CG Ray Tracing

Jindrich Pohl 5062489, Mert Gokbulut 5044065, Alexandru Bobe 5069831

October 2020

1 Introduction

2 Shading

2.1 Preparation - hitInfo's normals

Before working on the shading, it is important to make sure that hitInfo's normals in ray_tracing.cpp are set correctly. In this case, each triangle will have just one colour and one normal corresponding to the plane of the triangle. In intersectRayWithTriangle, a triangle plane is created for the triangle that is currently being intersected. This means that when a ray intersects the triangle inside this plane, the hitInfo's normal can almost directly be assigned to this normal. However, the normal must always be facing "towards the camera" - in other words, the dot product between the normal and a vector from the intersection point towards the camera should always be greater than 0. If that isn't done, some triangles will be illuminated even if they should not, because from the angles for diffuse and specular shading, the program will think that the triangle should be illuminated when it shouldn't. See figure 1.

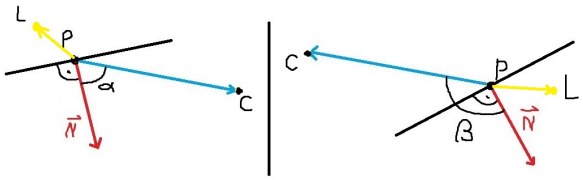

Figure 1: Normal facing vs not facing "towards the camera"

Figure 1 describes the difference between a correctly facing normal versus one that is facing in an incorrect direction. On the left, we see the case when $\alpha < \frac{\pi}{2}$ or $\alpha > \frac{3\pi}{2}$. Here, the normal is facing in the correct direction, because the absolute value of the angle with the PC vector does not exceed the right angle ($\frac{\pi}{2}$). If we are at C and the light is at L, from our perspective, P should not be illuminated. On the right, the angle β exceeds $\pi/2$ in absolute value, so

the dot product between the normal and the PC vector would be negative. If the first normal we compute for the triangle plane is the one on the right (i.e. if the dot product of the two normal and the PC is negative), this normal's sign must be flipped in the hitInfo for the shading to be correct. Otherwise, like in this case, the triangle would seem to be illuminated when in fact, it is not.

2.2 The shading itself

The logic for shading was done in a new "shading" function to make the code more organised. This function is called from the getFinalColor function when there is an intersection. The idea is the following: for each light in the scene, compute the diffuse and phong shading from the available parameters (ray, hitInfo, light), and add their sum to the final colour that the current intersection point will have. The final colour is set to black by default and is made brighter whenever a nonzero shading colour is computed with a light source for this intersection point.
Here, diffuse shading is the diffuse component of the Phong model with as formula $I_d * K_d * cos(\theta)$, and specular shading is the specular component of the Phong model with as formula $I_s * K_s * cos(\theta)^n$. It is important to note that when any of these cosines (calculated by dot products) is smaller or equal to 0, no colour is added from the current point light - the triangle is either facing away, or we are looking at it from a wrong position (no specularity will be visible).

2.3 Visual debugger

See figure 2 - the white ray has been shot, the blue ray is the normal and the red ray indicates that the face should not be illuminated by this light.

2.4 Rendered image

See figure **??** - we are looking at the cornell box from the bottom left. We see that the bottom and left sides of the cornell box remained black to us, the green side has the correct diffuse shading, and the mirror is slightly reflecting the light. Note: this rendering was done before the recursive ray tracer was implemented.

3 Recursive Ray Tracing

3.1 Introduction

Recursive ray tracer allows us to use the specular component of the materials to reflect and refract light to give glass and mirror like behavior. In this project, we only implemented the reflections. The principle behind the recursive ray tracer is to trace the ray reflected from the surface. The reflected ray will give us mirror-like behavior. As you see in figure 4, we can figure out the reflected ray by knowing the incoming ray and the normal of the surface. The reflected

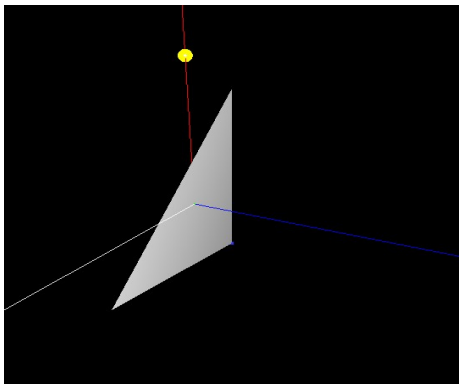

Figure 2: Side facing away from the light - should become black when ray tracing

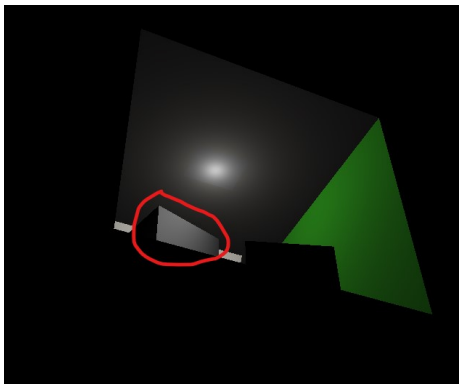

Figure 3: Cornell box - position from which we see the mirror's specularity

ray equation is $r = i - 2 \cdot (n \circ i) \cdot n$, where i is the incoming ray, and n is normal from the surface.

