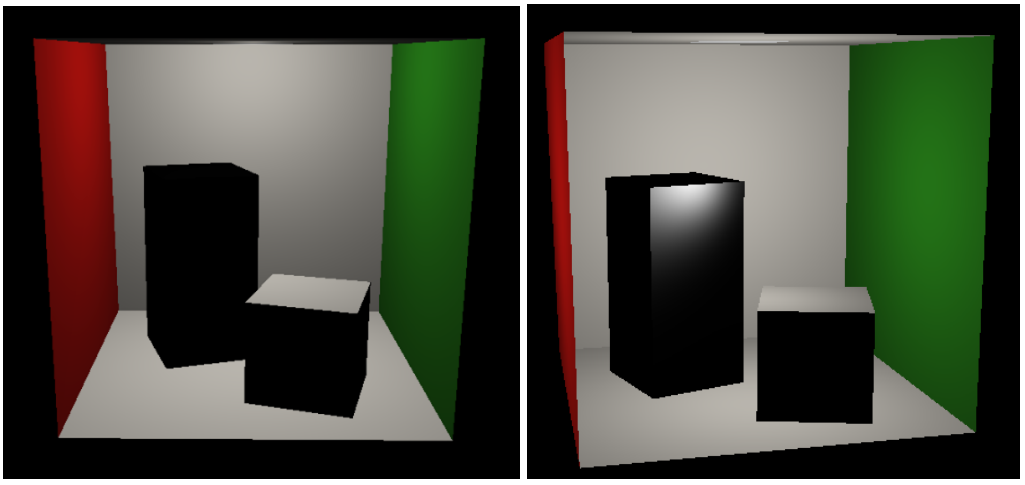
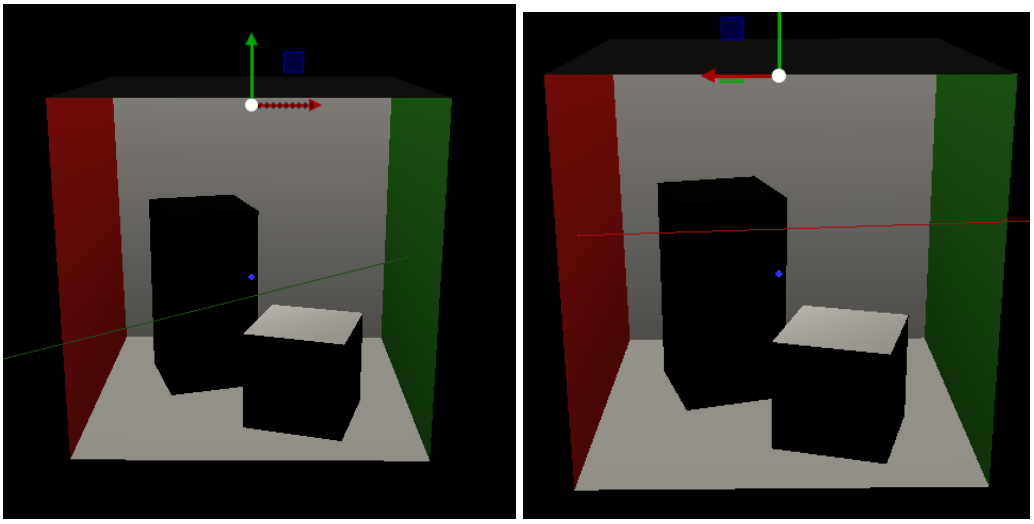
CG Project Report
CG-26

- ## 3.1 Shading - Jeroen Kok

Shading works very simply by calculating phong shading for each light in the scene. All the contributions of each light are added together for each point and then returned as the final color for that pixel. The visual debug shows the final calculated color as the color of the ray itself

Pictures of feature in ray-tracer mode:



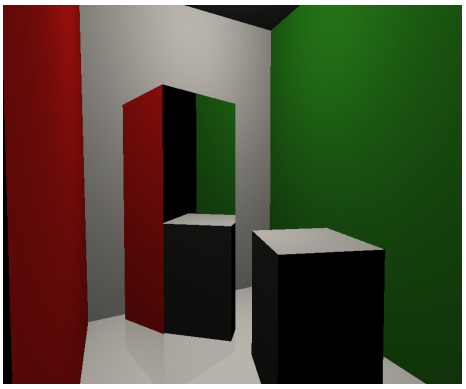


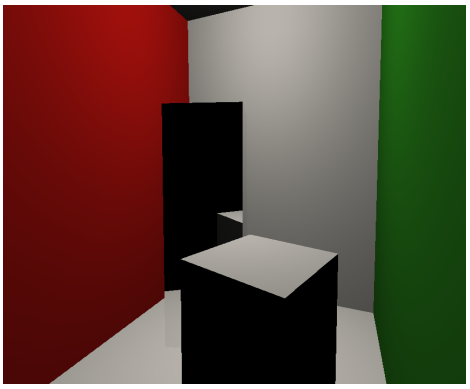
- **3.2 Recursive Ray-Tracer - Ivan Virovski, Erfan Mozafari**

This feature uses the calculations from the previous feature [Shading] by calling the computeLightContribution method recursively to compute the final color and the method computeReflectionRay is also called recursively in getFinalColor to return the reflected ray from the reflective surface if it is hit by a ray. Given an incoming ray and the surface that has been hit, the computeReflectionRay method calculates the origin for the new array and adds a small value to it to avoid floating point problems that can lead to the origin being behind a triangle. The direction of the reflected ray is calculated using the specular reflected ray formula and the distance of the ray is set to infinity. After the new reflected ray has been computed we add the specular color to the final color of the pixel. Then the recursion continues until we have reached 5 reflected rays or the current ray of the recursion has hit a non reflective surface or nothing.

