounds for the triangles inside it.

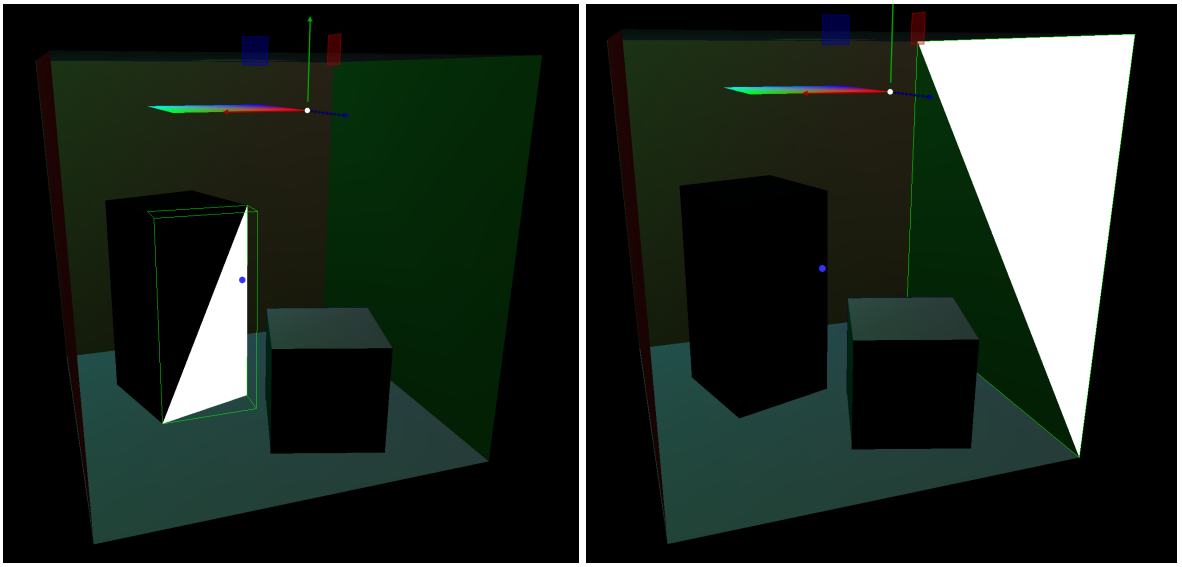


The model I think represents the best the corectitude of the BVH creation is the **teapot**. On the <u>last level</u> (only leaves with 1 triangle) it should show a nice wrap of the model

in small boxes. If the implementation wasn't correct or had some small bugs, the model would have had *some boxes* around the handle and spout which would connect to the main body and ruin the visual like this:



You can also visualize the individual leaves of the BVH to see the box and the triangle it contains. This is best shown in the *Cornell box* model:



There isn't a model provided to showcase what is drawn if there is *more than 1 triangle* in a leaf but the feature is there and implemented (*20 maximum depth is bigger than expected*).

Acceleration data-structure traversal

The traversal is an *easier* part than the creation since we don't need to *optimize it* as much ourselves because *c++* will do it for us. The data structure we're going to use for this is the **std::priority queue**. Why? Because we need to be able to process first the bounding box that is the closest to our ray/ viewpoint because we want to be able to **stop almost immediately** when we hit something that satisfies us. Since the *property* of the priority queue is that it gives us something that is the closest and we want to hit the closest triangle first so we don't *waste time* on other computations, we process them this way because if we hit something that is closer the rest don't matter anymore (they're behind and can't be seen).

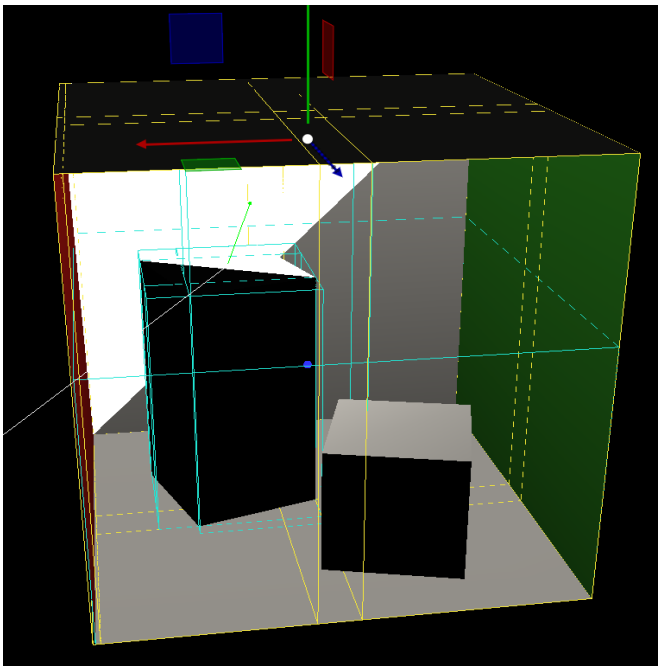
The way we store this information is again with **std::pairs** of type (*float, int*) like so:

(*- the t of the ray that hit this node, index of the node)*

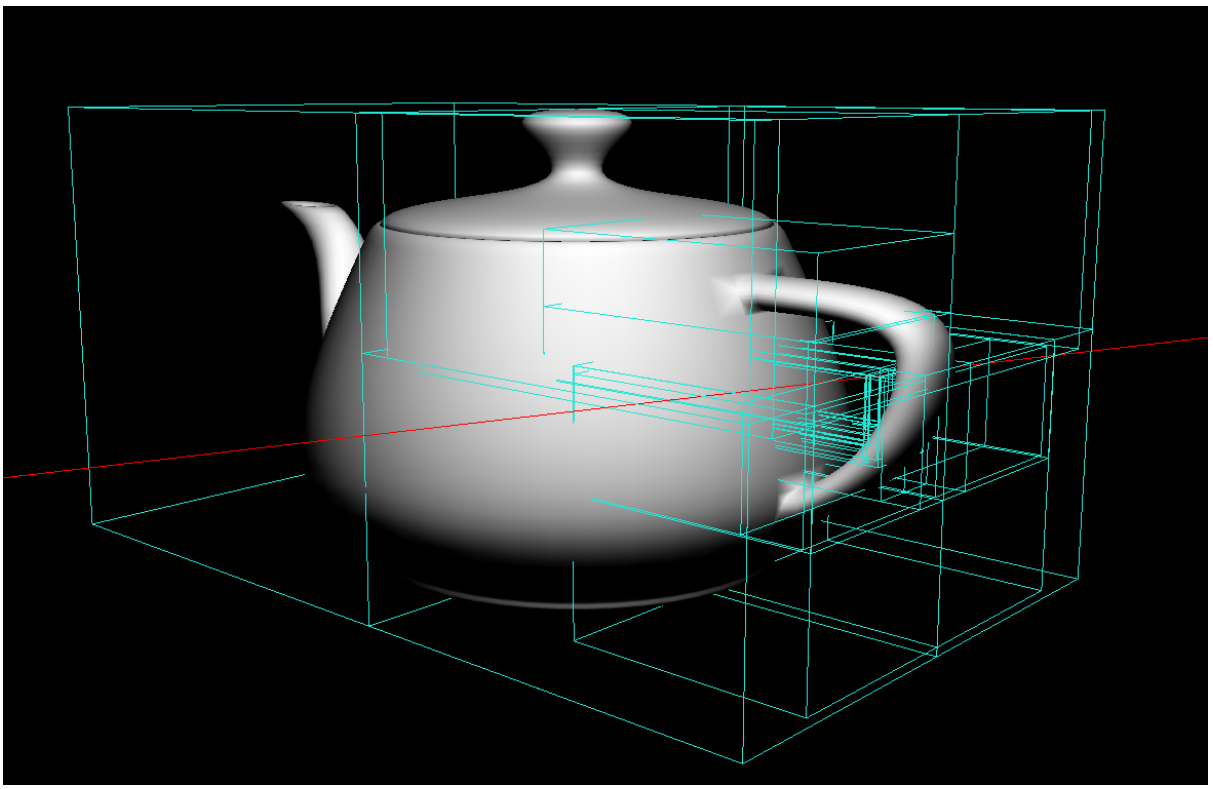
Now since we store the elements of the priority queue as pairs, the c++ compiler knows how to compare them already (by the first value first, and then by the second), we don't need to provide our *own comparator*, and the queue will give us our best element just fine. However for that we need to get the **biggest ray.t** number and by itself the priority queue returns the **smallest element**. That's where the minus from the pair comes from, it's **not a typo**. Doing this we can trick the priority queue into giving us the biggest element by multiplying it with -1 (since it's negative). Now if we get *a leaf node* from the queue we check if we intersect *a triangle* in it, and if it's an *interior node* we check if we intersect the *bounding box* of a child and *put it in the queue* if we do. For *debugging purposes* we also store the nodes we traversed in a stack and if we hit a triangle we store it also for *drawing* at the end.