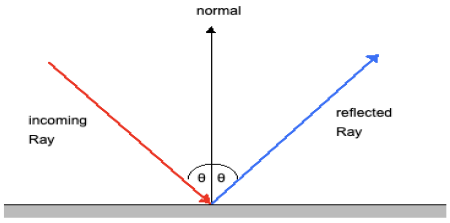

Figure 4: Normal facing vs not facing the light

3.2 Algorithm

Recursive ray tracer shoots one ray for each of the pixels and traces each of these rays to create the scene. There are two functions that repeatedly call each other to trace the rays recursively. The two methods are called **trace** and **shade**. Trace method is used to check if the given ray intersects a mesh in the scene. If the given ray intersects a mesh, it calls a method called shade to calculate the color value of the surface. If not, the color is returned as the background color. Trace and Shade method work together until the ray doesn't hit anything or recursion threshold is reached (which is checked in the trace method). If Ray doesn't hit anything or the number of reflection is exceeded, the shading function will return the background color which then can be added to the final color of the pixel. When the recursion unravels, the colors of reflec-

tions are added to the final color value depending on their specular component of the surfaces the rays hit. This is how the reflection is calculated on specular surfaces. The direct color of the surface is calculated using the shading function which is described in section 1.

3.3 Optimisations

By itself, the designed algorithm works fine, but we can tune it to be more efficient. In the final version of the algorithm, before it recursively calls the trace function from the shade function, it checks if the surface that was interfered with has a specular component which is significant. If not, it won't trace anymore since the final color of the pixel wouldn't be affected by the extra calculation. In our testing, this reduced the time taken to render a scene by one third.

3.4 Problems encountered

When we designed the algorithm, we realized that the rendered scene has black spots (noise) that were not getting rendered properly. Once debugging it, we realized that the reflected rays were interfering with the surface that they were supposed to bounce from. To fix this issue, we moved the origin of this reflected ray by an epsilon (around 0.001) in the direction it was supposed to go in. This solved the interfering problem and fixed the algorithm which now creates a crisp rendered image.

3.5 Visual Debugger

In this section you can see the ray is bouncing in the scene on figure 5, figure 6 and figure 7. The noise explained in description of the algorithm can be seen in the figure 8. Some of the rendered images of the final algorithm can be seen in figure 9 which took 508.005ms when maximum number of bounces equals to two.

4 Hard Shadows

An intersection point P being in shadow can be identified when a ray shot from that point towards a point light L intersects an obstacle before reaching the light. Whenever a point is in shadow, the shading for the current light is ignored. If multiple lights illuminate the same point, even if the point is in shadow for one light, it can still have some colour thanks to the other lights.
When a ray is shot from P to L, we must move the ray's origin a bit in the direction of PL - otherwise, the intersect function would think that we are intersecting P itself and all points will be in shadow. To shift the ray's origin O by some epsilon, we normalise PL and add an epsilon multiple of it to O.
The ray we shoot is directed towards the light, but has infinite length - after