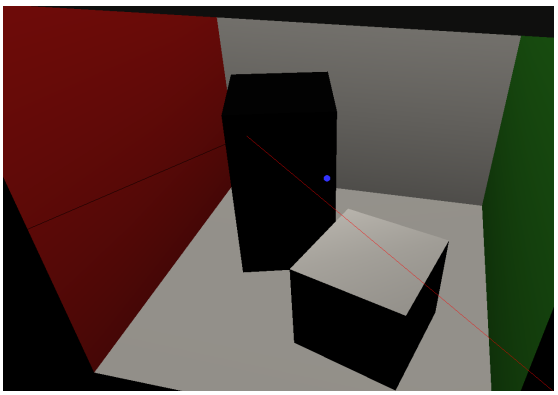
Debug works by drawing a ray everytime the recursion is called and the final color of all the rays is the final computed color of the pixel, given all the recursions and intersected surfaces.

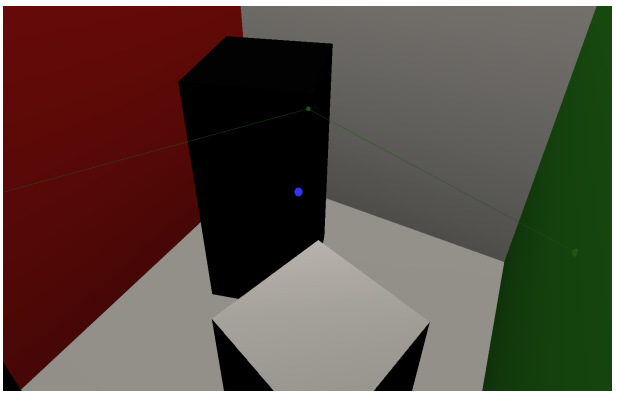
Pictures of feature in ray-tracer mode:





Pictures of feature debug:

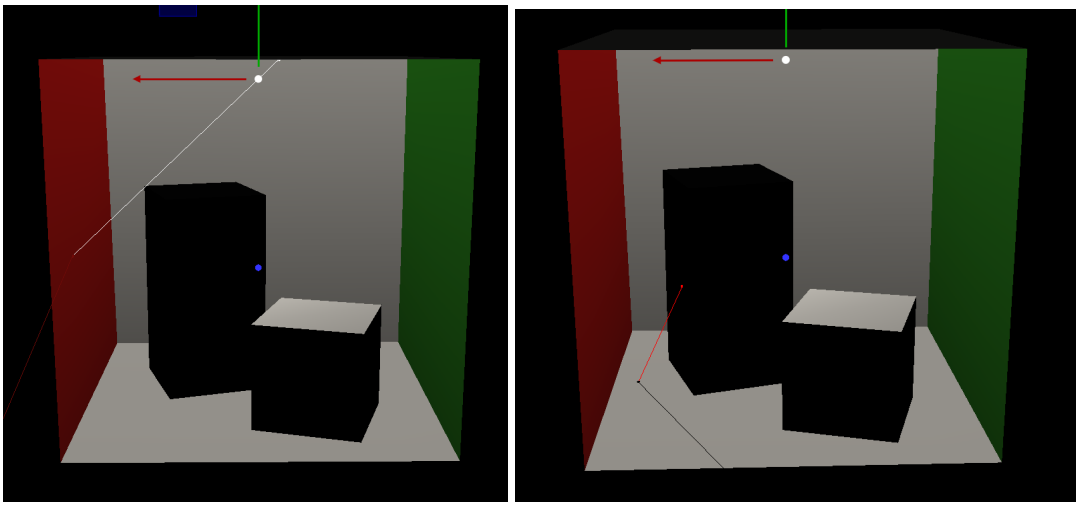




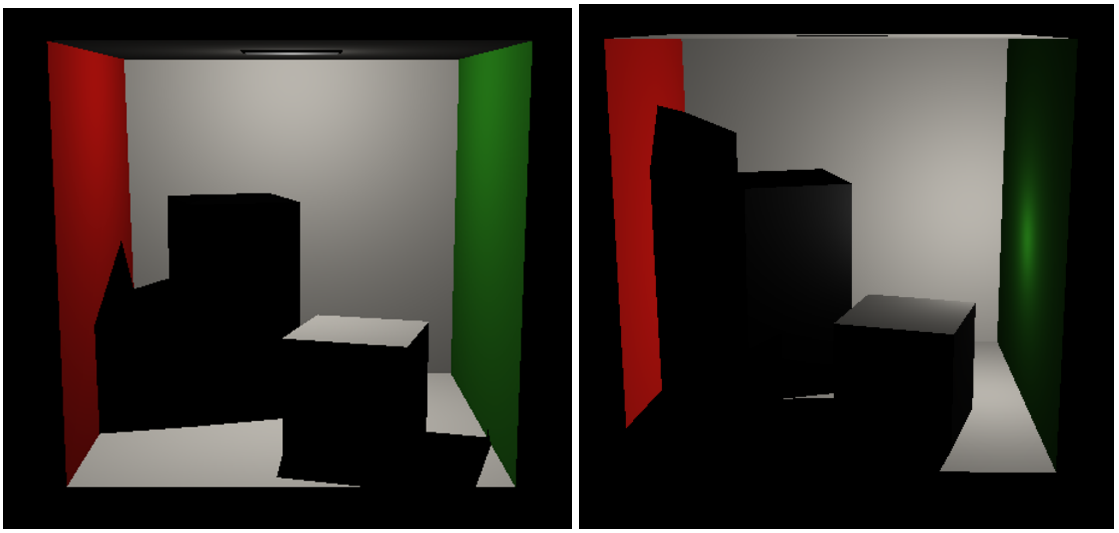
- ## 3.3 Hard shadows - Jeroen Kok

Hard shadows are calculated by casting a shadow ray at the intersection point towards the light source(s) and checking if that shadow ray intersects with the scene. If this shadow ray intersects with the scene, that would indicate that there is an object between the light and the intersection point, thus we return a black color. If the shadow ray reaches the light without intersecting with anything, the normal calculation can be done for computing the resulting color. The visual debug below shows 2 rays being drawn, one is the ray from the camera and the other is the shadow ray. If the shadow ray intersects with the scene the shadow ray is drawn in red and if the path is clear to the light, it is drawn in white.

Pictures of the hard shadows visual debug:



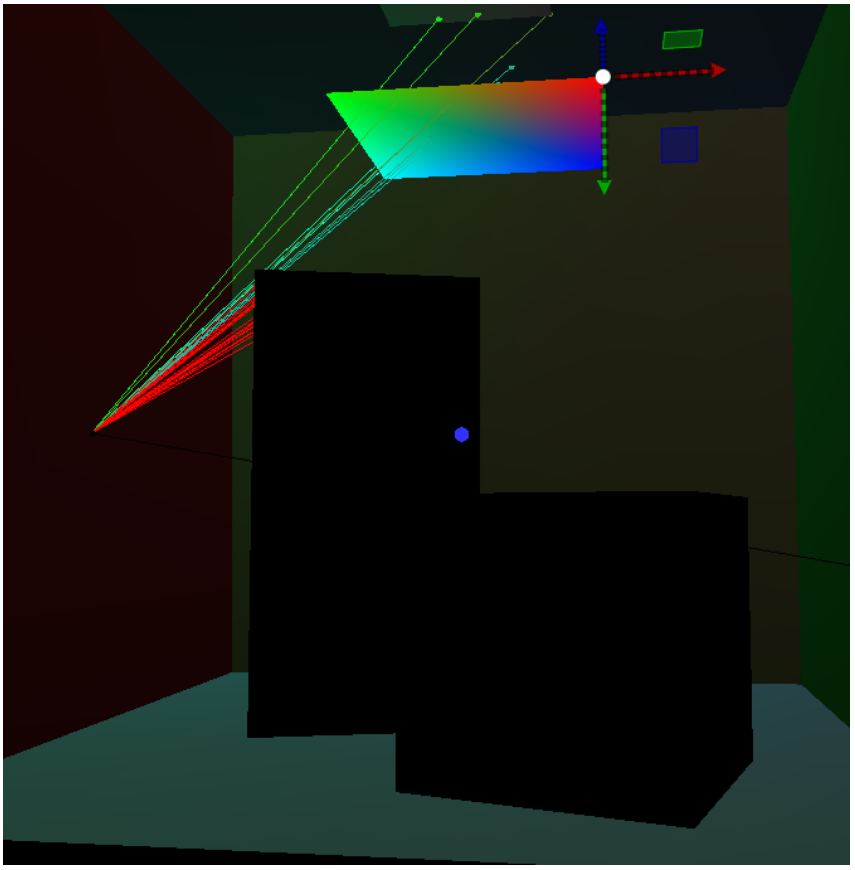
Pictures of the hard shadows visual debug:

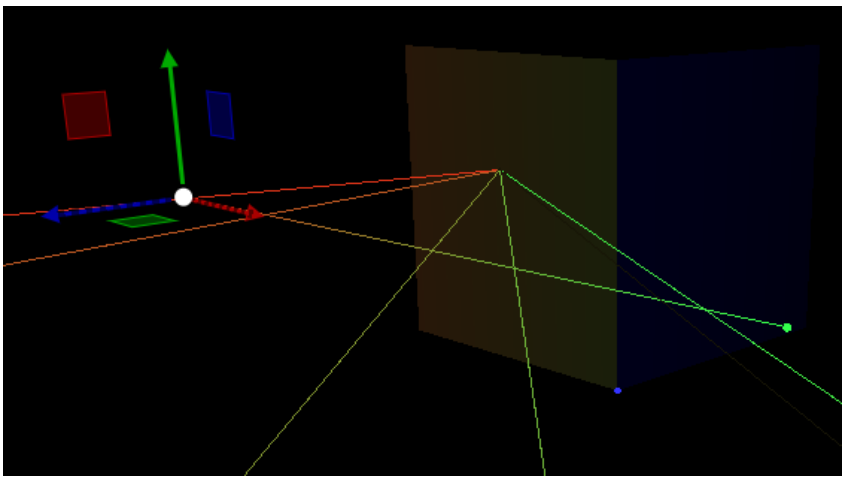


- ## 3.4 Area Lights - Erfan Mozafari

By following the standard linear and bilinear formula for interpolating colors, the shader computes the color according to the position at which the sampled light rays hit the scene.