Visual debug and functionality:

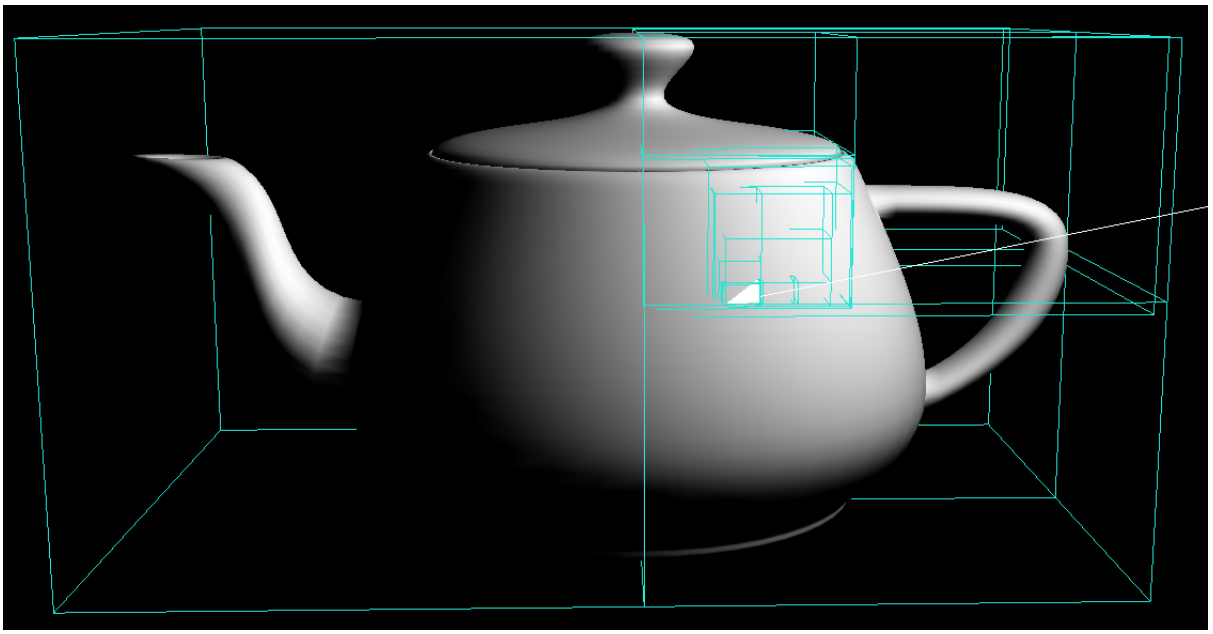
*Inside bvh.cpp there is a flag that needs to be set to 1, called debugBVH because we can't pass parameters from bvh class to main sadly. Make sure you **Don't** have it on during raytracing because it will slow the process down by a lot.*



We don't have a good enough model with reflections but you can see in this image the blue wireframes from the first ray and the orange ones from the first recursion. Also both of the triangles hit are drawn on the model.

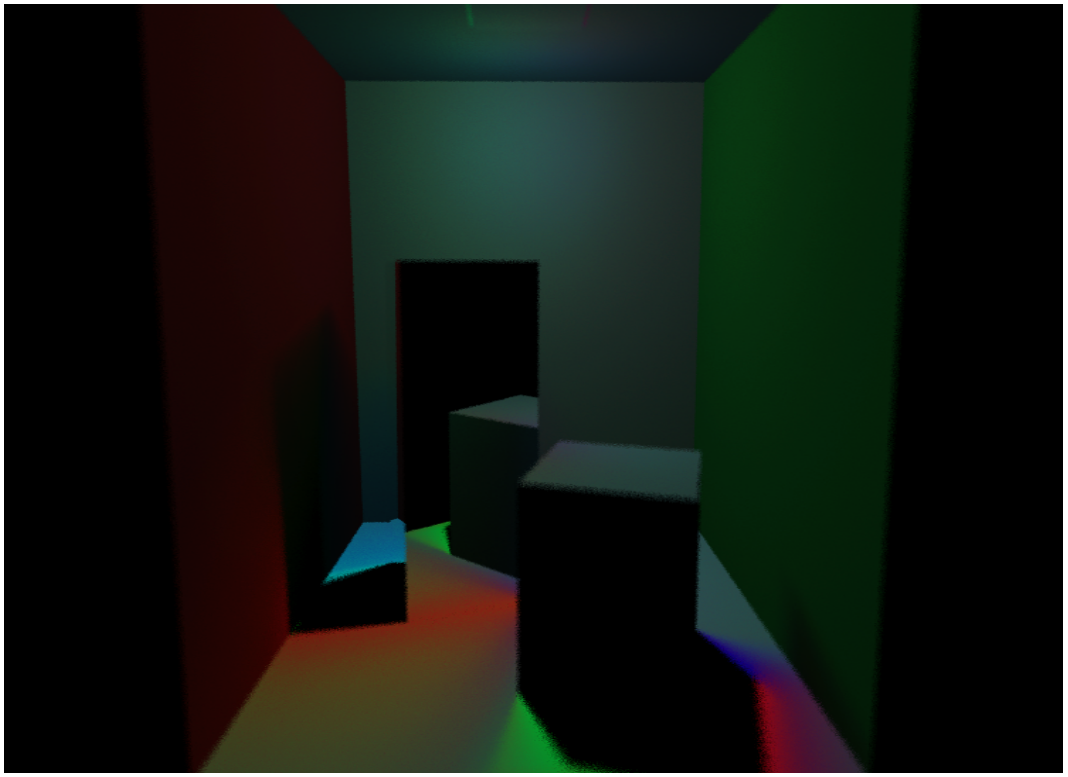


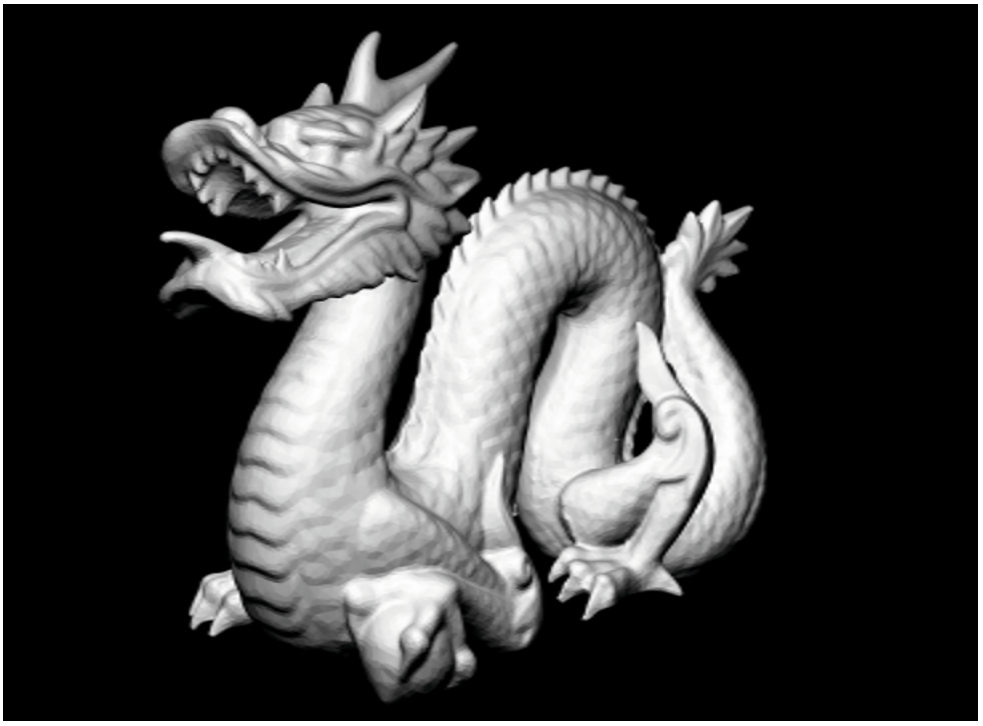
*Even if the ray **misses** everything, the bounding boxes it went through are still drawn on the screen.*