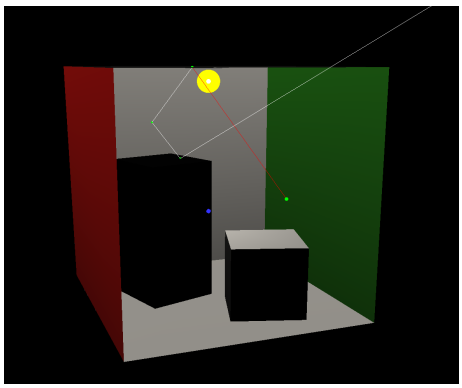

Figure 5: Debug Ray bouncing

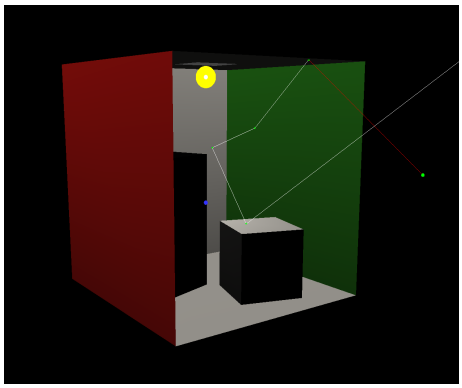

Figure 6: Debug Ray bouncing

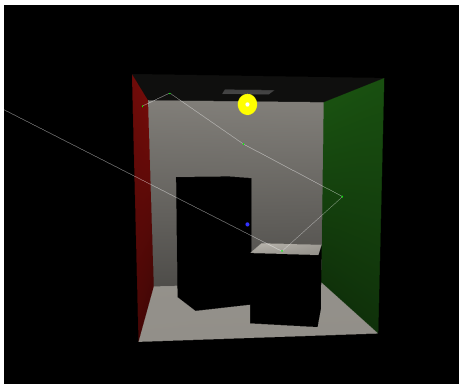

Figure 7: Debug Ray bouncing

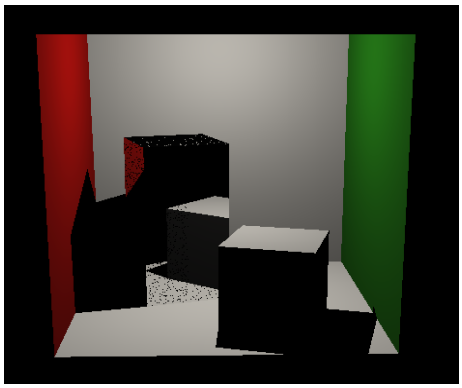

Figure 8: Noisy Render

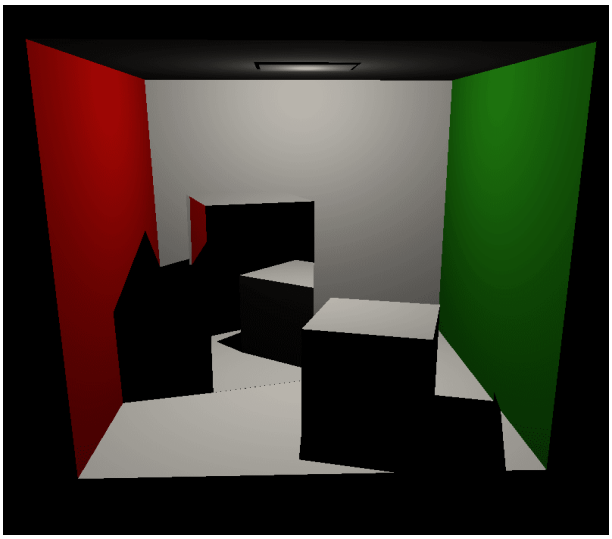

Figure 9: Recursive Ray Tracer Render

calling the intersect function, we get a distance to some intersected point B (or infinity). Iff $|PB| < |PL|$, the point is in shadow. Finally, we must make sure that we are passing only some dummy hitInfo to the intersect method.

4.1 Visual debugger

White ray: has been shot. Green ray: intersected something. Blue ray: no intersection. Figure 10 - self-explanatory. Figure 11 - there was an intersection, but behind the light, so P should still be illuminated. Figure 12 - there was no intersection in the ray from P towards L; P should be illuminated.

4.2 Rendering

Figure 13

5 Soft Shadows

Soft shadows are easily described as the region in which only a portion of the light source is obscured by the occluding body. The first idea when hearing this definition is to start shooting rays from the light source to a point in space and see if it is obscured by any body. But, the chance to find the good direction for the ray to hit that point is almost zero. Therefore, the solution to this problem was to shoot rays from the point in space to the light source and see if it was obscured by anything. Then, I had to choose a number of points on the sphere and a way of sampling the sphere. I chose to have the sphere divided in 200