Five Sampled points are chosen for segment lights as that is mostly enough to capture the nature of the whole segment and 30 are chosen for the parallelogram light as to cover at least 5^2 points. Each sampled ray is shadow tested and if not occluded, the result is calculated using the linear or bilinear formula for segment and parallelogram light respectively. Examples of Visual Debug is shown below.





- **3.5.1 Acceleration Data-Structure Generation - Ivan Virovski, Jeroen Kok**

Structure:

There is only one additional parameter to the BoundingVolumeHierarchy class - listOfNodes. This parameter is a vector containing all the Nodes of the constructed BVH of the scene. This parameter is used to easily traverse all nodes without having to go into the nodes themself and receive their children. It is used for example in debugDrawLeaf to get all nodes that are leafs.

Two new structures have been added. Node and Triangle BVH. The Node structure represents every node constructed by the BVH. It has a lower and upper Bound which are used for the construction, visualization and splitting of the nodes themselves. A boolean if the node is interior or a leaf, it is used only in debugDrawLeaf and the Traversal Feature. A vector named subs containing the indexes of the children of the node. If the node is internal it contains 2 indexes -> the indexes of its children in listOfNodes. If it's a leaf it contains 4 indexes for each triangle in the node. 1 for the mesh index and 3 for the triangle coordinates indices which it takes from the triangleMesh parameter of TriangleBVH, which is described below. The last parameter of the Node struct is its level stored in an integer. This parameter was added for easier implementation for debugDrawLevel. The TriangleBVH struct contains 3 parameters. Its mesh index, a vector representing the indices of the 3 vertices of the triangle, which are stored in a vertice array and a vector of the triangle centroid coordinates.

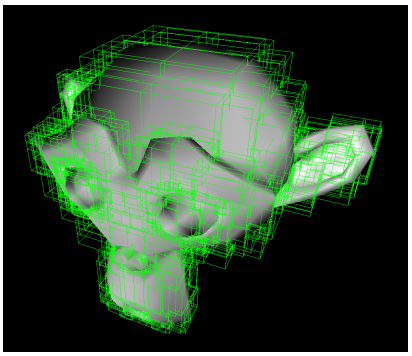
Implementation:

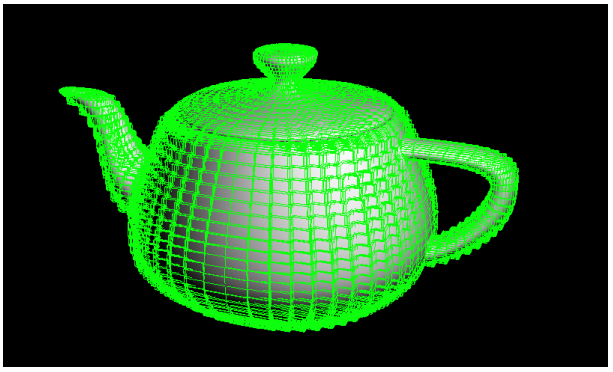
First all the triangles of the scene are transformed into a vector of TriangleBVH struct so that we can use the extended parameters of this structure. Then the root node is created by putting all the newly created TriangleBVH structures as its children and adding this root to the vector of all nodes.

Then we check if the BVH generation should be SAH or Standard. If the latter we go into the recursionBuilder method. This method recursively receives the given Node from the listOfNodes, starting the recursion at the root. Checks if the node can be split or if the BVH level is too high, base case. If so it updates the number of leaves and levels and makes the current node a leaf node. If not, the current node is set as an interior node. We sort all the triangles that have been given to the method, based on their centroids coordinate based on the level

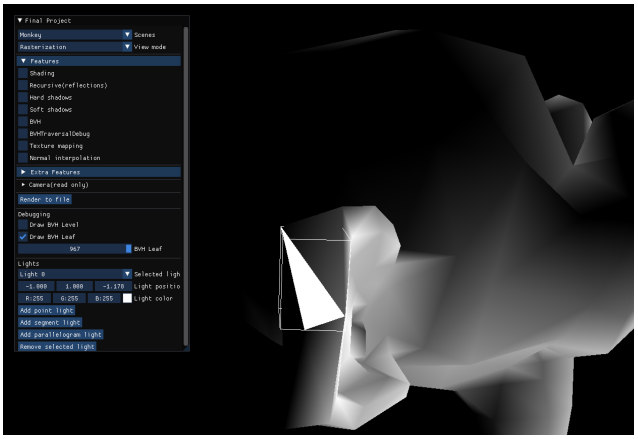
that has been given. After that we split all the sorted triangles into 2 groups in the middle. After that we create the 2 new child nodes with these 2 groups of triangles as their children, update their levels, and set them as leaves. We replace the current node children to these 2 new nodes and continue the recursion by calling the method with both of the new children indexes in listOfNodes, and updated level and the triangle children of the according node. The performance of this feature is outputted on the terminal with time taken in ms and number of triangles.