*When the ray **hits** the object, it displays the boxes it went through and the triangle it hit in a white solid color.*

Since there isn't a specific rendered visual for the traversal, *this image* represents it since it was rendered with 10 samples per pixel, DOF, soft and hard shadows, recursive ray tracing and shading in `Time to render image: 65115 milliseconds`, and with the normal naive implementation of the *intersect* we would get it in **a way longer time,** >10 minutes (that's how long I kept the *renderer* open until I was bored). Also there is a **HQ** render of the *dragon* with 400 samples per pixel (*Antialiasing* and *Shading* on) which took a little less than *20*s





Normal interpolation with barycentric coordinates

computeBarycentricCoord(...):
The point p can be represented by a linear combination of the vertices v0, v1, v2:

$$p = (1 - \alpha - \beta)v0 + \alpha v1 + \beta v2$$
$$p = v0 + \alpha(v1 - v0) + \beta(v2 - 0)$$
$$p - v0 = \alpha(v1 - v0) + \beta(v2 - 0)$$

Substitute the following:

$$A = v1 - v0,\ B = v2 - v0,\ C = p - v0$$
$$\alpha A + \beta B = C$$

Create a linear system of equations and taking the dot product of A on the top row and B on the bottom row:

$$(\alpha A + \beta B) \bullet A = C \bullet A$$
$$(\alpha A + \beta B) \bullet B = C \bullet B$$

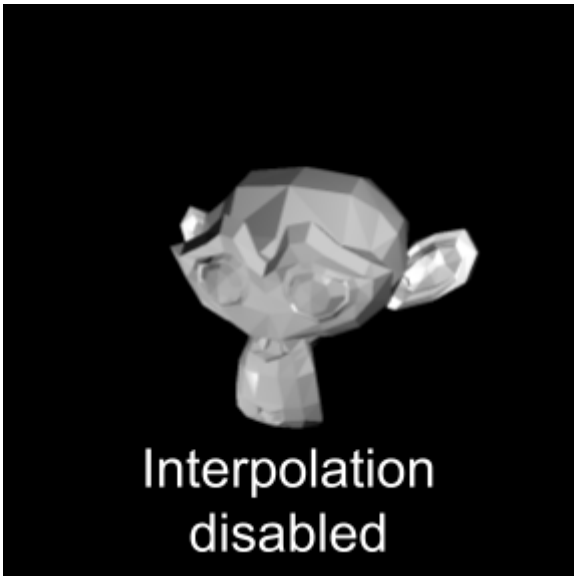
$$\alpha(A \bullet A) + \beta(B \bullet A) = C \bullet A$$
$$\alpha(A \bullet B) + \beta(B \bullet B) = C \bullet A$$

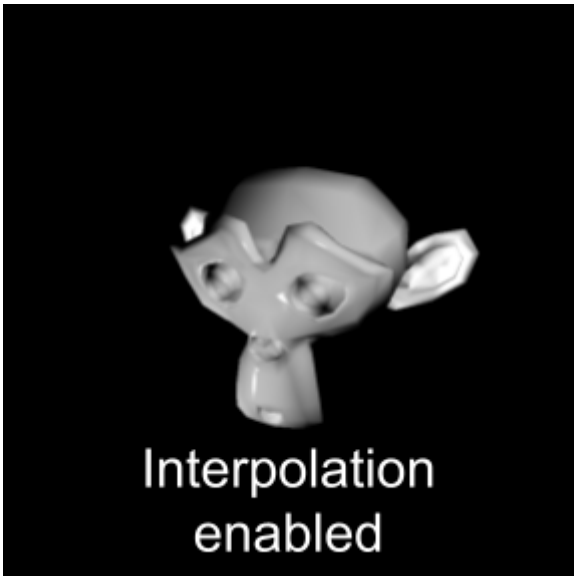
Now apply Cramer's rule:

$$det = (A \bullet A)(B \bullet B) - (A \bullet B)(B \bullet A)$$
$$\alpha = \frac{(C \bullet A)(B \bullet B) - (C \bullet B)(B \bullet A)}{det}$$
$$\beta = \frac{(A \bullet A)(C \bullet B) - (A \bullet B)(C \bullet A)}{det}$$

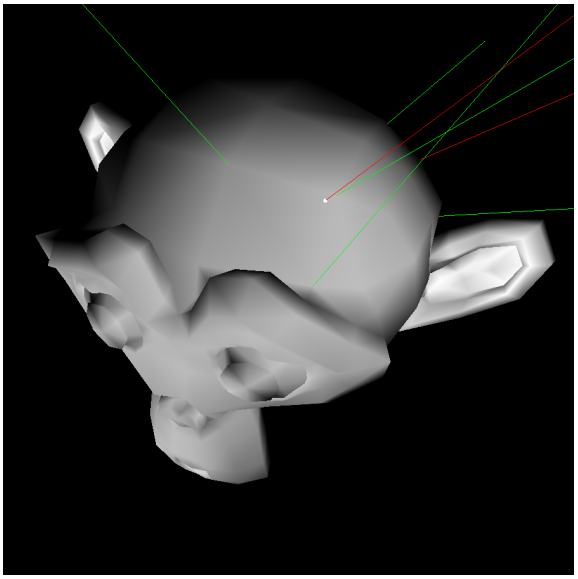
interpolateNormal(...):
It takes the barycentric coordinates and multiplies the x-coordinate with the first normal, the y-coordinate with the second normal, and the z-coordinate with the third normal. Then it adds them all up and returns the outcome.

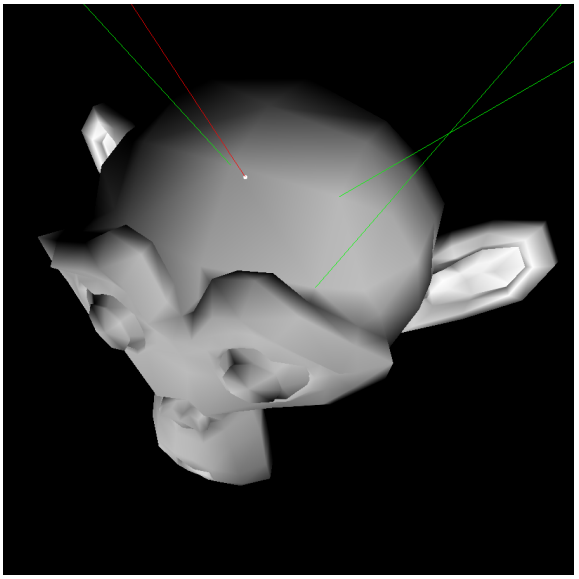
When normal interpolation is enabled, the hitInfo.normal in *intersect(...)* in "bounding_volume_hierarchy.cpp" is overwritten to be interpolated normal of all the vertices' normals.





Visual debug:
The red ray is the interpolated normal and the green rays are the normals of the vectors.



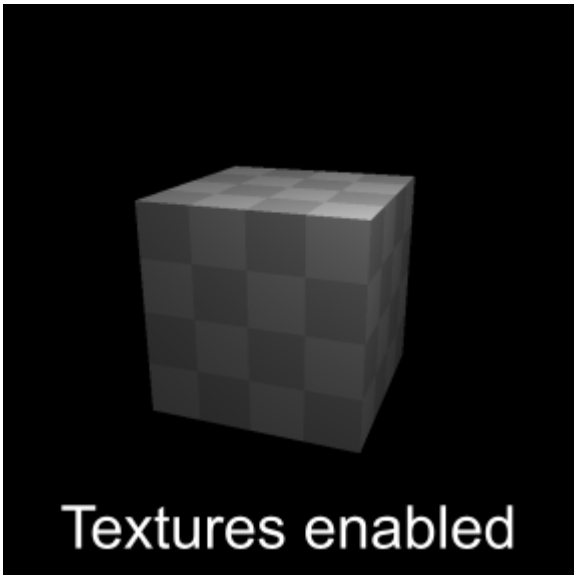


Texture mapping

interpolateTexCoord(...):
It takes the barycentric coordinates and multiplies the x-coordinate with the first texCoord, the y-coordinate with the second texCoord, and the z-coordinate with the third texCoord. Then it adds them all up and returns the outcome.

acquireTexel(...):
The texCoords range from 0 to 1. To find the x-coordinate of the pixel we multiply texCoords.x by image.width. To find the y-coordinate of the pixel we multiply texCoords.y by image.height. The pixels are stored in a 1D array of row major order. To get to the right index we multiply the x-coordinate with the image.width and add the y-coordinate, as discussed in the lecture. Then return the pixel at this index.