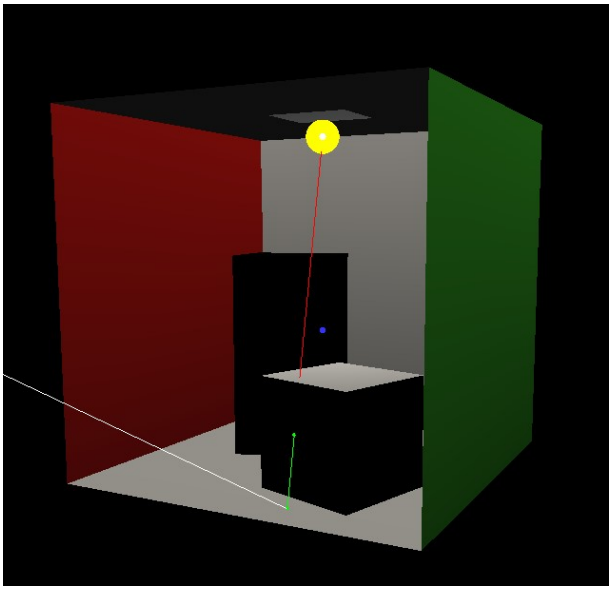

Figure 10: Point in shadow

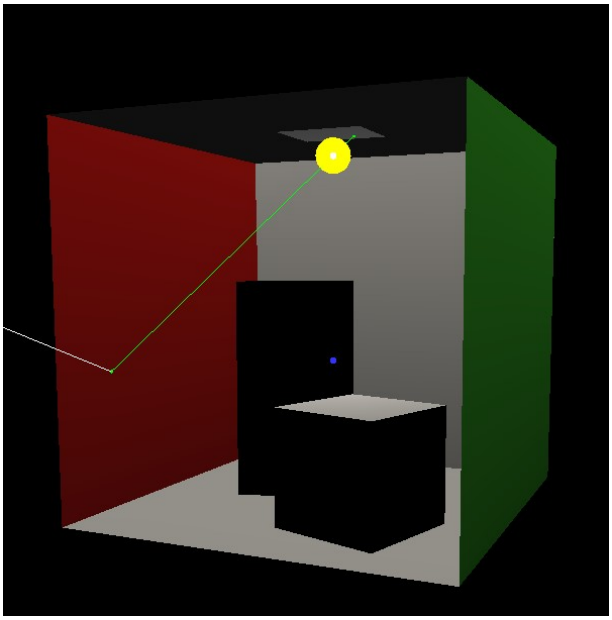

Figure 11: Intersection behind the light - point illuminated

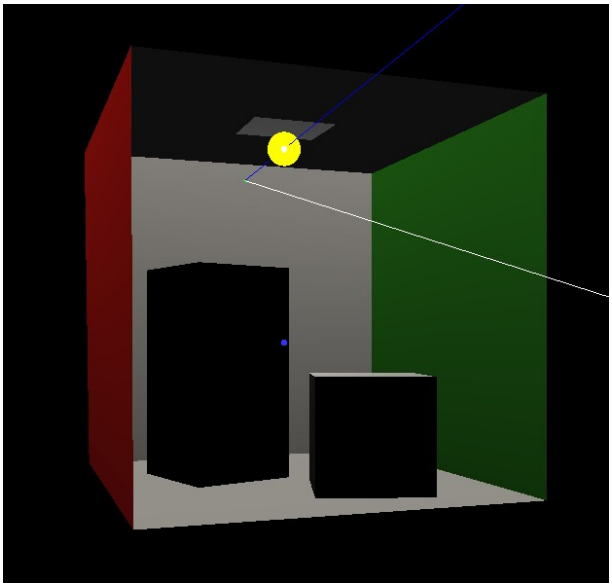

Figure 12: Blue ray - no intersection at all - point illuminated

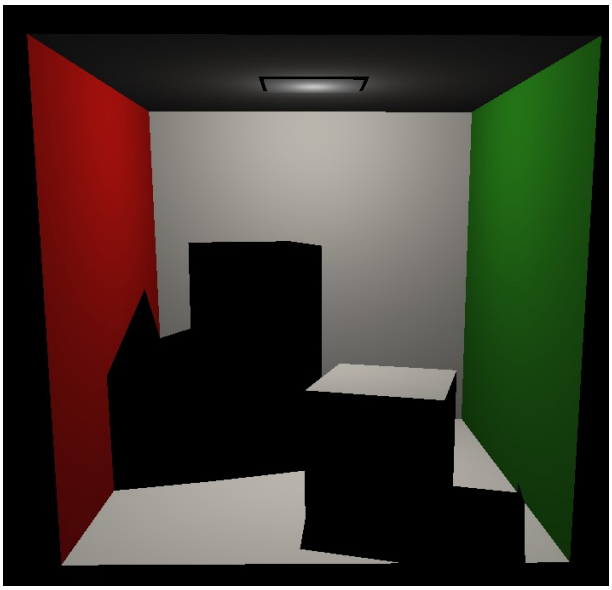

Figure 13: Rendering of the cornell box with only hard shadows

points and to use an uniform distribution on the surface of the sphere, which means that I took a direction vector with x,y,z components normally distributed with mean=0 and standard deviation=1 and I scaled this vector with the length of the radius in order to get these points. As a way to verify that until now my method was working, I drew the rays that I was shooting. The rays that do not intersect anything except the sphere should be white and otherwise red. 14

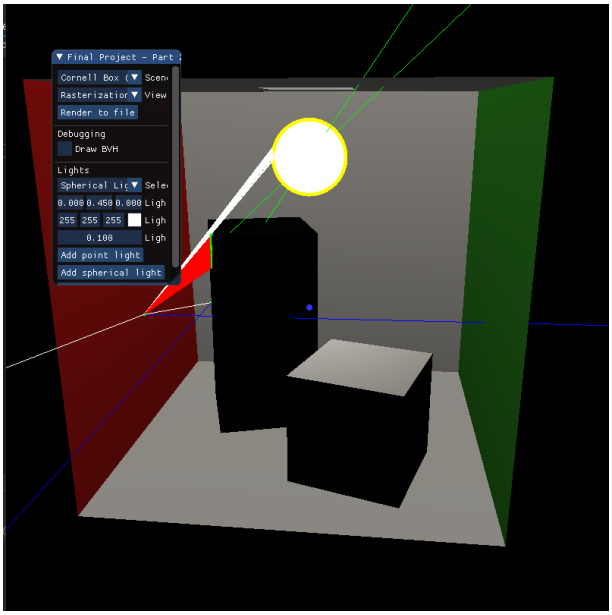

Figure 14: Debugger for the rays

As I was shooting the rays, I also kept track how many of these rays intersected the light and nothing in between. At the end, I divided this number with the total number of rays shot from that point(200) and I got the shadow: darker if less rays intersected the light or lighter if more points intersected the light. 15

6 Acceleration Data Structures

6.1 Introduction

The point of using Acceleration Data Structures is to make our ray tracer run faster and more efficiently. For this, we implemented a Bounding Volume Hierarchy (BVH). The BVH is a tree-like structure where each node is an Axis-Aligned Box, and the children of each node are contained within the Axis-Aligned Box of its parent.

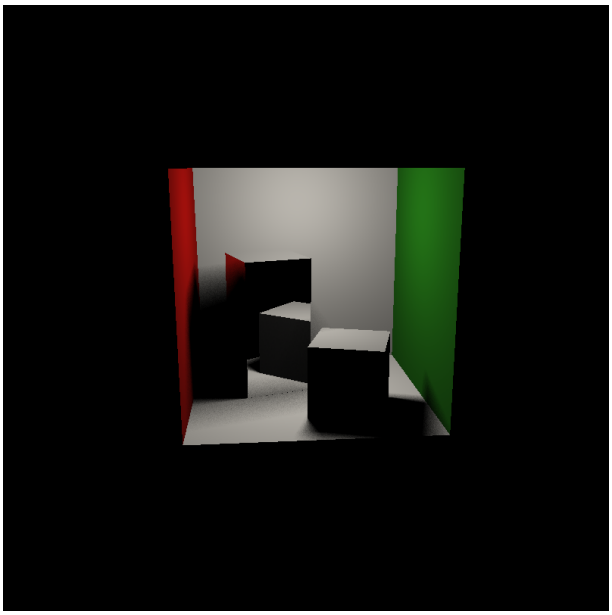

Figure 15: Soft Shadows rendered

6.2 Creating BVH

As we talked about in the introduction, it is a tree-like structure that is composed of nodes. To be more precise, it is a binary tree. In the BVH structure, all the nodes are stored in one large vector.

6.2.1 Nodes

Each node contains information about its child nodes which is a vector of integers pointing at a position in the nodes vector. Nodes also contain a boolean to indicate if they are leaf or not, an integer indicating the level in the tree, a vector of custom meshes (explained later), and an Axis-Aligned Box which encapsulates the meshes.

6.2.2 Creating the Structure

After defining what the nodes are storing, we can design the algorithm. We need to design an algorithm to create this tree-like structure. For this, we created a method called **createTree** which starts from the root node (which contains all the meshes in the scene), then splits the root node into two child nodes using the **getSubNodes** method and adds these children to the nodes array defined in BVH structure.