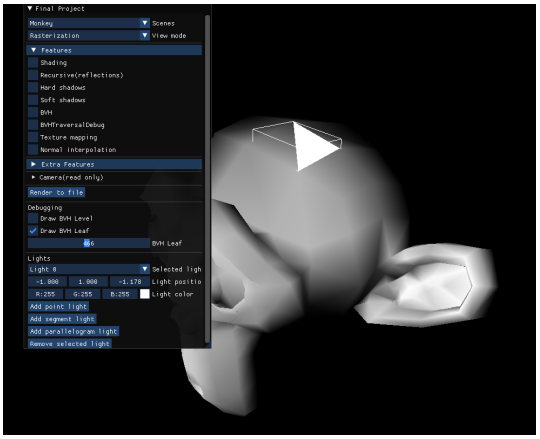
Pictures of level Debug:





Pictures of leaf Debug:





Performance Table (in Release mode):

	Cornell Box	Monkey	Dragon
Number of triangles	32	967	87130
Time to create in ms	0	4 - 20 ms	788 - 1061 ms
Time to render in ms + Shading, without Traversal	1200 ms	32478 ms	too long
BVH Levels	5	10	17
Max triangles per leaf Node	1	1	1

- **3.5.2 Acceleration Data-Structure Traversal - Ivan Virovski**

This feature makes use of the BVH generation to speed up the render by recursively going through nodes until it hits a triangle in a leaf node. This feature is implemented in the intersectRecursion method which is called from the intersect method.

This method starts by first creating an AxisAxignedBox from the current node and checking if there is an intersection with the ray. If there is no intersection it means the ray misses the box and the method stops. If there is an intersection then it checks if the node is a leaf or an interior node.

If it is a leaf it creates a triangle from the children of the leaf node. Then it checks if this triangle intersects the ray. If it does it updates the hit Information and returns true. If does not hit the method returns false

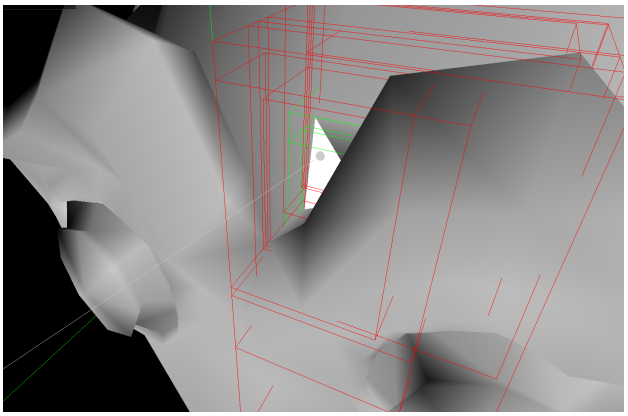
If it is an internal node, it means that it has only two children in the subs parameter -> the indexes of the children Nodes. Using these 2 children Nodes we determine which one is closer to the ray, meaning which one will be intersected first. If the first child is closer than the second we call the method recursively with the first child and then again with the second child and vice versa. This way we iterate through both children in case the closer child doesn't have a triangle that intersects the ray.

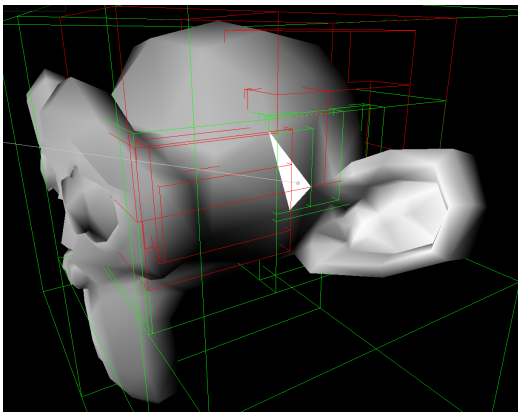
Debugs works by coloring all the intersected boxes that have a child with intersected Triangle green. Colors all the intersect boxes that don't have a chi;d with the intersected Triangle red. If there is no intersected triangle the ray is colored red.

Performance Table (in Release mode):

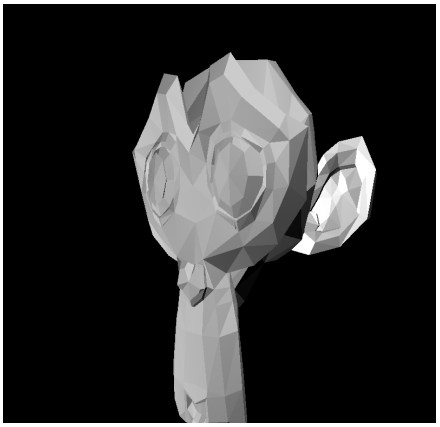
	Cornell Box	Monkey	Dragon
Time to render in ms + Shading + Traversal	1200 ms	2000 ms	2800-2200 ms