When texture mapping is enabled, the diffuse coefficient (Kd) is changed to the right pixel in the image.



Visual debug:

Extra features

Texture filtering: Bilinear interpolation

For this feature the method *acquireTexel(...)* in “texture.cpp” has been updated, and a new method *bilinearInterpolation(...)* has been created in “interpolate.cpp”.

bilinearInterpolation(...):
Input variables are: 4 colors of the corners, the (x,y) texCoords and the x- and y- coordinates of the four corners of the square. The method to calculate the bilinear interpolation given in the lecture, using alpha and beta, did not want to work. This had probably something to do with a plus or minus sign being flipped. The method used on wikipedia, which is more general, was used and does work.

The risk with this was however, that there would have been a division by zero in some cases. These cases are: the texCoords would lie directly on a center or on the line between neighboring centers.

In the first case it just returns the color of *x1y1*, as there is no interpolation needed. The second case has itself two cases: the texCoords lie on a horizontal or vertical line, meaning the y-coordinate is the same or the x-coordinate is the same. When the y-coordinate is the same, it only needs to linearly interpolate over the x’s. The reverse is true for when the x-coordinate is the same.

acquireTexel(...):
The method now checks first if texture filtering is disabled, if so it works just like before.

The centers of the pixels are offset from the grid by (0.5, 0.5), but if the texCoords are translated by (-0.5, -0.5), all the centers lie exactly on the grid points, which makes life easier. The minimum with zero takes care of the possible negative coordinates.

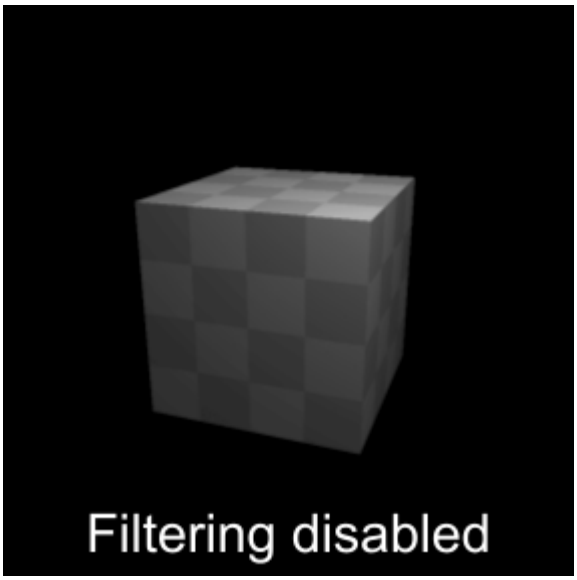
Booleans *right* and *above* are used for clarity. Variables *x1, x2, y1, y2* are initialized with the same coordinates as the closest pixel, called *center*.

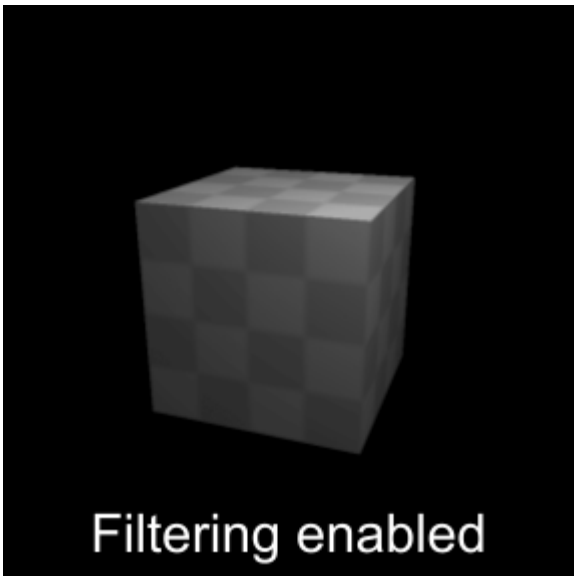
If the texCoords lie to the right of the closest pixel, then x1 will be the x-coordinate of *center* and *x2* will be one more. This could lead to a coordinate outside the image, maximum with *image.width* prevents that.

If the texCoords lie to the right of the closest pixel, then x2 will be the x-coordinate of *center* and *x1* will be one less. This could lead to a negative coordinate, minimum with zero prevents that.

The same goes for the texCoords lie above or below *center*, just with the y-coordinate.

When *x1, x2, y1, y2* are calculated, the colors of the corners can be looked up. Then just return *bilinearInterpolation(...)*.





I could not come up with a useful visual debug.

Cast multiple rays per pixel with Irregular Sampling or Antialiasing

This feature has a *long and complicated name* but can be reduced down to **antialiasing**, because basically this is what it performs. This means that instead of shooting **one ray** per pixel, we shoot **multiple and average the color** together to get a much smoother result, however this can produce a *bit of blur*. In the book there were explained *3 main methods*, and I chose to implement 2 of them, mainly, the *purely random sampling* with a uniformly distributed random number and the *hybrid - jittered sampling*, because I wasn't able to see a big difference between the 2. They are *very well described* inside the book so I will also brief them so that we come to an *understanding* about what we're about to do.

For this feature we worked mainly in the *render.cpp* file, and we created some additional fields in the camera class (*Trackball*) **-** the *number of samples* we want to use, and the *algorithm* we want to choose.

Algorithm 0 is the *random sampling* one. What we *do additionally* in the generation of rays, is that we first create a ***color** vector* that stores what each additional sample ray we create *adds to the ligh*t. Then with a *for loop* create rays based on the *sample number* with the pixel *x and y position* offset by a *random uniform float* in the interval *[0, 1]* using the *randoms.h* header. That means that the ray can start from anywhere inside the pixel - (x, y) , (x+1, y+1). After the color vector is ready, we call the *setPixel()* function which now additionally averages the color vector it receives by the number of the samples of the camera so it multiplies *color* with 1 / *camera.getSamples()*.

```
for (int i = 0; i < camera.getSamples(); i++) {

    const glm::vec2 normalizedPixelPos {
        float(x + random_float()) / float(windowResolution.x) * 2.0f - 1.0f,
        float(y + random_float()) / float(windowResolution.y) * 2.0f - 1.0f
    };
    const Ray cameraRay = camera.generateRay(normalizedPixelPos);
    color += getFinalColor(scene, bvh, cameraRay, features);
}
```

Algorithm 1 is the jittered sampling which basically means that first we split the pixel into a **grid** and inside *each grid square* we pick a point from where we start the sampling ray. The *ray creation* is done with *2 for loops*, one for the *row of the grid* and one for the *column*. Basically now the t*otal number of samples* would be the samples we set *squared* because it would make the **implementation easier**. We apply the *same principle* as algorithm 0 when generating the rays but instead for each pixel position:

- X - we add to it the *row number* together with a random float divided by the number of grid squares in the row to *get the exact coordinate*.
- Y - we add to it the *column number* summed up with a random float which is then divided by the *number of squares* in the column

```
for (int i = 0; i < camera.getSamples(); i++)
    for (int j = 0; j < camera.getSamples(); j++) {
        const glm::vec2 normalizedPixelPos {
            float(x + (i + random_float()) / camera.getSamples()) / float(windowResolution.x) * 2.0f - 1.0f,
            float(y + (j + random_float()) / camera.getSamples()) / float(windowResolution.y) * 2.0f - 1.0f
        };
        const Ray cameraRay = camera.generateRay(normalizedPixelPos);
        color += getFinalColor(scene, bvh, cameraRay, features);
    }
```

Visual debug and functionality:

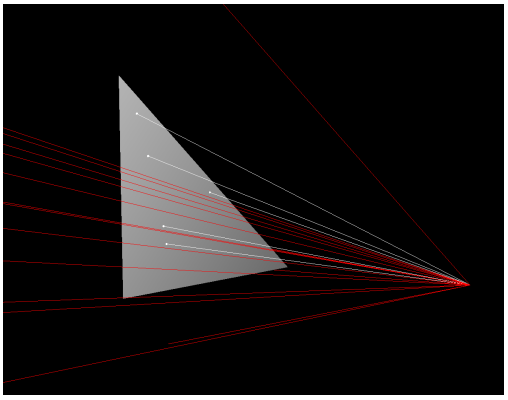
For visual activation of the feature you can find a *checkbox* under *Special Features* called **Antialiasing** which when activated creates *2 sliders*: one for the *number of samples (maximum number is 20 so - 20 for* random sampling *, 20 * 20 for jittered sampling)* and one for the *algorithm* you want to choose for the effect. **0** is *random sampling*, **1** is *jittered sampling*. We can best see the function of this when we look at the basic single triangle:

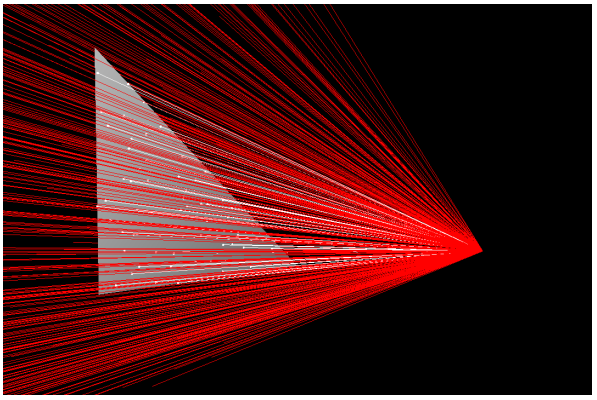
First image is triangle with no antialiasing
*Middle image is random sampling with **20** samples*
*Right image is jittered sampling with **20 * 20** samples*

At the **same number** of *samples* per pixel both algorithms perform **virtually the same;** the jittered sampling wins this representation (the middle photo contains some artifacts around the edges, it's not entirely smooth) however because it can use the maximum number of samples in the GUI (20) to generate more rays per pixel.

The GUI also contains another checkbox for debug under Antialiasing. When you press R in rasterization mode it draws the sample rays of the algorithm chosen in the direction of the cursor. The settings are als interactive and will update the view in ray tracing mode but also the debug if you press R again.

Picture on the left is algorithm 0 with 20 rays, right one is algorithm 1 with 400 rays.





Glossy reflections

This effect is achieved by taking an average of multiple reflections, each slightly perturbed from the perfect reflection ray. A perturbed ray is calculated with the *computeGlossyReflectionRay(...)* method in the *shading.cpp* file, using the technique described in the course book.

Ray computeGlossyReflectionRay(Ray reflection):
This method takes in the perfect reflection ray which is calculated in the recursive feature and returns a new ray that is perturbed compared to the original. The direction in which the ray must be altered can be calculated by constructing an orthonormal basis perpendicular to the direction of the perfect reflection r and using a random vector t:

$$u = \frac{t \times r}{\|t \times r\|}$$

$$v = \frac{r \times u}{\|r \times u\|}$$

Using basis u, v a square can be constructed that serves as the field from which the perturbed direction can be picked. The larger the square, the greater the average perturbation from the perfect reflection. Letting a be be the side length of the square, a will also be the magnitude of average perturbation and thus the amount of gloss on the surface. To pick the amount of perturbation, a random sample will be taken from the square using the Monte Carlo method. Random floats ξ and ξ' are taken using the same technique as described in Area Lights. These are then used to construct two random scalars p, q for the basis:

$$p = -\frac{a}{2} + \xi a$$

$$q = -\frac{a}{2} + \xi' a$$

By subtracting $\frac{a}{2}$ the middle of the square is automatically moved to the origin of the basis. These scalars can then be combined with the basis to calculate the direction of the perturbed ray r':

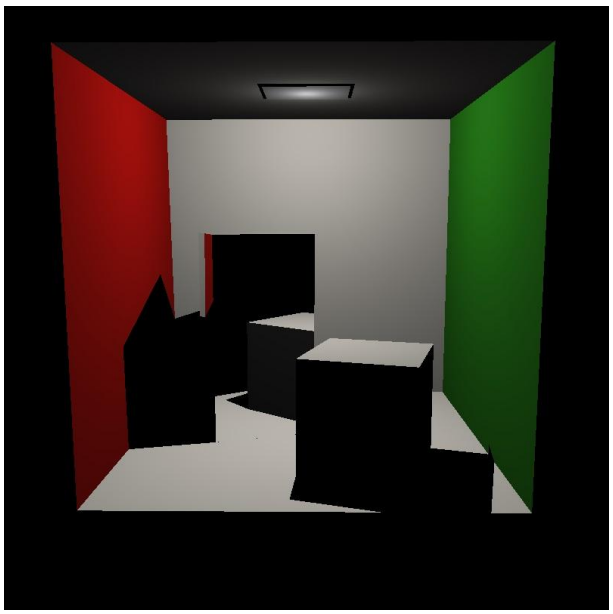
$$r' = r + pu + qv$$

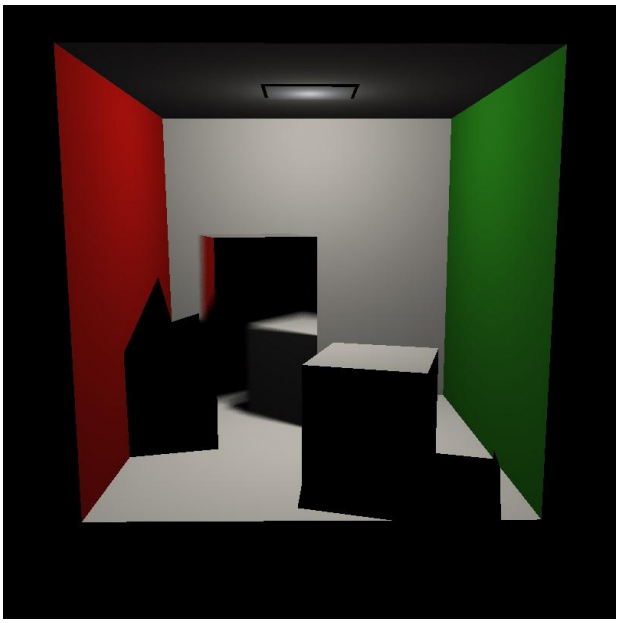
Finally r' used to construct the perturbed ray.

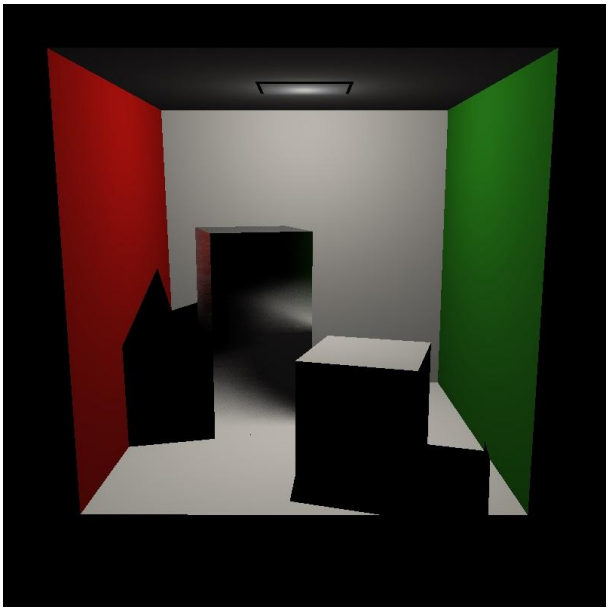
getFinalColor(...):
In this method (in *render.cpp*) an extra clause is added in the recursion section. This code will only be executed for surfaces that have a reflection. A loop is used that creates perturbed reflection rays using the previously described method and calculates their color using the same code as with the perfect reflection. The random seed for the sampling is defined above the loop.

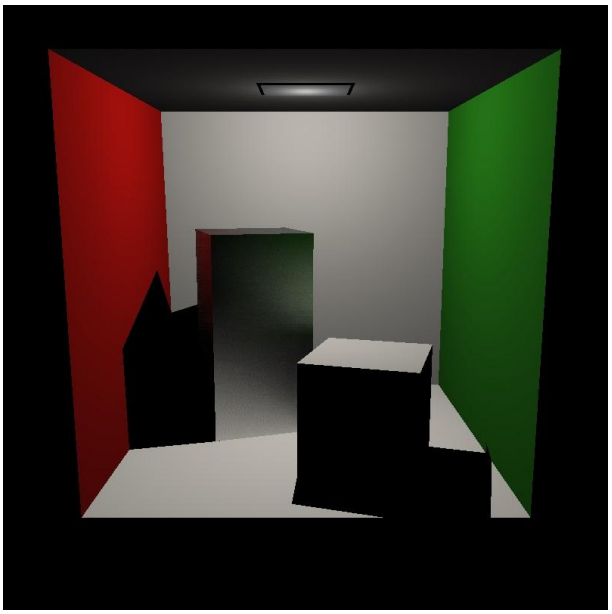
In the implementation, 1000 samples are taken to calculate one glossy ray. This number was chosen because of the observation that fewer samples lead to an inaccurate and noisy reflection that does not match the desired effect. The downside of this is the negative effect on the framerate in the live raytracing mode. However, since this is an extra feature that is not enabled by default and the ability to produce very accurate glossy reflections using a single render, it is felt that this sacrifice is justified.