The **createTree** method is similar to a breadth-first search, with the difference being that we use an std::vector instead of a queue and the nodes are

being added to the tree as we traverse it. We start with only the root in this vector. The algorithm traverses one element at a time (we must keep an index pointing to the position we are at). Each element we have not yet traversed is an incomplete node, because it has no indices pointing to its children. Every time we move further in the vector and the node we are considering is not a leaf, we call **getSubNodes** to get its children, we add these incomplete children to the vector so that they can be processed later, and we set the correct indices for the parent node to point to its children. We stop traversing the vector once we reach the end of the vector of nodes and the current node is a leaf. This leaf mark is given in the getSubNodes method. A node is marked as a leaf if the desired maximum depth is reached, or there is only one triangle in the node which can not be divided anymore.

6.2.3 Dividing Meshes

Now we know how the tree is formed, we need to implement a method to divide the node. What we want is to get half of the meshes in one sub-node and the other half in the other sub-node. There are two possibilities for this. Either the node that we want to divide contains a single mesh or it contains multiple meshes. If it contains a single mesh, then we need to split the triangles of that mesh in 2 groups and create a mesh from each, and if it contains multiple meshes, we can divide the meshes into 2 groups. So how can we group the meshes so that two of the subnodes will have approximately equal axis-aligned boxes? We calculate the longest axis of the Axis-Aligned box that contains all the meshes in the node we want to divide. Then we sort the meshes using the centres of their median triangles according to how close to the origin they are in the longest axis (x/y/z). We acknowledge that this approach might not be the best to divide since we are not taking the sizes of the meshes into account. By taking the first half and the last half of the sorted array we can get two nodes that are generally about equal in size (size of Axis-aligned Box). This step is computationally expensive and might not be the most efficient in dividing the nodes however the construction is used once at the beginning of the scene so this is an expense we can afford. The idea for dividing the one mesh is similar. We just sort the triangles by their centers and divide the triangles array into two to create 2 meshed from it. This process continues until we hit the desired maximum depth in the tree or there is only one triangle in a node that can not be subdivided anymore therefore a leaf. The node structure of the tree can be visualized in figure 21 where each picture shows a different level of the Axis-aligned boxes of the nodes.

6.3 Intersecion method using BVH

The implementation starts by checking that there are some meshes in the scene - if that is not the case, no intersections take place in the structure, and only spheres can possibly be intersected.

6.3.1 Initialising the recursion

There are 2 possibilities when we should try entering the root and checking for intersections in it: the ray that is shot can intersect the root's bounding box or it can start inside the box (or both). If any of these conditions is met, we can enter a recursive function that will try to go deeper and deeper in the tree to eventually end up in a leaf and intersect triangles in that leaf.

6.3.2 Leaf intersections

In the case that the current node is a leaf, we can simply reuse the code that was originally in the intersection function, i.e. go through all the meshes in the current node, for each mesh, go through all the triangles, and try intersecting the ray with all these triangles. If the ray intersected the triangle, we update its t value, make sure the intersection method will return true (*but only after all triangles were intersected - if in fig 16, we first check against the right triangle and would directly return true, the ray's t would be incorrect*), and continue intersecting. If it did not intersect any triangle, the ray's t simply stays at infinity and we return false.

Figure 16: Hit should be true but should not directly return true

6.3.3 Non-leaf intersections

When the current node is not a leaf, things become more complicated. First, we get some initial information to work with.
We attempt to intersect the children's axis-aligned boxes and store the t values of these intersections. After each such ray-box intersection, we reset the ray's t to its previous value. If we would keep it, the intersections with the contained triangles or the child nodes' bounding boxes would break. This is because the intersection functions should ignore an intersection if the ray already has a smaller t, and since triangles and child AABBs are always contained in parent AABBs, the ray's t from the parent intersection will always be smaller. In short, without resetting the t, no triangle would ever be intersected. We store tLeft as the t for the intersection with the left box and tRight as the intersection with the right one. Both are initially set to -1.0f: if this value remains after trying to intersect the ray with one of the boxes, it means that the box was not hit by

this ray.
Then, we determine whether the ray starts in any of the boxes. This is done simply by comparing the ray's origin's coordinates against the coordinates of the lower and upper points of each bounding box. It can start in both if they overlap, in only one of the two (without any possible assumptions on overlapping) or in neither.

Now, we have information about which child boxes were intersected, and if the ray starts in any box. We can categorically check against all the possibilities and see which applies to the current situation.
First, we can check whether the ray starts inside any of the boxes. If the ray we shoot has "infinite" length, this is irrelevant, because at some point, we will end up intersecting the box anyway, but if the length is defined, we might not intersect the bounding box itself, but we might intersect some triangle inside this bounding box. For example, in soft shadows, rays that are passed to the intersection function already have a defined length (the distance between the intersection point and a point light). The cases are the following:

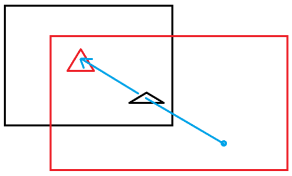

Figure 17: Both the box in which the ray starts and the one that it intersects must be checked

- the ray starts in both child boxes - this means that we must call the intersection function recursively for both children
- the ray starts in only one of the child boxes (let us call this box B) - in this case, we prioritise the node with box B when recursively calling the intersection function. If we intersect a triangle inside the node of B before even touching the other box with the ray, we do not have to check the other box at all. Figure 17 shows the worst case - here, prioritising does not help. However, if the ray would be short enough not to touch the black box at all, or if a triangle was intersected before reaching the black box, we could safely ignore the black box.
- the ray starts outside of the two child boxes

If the ray starts outside of the two boxes, the following cases show up:

- the ray does not intersect any of the children - return false

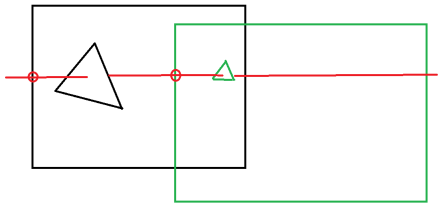

Figure 18: Black box intersected first, green can be ignored

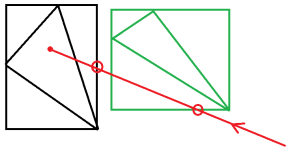

Figure 19: Prioritise green, but then also go for black

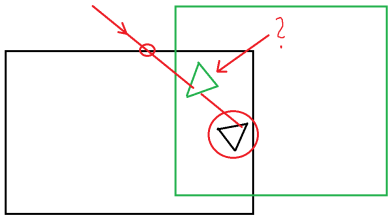

Figure 20: Black prioritised and intersected, but green must also be explored

- the ray intersects only one child - call the intersection function just for this child
- the ray intersects both children - when doing recursive calls, the child that was intersected first should be prioritised. In figure 18, we intersect the black triangle way before intersecting the green box, so the node of the green box can be ignored. However, we might also have to intersect both children as is shown in figure 20 - here, we intersected a triangle in the black box, but we entered the green box before that, so we also need to check the green box. Finally, in figure 19, we see the case when the black box must be checked after checking the green box simply because there was no intersection in the green box. The left child is intersected first when $tLeft < tRight$ and vice versa for the right child.

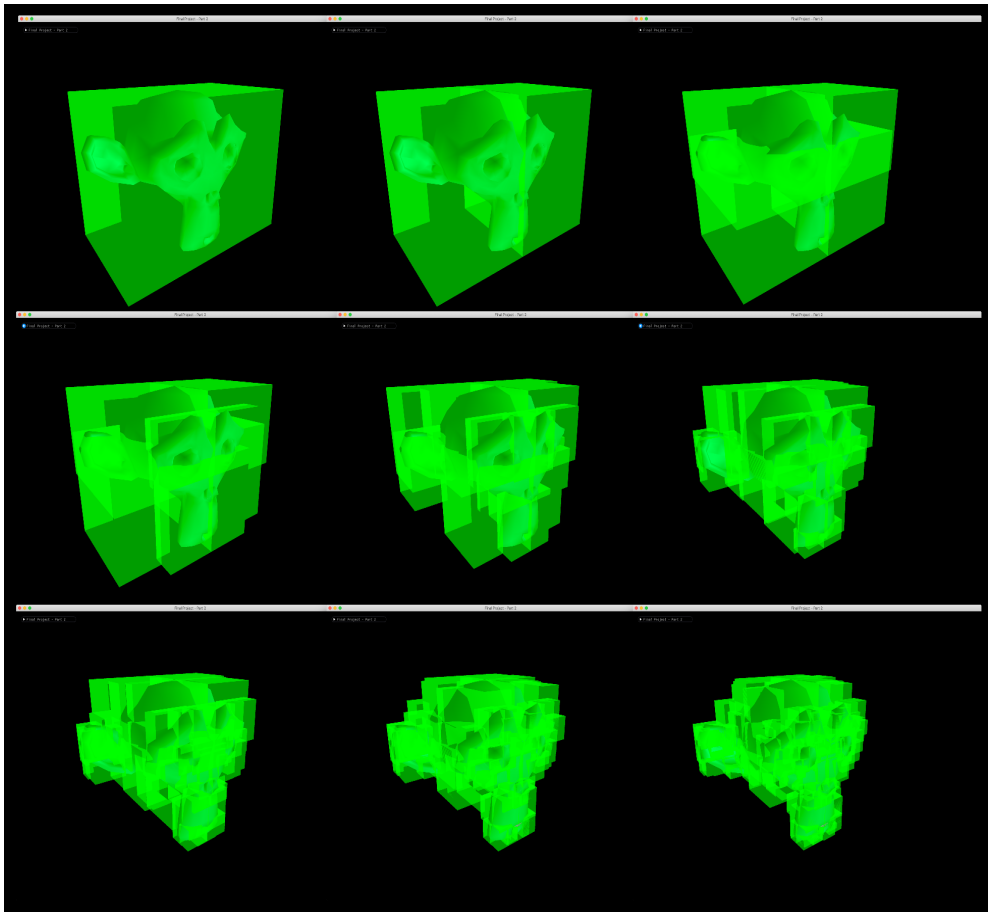

Figure 21: Nodes in each level of the BVH tree structure

7 Extra features

7.1 Interpolated values over the triangle via barycentric coordinates

Interpolated values over the triangle via barycentric coordinates makes an image made from triangles to look smoother. I decided to interpolate the normals of the triangle. In order to achive this I used the following formulas from the lecture:22 In my case, "p" was the resulting interpolated normal, after multiplying every

$$p = \alpha v_0 + \beta v_1 + \gamma v_2$$

$$\alpha = \frac{A(pv_1v_2)}{A(v_0v_1v_2)}$$

$$\beta = \frac{A(pv_0v_2)}{A(v_0v_1v_2)}$$

$$\gamma = \frac{A(pv_0v_1)}{A(v_0v_1v_2)}$$

Figure 22: Barycentric coordinates

vertex's normal with their weight. The areas were calculated with the help of the cross product. 23

7.2 Bloom filter

For the bloom filter, I used the method described in the lecture: I firstly stored only the pixels with a value above the threshold. After several trials, I chose the value for the threshold to be equal to 1. That means that I only stored the pixels with R + G + B greater than 1. Than I blurred the obtained image, by going 10 pixels up, 10 pixels down, 10 pixels to the left and 10 pixels to the right for every pixel and average the values. When this was done, I applied the new image to the first image, by summing all the values.24