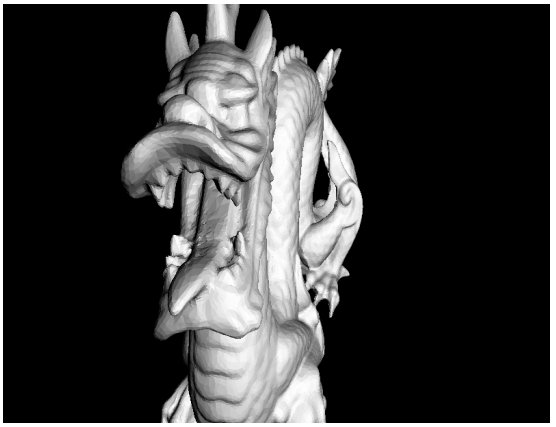
Pictures of Debug:





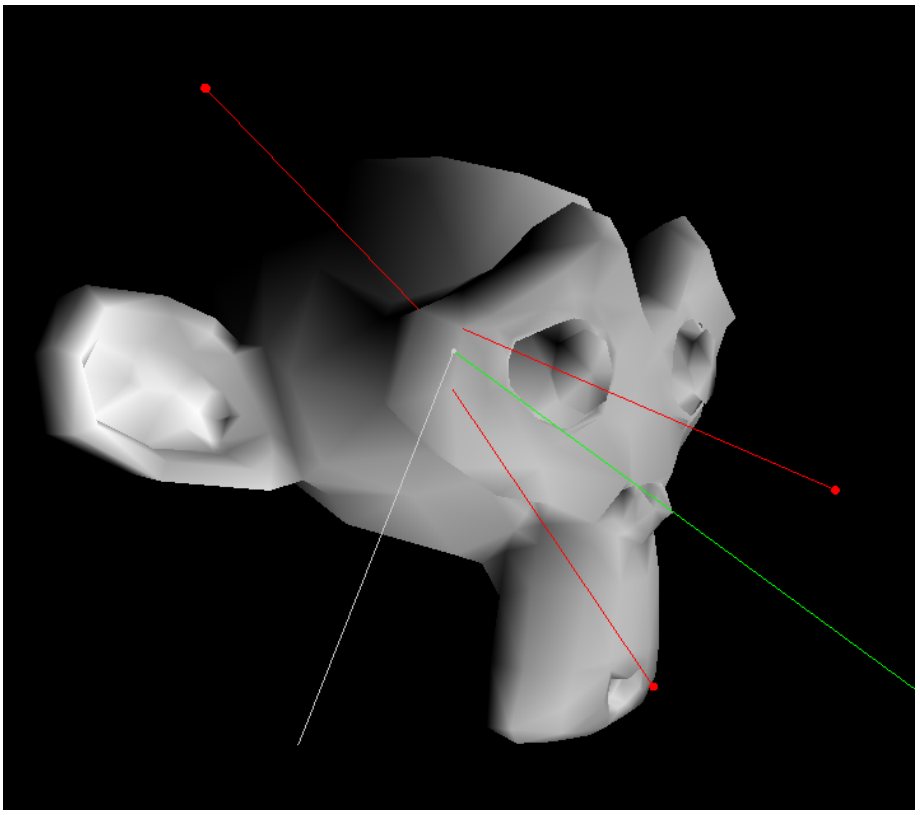
Pictures in ray-tracer mode:





- **3.6 Barycentric Coordinates For Normal Interpolation - Erfan Mozafari**

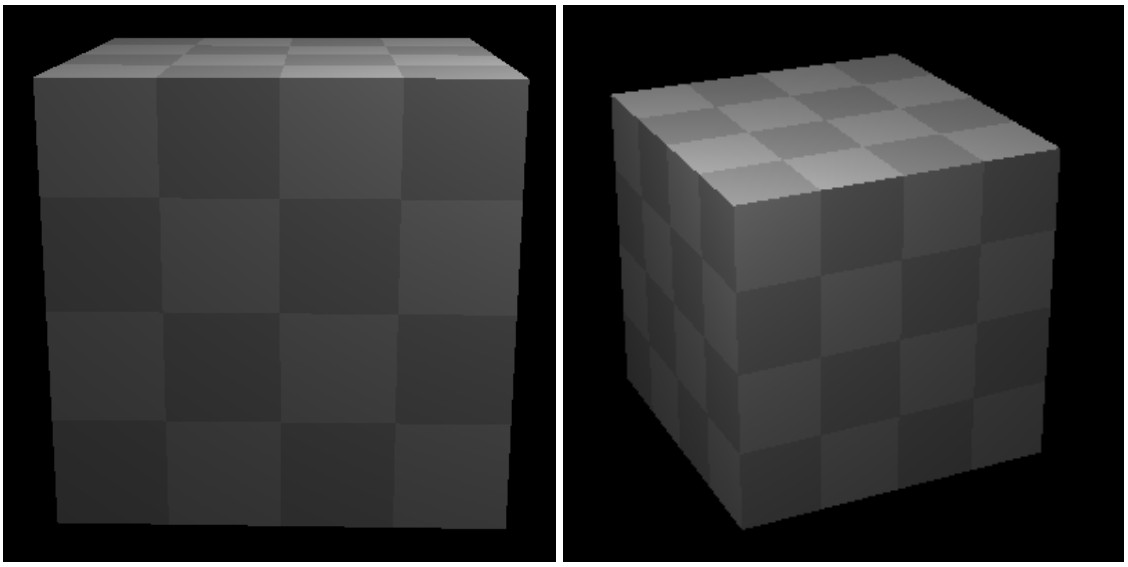
To smoothen the computations, we calculate the normal at the middle of a triangle using the normal at each respective vertex. Using the cross product to calculate areas, the area ratio of the three newly formed triangles against the mother triangle decides the respective weight for the final normal. Below is an example of the visual debug. As observed, each normal at each vertex is drawn in red while the interpolated normal is drawn in green.