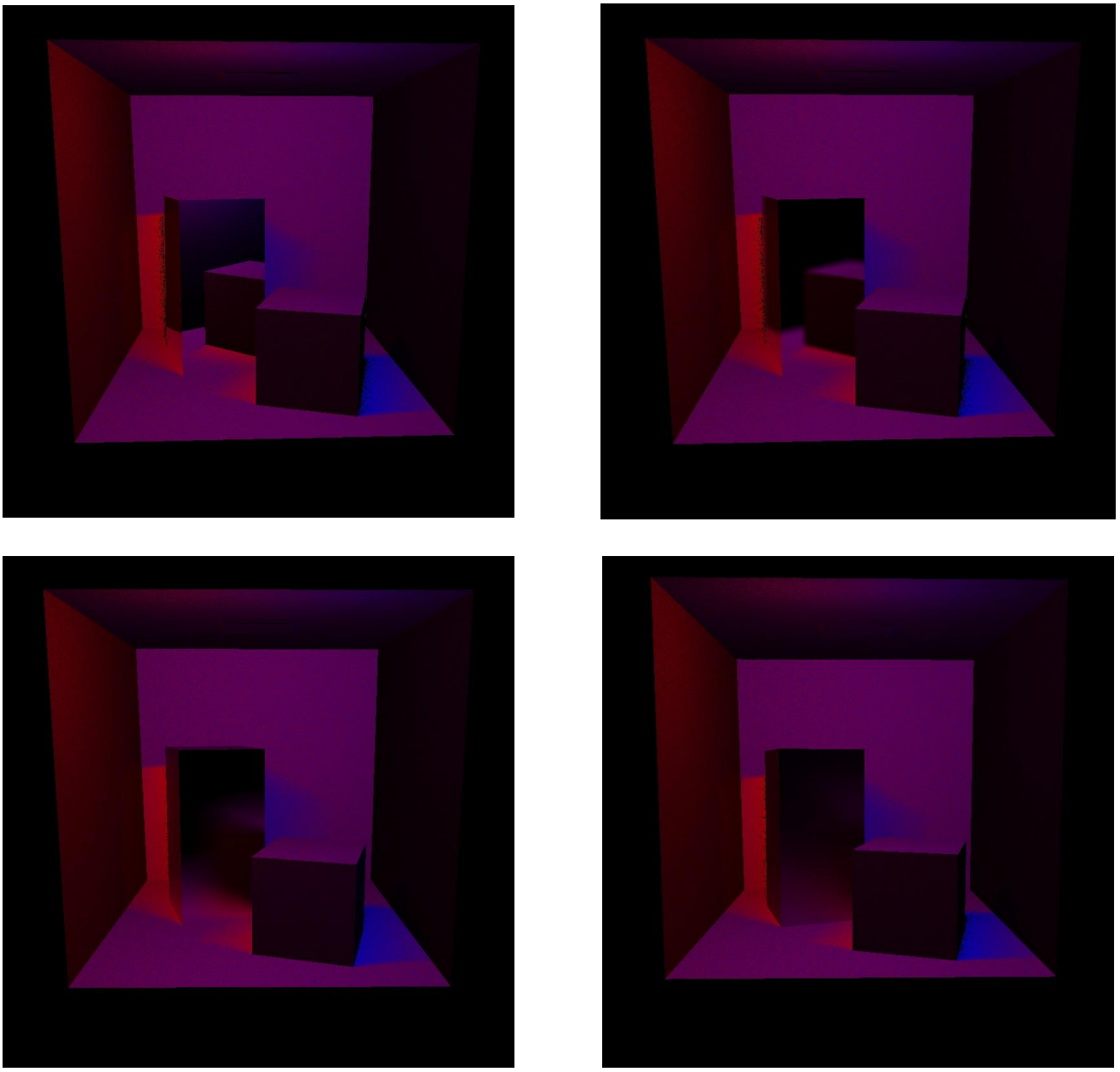
Rendered images:
Below are eight rendered images with different values of a and thus different amounts of gloss. The first image has an a-value of 0 and has thus a perfect reflection. In the second images a=0.075: the reflection is slightly blurred as if it were out of focus. The third has an a -value of 0.5 and thus a lot more gloss: the outline of the cube is barely visible. In the fourth image, with an a-value of 1.5, the effect looks more like a matte metallic surface than a mirror. The bottom four images have the same four values, only here the light source is a segment light with red and blue. The general observation is that a low a-value gives a blurred effect like an imperfect mirror and becomes more and more metallic as it increases.