7.3 Motion Blur

As its name states, motion blur happens in the presence of motion. As our objects are not moving, we have to create the motion ourselves. I chose to create this movement by changing the camera's parameter "lookAt". Then,

Figure 23: Monkey with interpolated normals

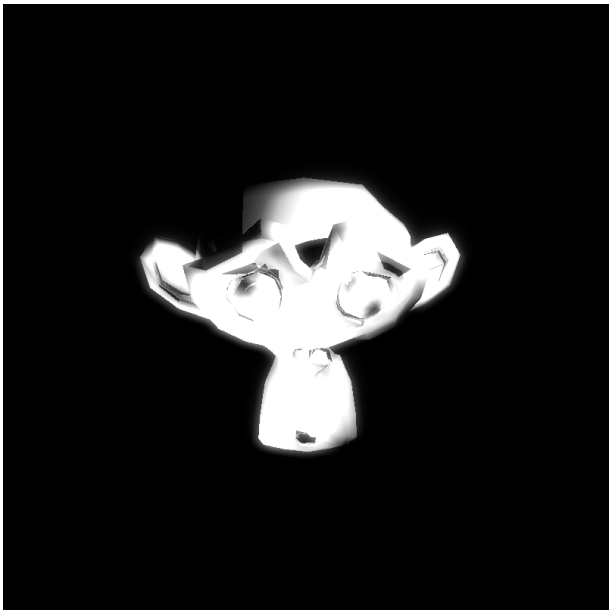

Figure 24: Monkey with bloom filter

after trying different values for "lookAt" I chose to move it every time by 0.01 on the x axis. Trying different numbers of scenes, I came to the conclusion that 16 scenes look the best, also taking into account the computational time. At the end, I added all the pixels' values from all the scenes and averaged them. In order to get more information about the motion blur, I read the following publication:Motion Blur 25

Figure 25: Motion blur

7.4 Anti-Aliasing

Anti-aliasing helps to minimize the distortion in the rendered image so the lines look more straight and crisp. Anti Aliasing can be implemented in different techniques which are pre-filtering, post-filtering, and pixel phasing. We implemented using post-filtering which is also known as supersampling. Post-filtering treats the screen as if it was 4 times larger in resolution, then it averages the 4 pixels for 1 pixel so we get a more accurate color value for it. We opt for supersampling because we wanted our Anti-Aliasing algorithm to make the rendered image as crisp as possible. We could have used FXAA which would be faster and may use less memory however it wouldn't be as crisp. This functionality can be used by checking a box on the menu. Figures 26 and 27 show pictures with anti-aliasing and without it. It is not immediately apparent, but if you look at the lines on rectangles, you can see that with anti-aliasing enabled, they look more straight.

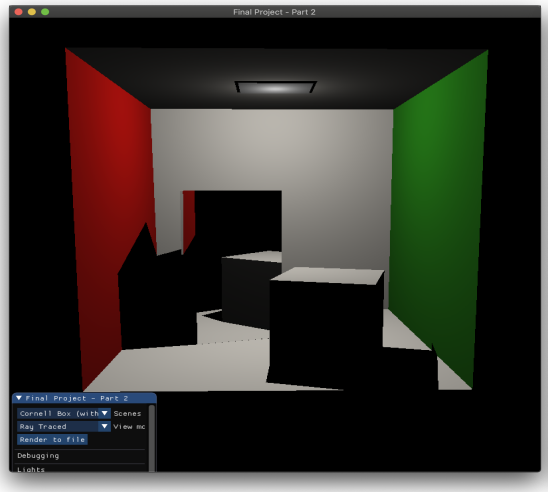

Figure 26: Rendering without Anti-Aliasing

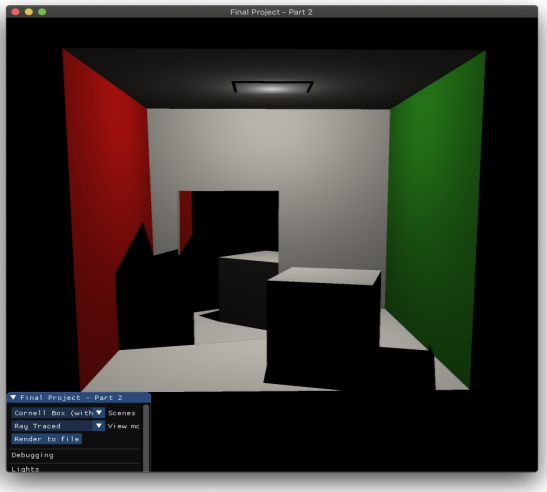

Figure 27: Rendering with Anti-Aliasing

8 Work Distribution

9 Performance test

See table 2 and the rendered images at the end of this report. Note: Jindrich's model was throwing an exception even after recomputing normals and triangulating faces when exporting, so his render is unfortunately not included.

9.1 Rendered images

10 Competition

The last figure of this document is the one we would like to send to the ray tracing competition. Anti-aliasing, bloom and interpolated normals were the things used in it.