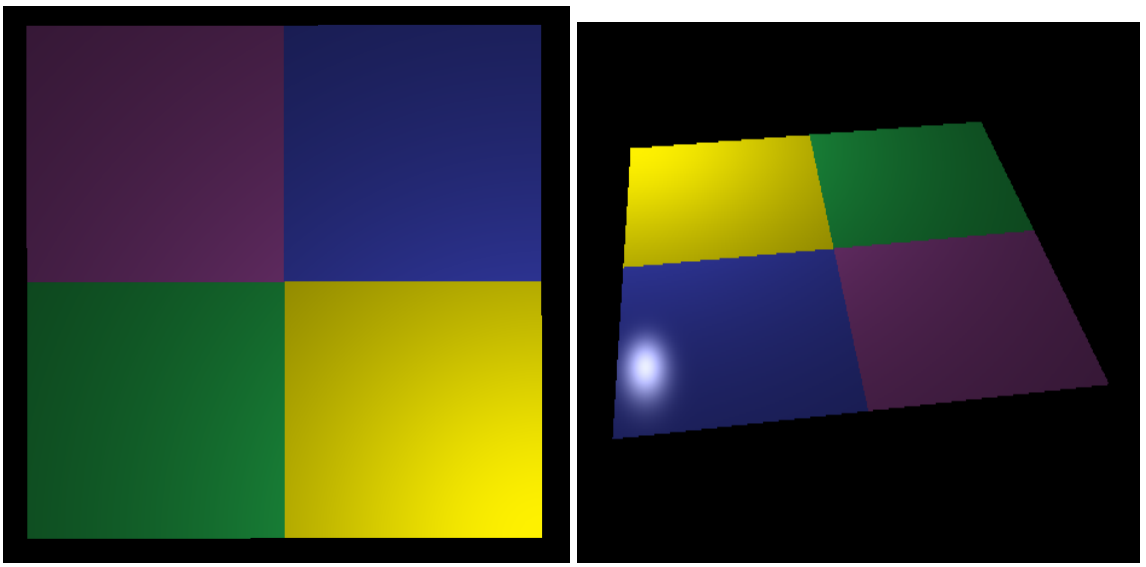
- ## 3.7 Texture mapping - Jeroen Kok

Texture mapping is done by first calculating the barycentric coordinates for the given point in the intersected triangle. This can then be used to interpolate the texture coordinates of each vertex in that triangle to get the texture coordinates of our intersection point. These texture coordinates are then passed along with acquireTexel(Texture.cpp) and this function will calculate the corresponding texel. Then acquireTexel will then return the color of that texel, which replaces the kd of the intersected object, thus creating a texture on that object. The visual debug is a quad located in the custom scene, which will draw a 2x2 texture. The result can be seen in the pictures below.

Pictures of the default textured cube:



Pictures of the visual debug:



- **4.1 Environment maps - Jeroen Kok**

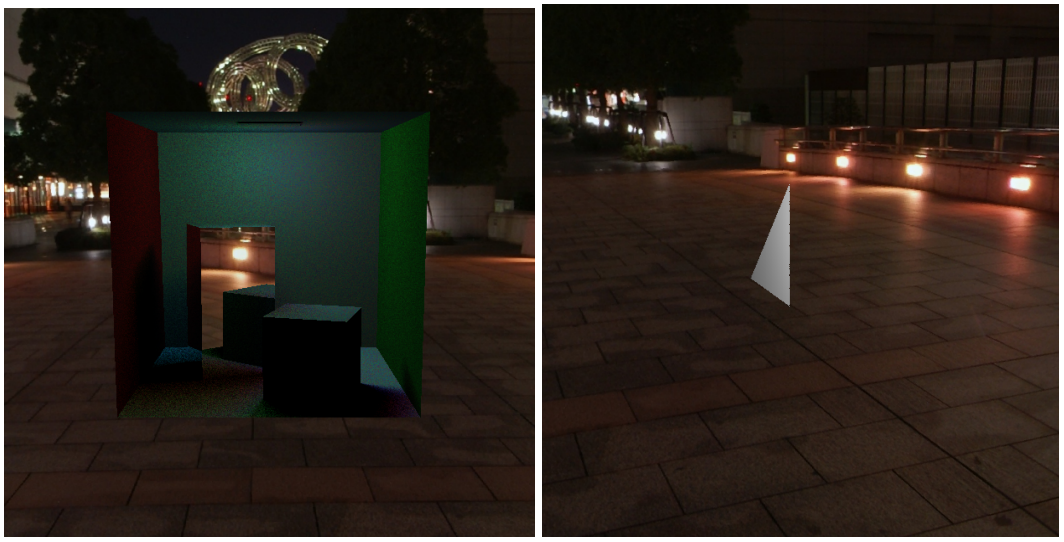
Code can be found in:
Render.cpp line 14-38, 67-91

Environment maps are implemented as a way to add a skybox to the scene. You can find this implementation within getFinalColor in the else statement. Environment maps are implemented using a cubemap that is split into 6 pieces. For each axis one positive and one negative. For each ray that misses the scene or for a ray that gets reflected into missing the scene, the ray.direction is used to determine which axis it points to. After that we need to transform our coordinates, since texture coordinates go from 0 to 1 and the normalized direction of a ray can go from -1 to 1. This is calculated in getUV in render.cpp. Those coordinates along with the corresponding face of the cubemap are passed along to acquireTexel in texture.cpp to find the color of the given texture coordinates.