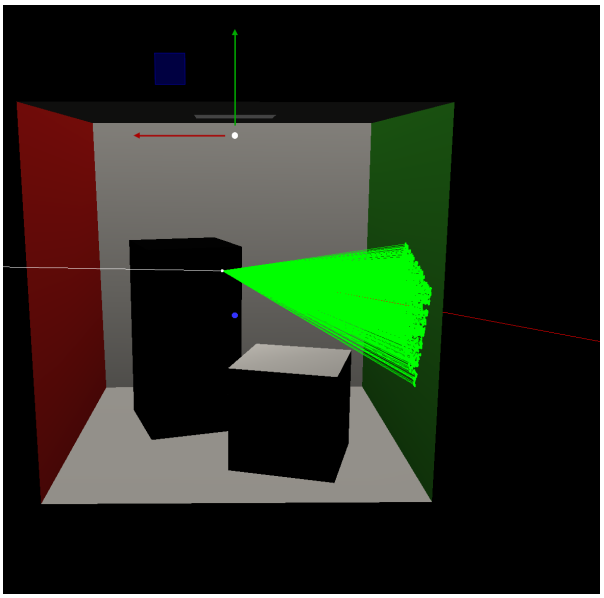


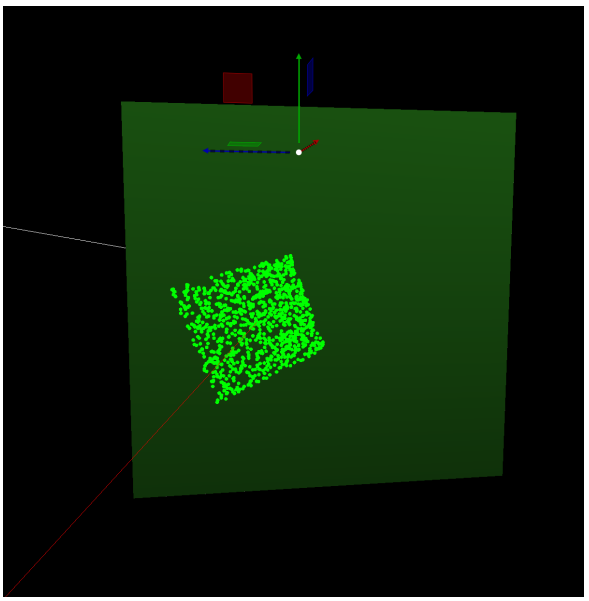






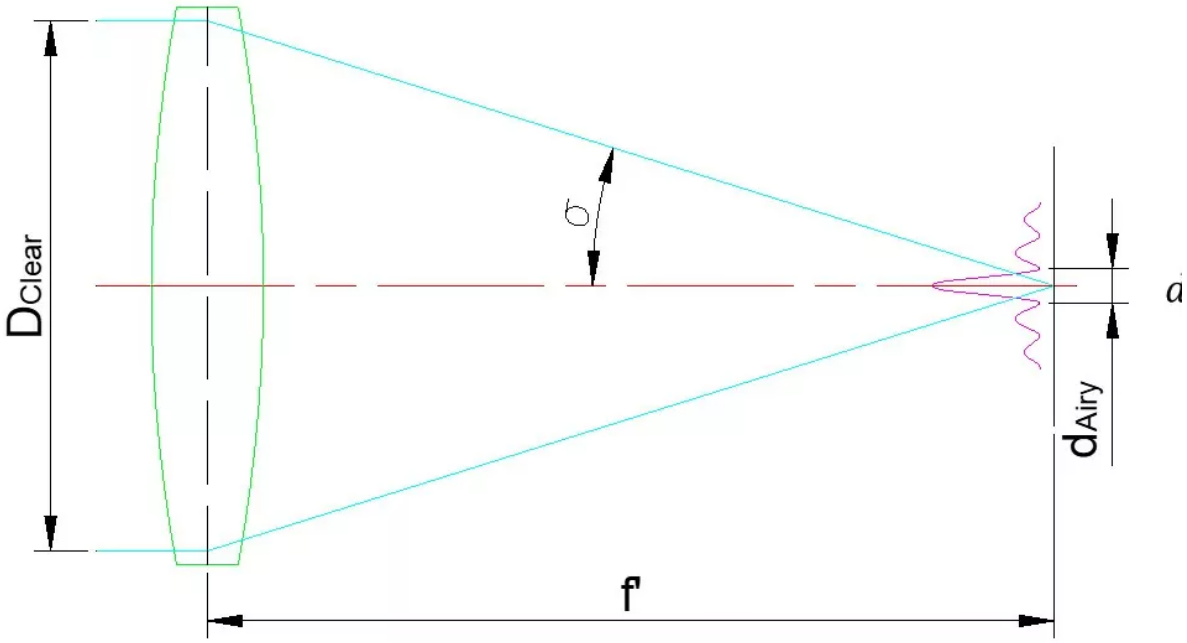
Visual debug:
The visual debug draws each calculated perturbed ray in green from the reflective surface to its hitpoint. The perfect reflection is drawn in red. Its length is infinite to make it more visible between the many green rays. The debug illustrates the sampled perturbed rays and how they deviate from the perfect reflection. Pictured left is the cluster of rays traveling from the mirror to the wall and the perfect ray in the middle. Right: the same situation seen from the back of the wall with the red perfect reflection sticking out in the middle of the square.





Depth of field

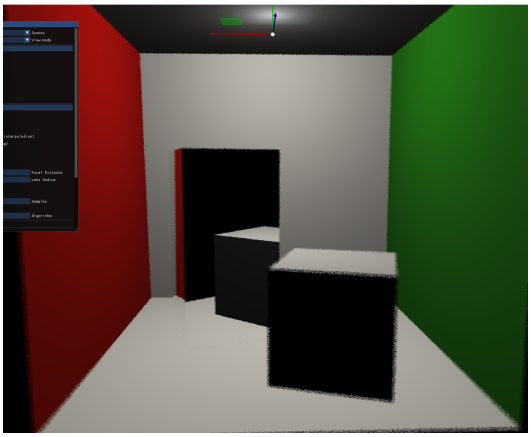
Depth of field is mainly an effect similar to antialiasing (irregular sampling of multiple rays per pixel) and both of them pair well together. Its main thing is that instead of generating rays straight on you simulate a *lens*.

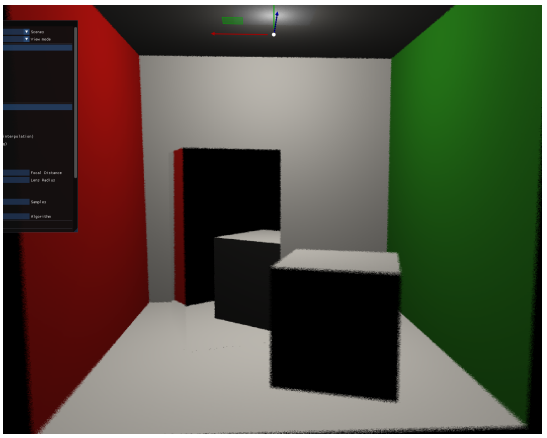


(Picture out of google images, best representative that I found, I am bad at drawing)

The *trick* for making depth of field is as explained before, for **each pixel** instead of shooting the ray straight on we imagine a lens such as the one above. Based on its *radius* and the *focal length* (the distance from the lens to the point where all rays intersect) we can shoot the rays starting from a circle and *average* the color of each of them such as in the *irregular sampling case*.

Now onto the *generation* part. *In the book* it is explained that it's enough to simulate a *square lens* since the effect is quite similar, but the complexity of making sure the lens is circular is trivial and it provides *more crisp focused points* and *more blurred unfocused* ones, **without a doubt** it's the best way of approaching this.





On the **left** is the *circular lens* and on the **right** is the *square lens*.

In this small test there isn't much of a difference, however you can see the *focus* on the cube in the reflection better in the first one and in the second one the blur is a bit more *aggressive*.

For this to work we first needed to create a way to generate uniformly random numbers, so with this in mind there have been created 2 new files randoms.h and randoms.cpp which implement **2 basic functions**:

- A **uniform random** float generator which generates numbers in *[0,1]* - *random_float()*
- A **random vector in the unit circle** generator - *random_unit()*

The trivial part of the generation of the vector comes from the math of the whole project using basic linear algebra and translations.

How this works is that first *the origin* of the ray is calculated using *2 parameters*, the *point the ray/camera is looking at*, and the *distance* to it which is calculated by the program automatically. Then we generate a vector in the origin of the system on the *negative z axis* with the size of our distance **(0, 0, -dist)** and translate it to our viewpoint using *eulerian rotation angles* and since we *can move* the camera around besides rotating it we add the position of the point we're looking at (*the blue ball in the center of the screen*). This gives us our origin if the ray would be **normal**.

Since we want to do depth of field, knowing the way translations work we can *simply* add the offset of the lens to our basic vector on the *-z axis* and let it be translated *by the rotation angles*.

```
//return a random vector in a unit circle lens (-1,1), (-1,1)
glm::vec3 random_unit()
{
    return glm::normalize(glm::vec3(random_float() * 2.0f - 1.0, random_float() * 2.0f - 1.0, 0)) * random_float();
}
```

The way we generate this vector is that first we get ourselves a random vector with the form **(x, y, 0)** since we *don't* want to add to the distance from the point we look at. Then we normalize this giving us basically a vector which points to the *circumference* of a 1 radius circle which we then multiply with a random float in the interval *[0,1]* giving us a point in the unit circle. This we can multiply with the **radius of the lens** to get the perfect way of generating the offset for the lens. The functions used for this are inside the generateRay() of the Trackball class.

For the direction we need to first get the vector which points from the *direction of the pixel in the positive z axis towards the world*. Since the coordinates for the pixels are normalized in the *[-1, 1]* interval we need to make sure that the **negative pixels** are drawn on the *left* and the positives on the *right* so the image doesn't get *mirrored*. We do this by creating a vector in the format **(-x * half_width, y * half_height, 1)**. *Why this format?* Because the coordinates of the *virtual plane* are inside the rectangle defined by **2 * half_width and 2 * half_height,** we want to generate the color for our pixel in the coordinate system that has its origin in the *center of our rectangle*.

Now, this is the **normal direction** of a ray from a pixel, but since we want to do *depth of field*, we need to find a way to use the offset created earlier to focus our ray in a point for the effect to work. If we simply leave things like this the whole world would be blurred with the strength of the *lens' radius* (it wouldn't be that different from the *antialiasing* effect).