Table 1: Work Distribution

Features	Alexandru Bobe	Jindrich Pohl	Mert Gokbulut
Shading[0.5]	0%	100%	0%
Recursive ray-tracer[1.5]	0%	0%	100%
Hard shadows[1.0]	0%	100%	0%
Soft shadows[1.5]	100%	0%	0%
Acceleration data-structure[6.0]	0%	55%	45%
Report[1.5]	33.3%	33.3%	33.3%
Interpolated values[0.5]	100%	0%	0%
Motion blur[1.2]	100%	0%	0%
Bloom filter[1.0]	100%	0%	0%
Anti-aliasing[0.5]	0%	0%	100%

Table 2: Performance test table

Dummy	Cor. box	Cor. box sph. light	Monkey	Dragon	Teacup	Donut
Num triangles	32	32	968	87K	48K	320K
Time	168ms	48.5s	0.5s	0.5s	0.43s	6.5s
Bvh levels	8	8	11	12	12	12

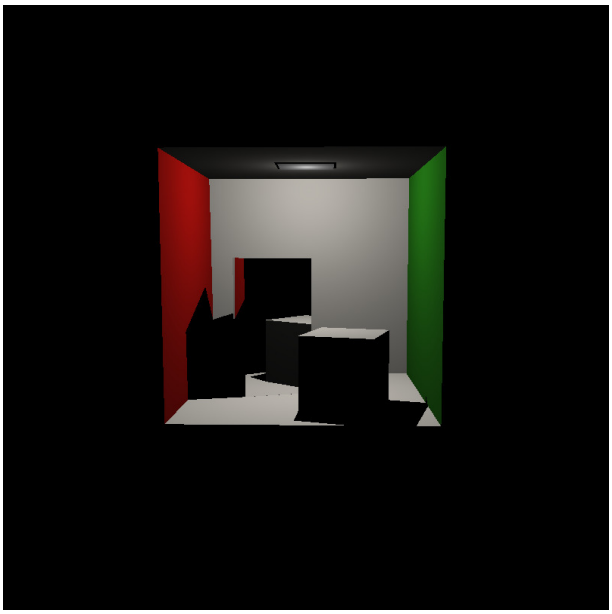

Figure 28: Performance render of the cornell box with a point light

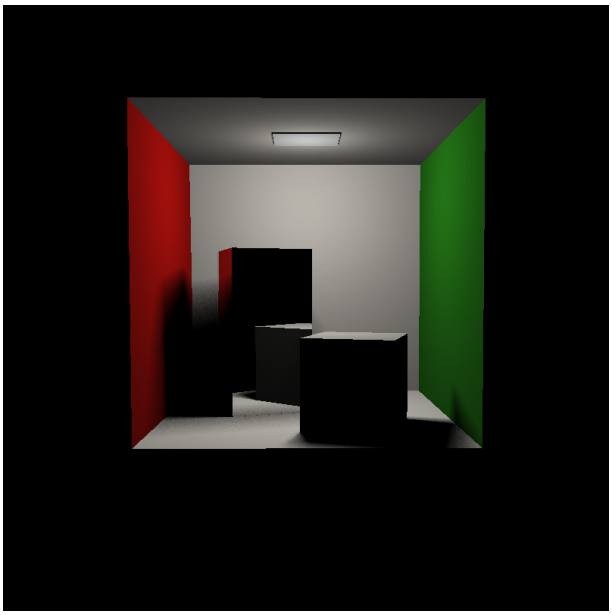

Figure 29: Performance render of the cornell box with a spherical light

Figure 30: Performance render of the monkey

Figure 31: Performance render of the dragon

Figure 32: Performance render of Alex's teacup

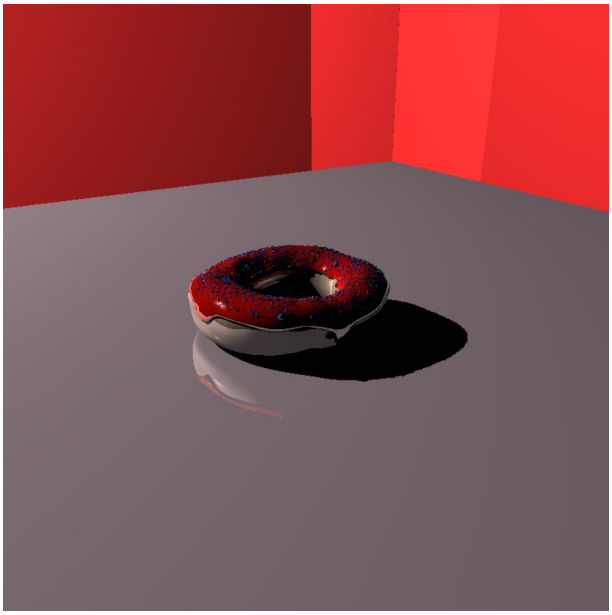

Figure 33: Performance render of Mert's donut

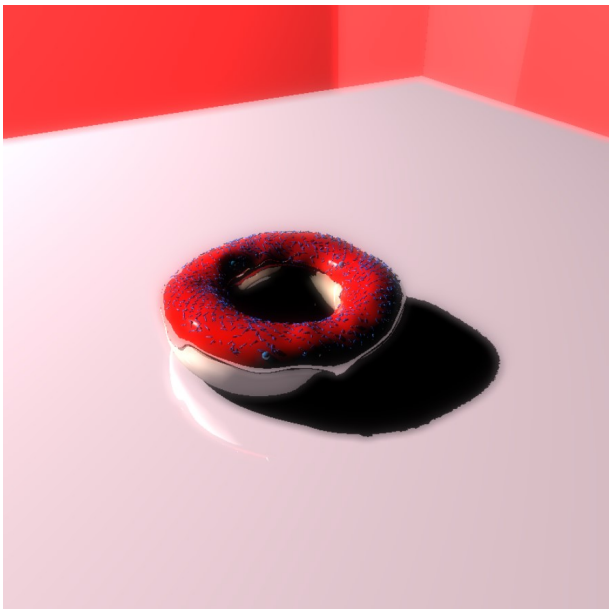

Figure 34: The render for the competition