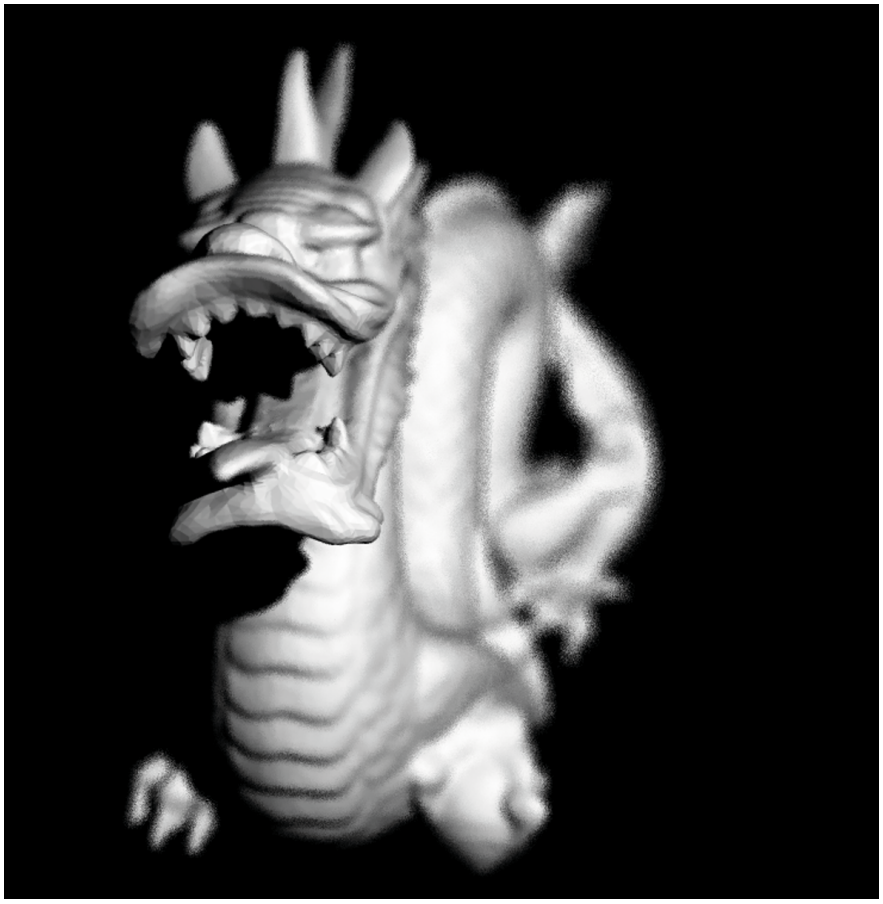
The *first observation* comes from the fact that since we added something to the origin of the ray, to get *a ray* to go inside towards the focus points we would need to **subtract something.** Since we know the offset that was added to the origin, we can just **subtract** it from the *direction vector* named earlier and we would get the rays to go towards a *point inside*, which is great but then the image would just *focus* at a distance of 1 unit from the

lens/camera.

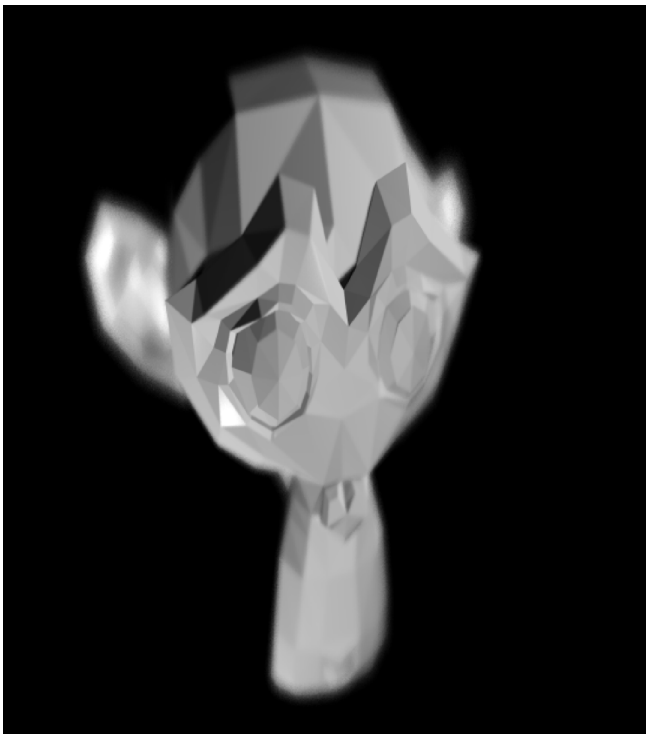
This is where the focal length comes in. A second observation comes from the fact that if the focal length is *bigger* then the point at which the *rays focus* would be at a greater distance, it means that it's **inverse proportional** to the angle of the ray.

Armed with this knowledge, we can **deduce** that if we *multiply* our offset vector with -1 / focal distance and add it to the *normal direction* vector we can get the rays to **focus** at *any point along our origin* we want. Now, as with the *origin* we translate this direction vector to our camera and the effect is done. Another *observation* we can quickly deduce from this is that if the *radius* of the lens is **0** then the camera works as our normal pinhole camera, without any other effects.

```
// Generate a ray with the origin at cameraPos, going through the given pixel (normalized coordinates between -1 and +1)
// on the virtual image plane in front of the camera.
Ray Trackball::generateRay(const glm::vec2& pixel) const
{
    const glm::vec3 offset = random_unit() * lens_radius;
    const glm::vec3 cameraSpaceDirection = glm::normalize(( -1.0f / getFocusDistance() * offset + glm::vec3(-pixel.x * m_halfScreenSpaceWidth, pixel.y * m_halfScreenSpaceHeight, 1.0f)));
    Ray ray;
    ray.debugColor = 0;
    ray.origin = positionWithOffset(offset);
    ray.direction = glm::quat(m_rotationEulerAngles) * cameraSpaceDirection;
    ray.t = std::numeric_limits<float>::max();
    return ray;
}
```



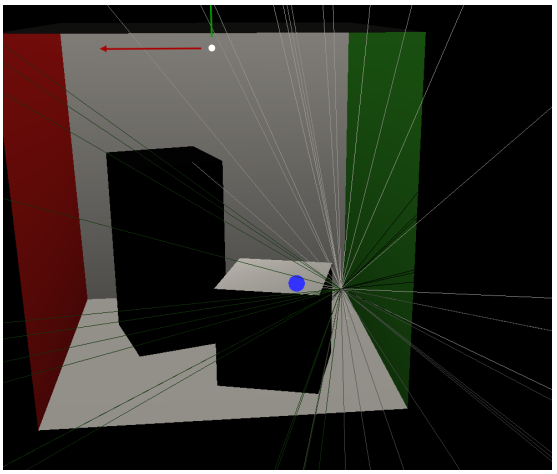
Artistic Dragon rendered with Shading, Antialising and DoF for a cool effect

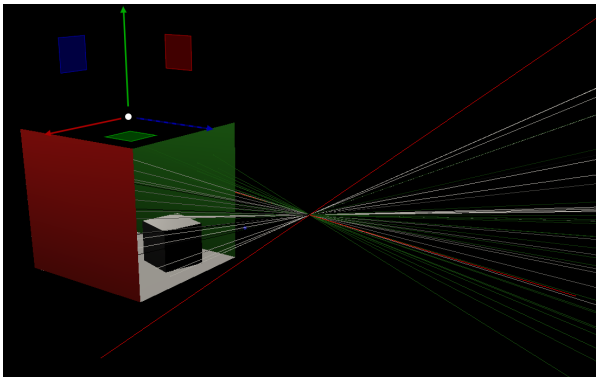


Artistic Monkey rendered with Shading, Antialising and DoF for a cool effect.

Visual debug and functionality:

For the visual debug, after activating the **DoF** option from the menu, under it would appear 2 sliders which control the *radius of the lens* and the *focal length* which can be controlled in **real time** in ray tracing view, where you can see how the rendered image changes based on the parameters. There is also another checkbox for debug. This only works in *rasterized* view. The way it works is that using **R** you can shoot 50 rays with the DoF applied to them towards the *mouse cursor*, and you can see what and how the rays get their color. *This also works in real time*. You can *change the parameters* for the DoF and see how the rays interact with the world.





DoF debug view from the front, after firing the rays, and from the side, with the focus point

Performance test

	Cornell Box	Monkey	Dragon
Num triangles	32	968	87k
Time to create	~ 50 ms	~ 50 ms	~ 250 ms
Time to render (no BVH)	180 ms	5800 ms	529000 ms
Time to render (BVH)	140 ms	140 ms	210 ms
BVH speedup	29%	4100%	251900%
BVH levels	9	13	20
Max tris p/ leaf node	1	1	1

References

Shading using Phong Illumination Model

https://en.wikipedia.org/wiki/Vector_projection

Area Lights

https://cplusplus.com/reference/cstdlib/rand/
https://cplusplus.com/reference/cstdlib/srand/
https://www.delftstack.com/howto/cpp/how-to-get-time-in-milliseconds-cpp/

Normal interpolation with barycentric coordinates

https://en.wikipedia.org/wiki/Cramer%27s_rule

Texture filtering: Bilinear interpolation

https://en.wikipedia.org/wiki/Bilinear_interpolation

Glossy reflections

Marschner, S., & Shirley, P. (2015). *Fundamentals of computer graphics, fourth edition* (4th ed.). A K Peters. https://doi.org/10.1201/b19564

Depth of field

Marschner, S., & Shirley, P. (2015). *Fundamentals of computer graphics, fourth edition* (4th ed.). A K Peters. https://doi.org/10.1201/b19564

Irregular sampling

Marschner, S., & Shirley, P. (2015). *Fundamentals of computer graphics, fourth edition* (4th ed.). A K Peters. https://doi.org/10.1201/b19564