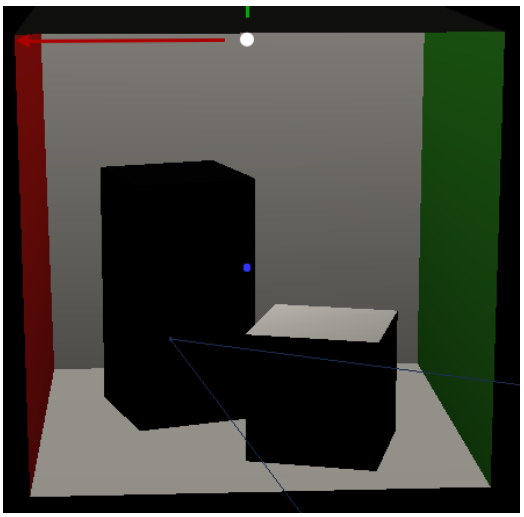
The visual debug has been implemented by setting the color of a ray that misses the scene as the color found from the environment map functions. Besides that the custom scene has a quad with a reflective surface to test the mapping on that.

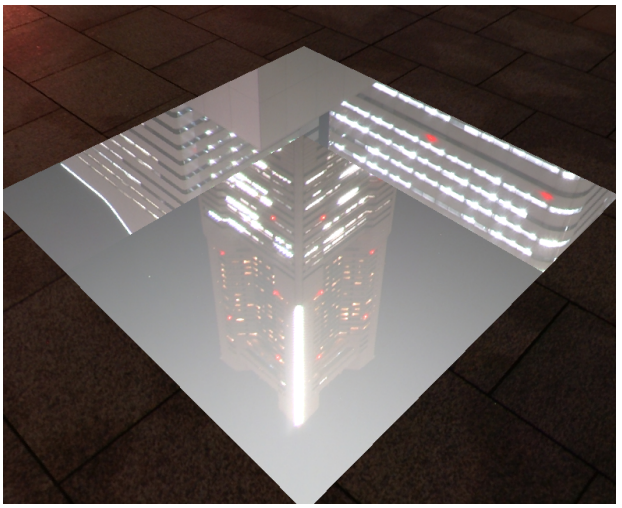
The skybox used can be found within data/enviromap and was taken from:
https://www.humus.name/index.php?page=Textures&start=0

Pictures of the environment map in use:



Pictures of the environment map visual debug:





- **4.2 SAH + Binning as splitting criterion for BVH - Ivan Virovski**

Sources: Lecture 9 slides and TAs
Location: bounding_volume_hierarchy.cpp; recursionSAHBuilder()

This feature builds upon BVH generation by improving the splitting of the children Nodes. It optimizes the child nodes of the current node by splitting them in a way that results in better probability of hitting a triangle inside a child node. This method is similar to BVH except for the addition of calculating the axis and triangle with which we split the nodes. They decide this by sorting all the triangles and iterating through them and computing the cost. We do this for all triangles and for every of the 3 axes. The cost of split is calculated by taking the percentage of the first child volume to the parent node volume, multiplying it by the number of triangles in the Node after the split and adding it with the same operations for the second child. The split with the minimal cost is taken as the best split and then we split the nodes based on it and the axis and from here it is the same operations as the generic BVH.

Debug: Because we don't split in the center every time like in normal BVH this can lead to unbalanced trees. There are more levels in BVH SAH than standard BVH, but the number of leaves stays the same.

Performance Table (in Release mode):

	Cornell Box	Monkey	Dragon
Number of triangles	32	967	87130
Time to create in ms	19 ms	7438 ms	tl
Time to render in ms + Shading + Traversal	30775 ms	32478 ms	too long
BVH Levels	5	17	17
Max triangles per leaf Node	1	1	1

- ## 4.4 Bloom Filter - Erfan Mozafari

The code for a bloom filter has been in implemented in render.cpp. First it thresholds the image then after generating a gaussian kernel it applies the kernel to the bright points. After that the results are combined.

We used the vector { 0.2125, 0.7154, 0.0721 } as our luminance gauge to dot the screen pixels with as this vector approximates the contrast differentiator mechanism of the human eye with respect to RGB colors.

We are sorry to inform that due to poor planning we could not offer a visual debug for this feature.

- ## 4.7 Multiple Irregular Rays - Erfan Mozafari - Ivan Virovski

The code for multiple irregular rays per pixel has also been implemented in render.cpp. It uses a structured but also jitterized newly generated rays per pixel to average the color of the final pixel. Similar to the previous feature this one also lacks a visual debug.

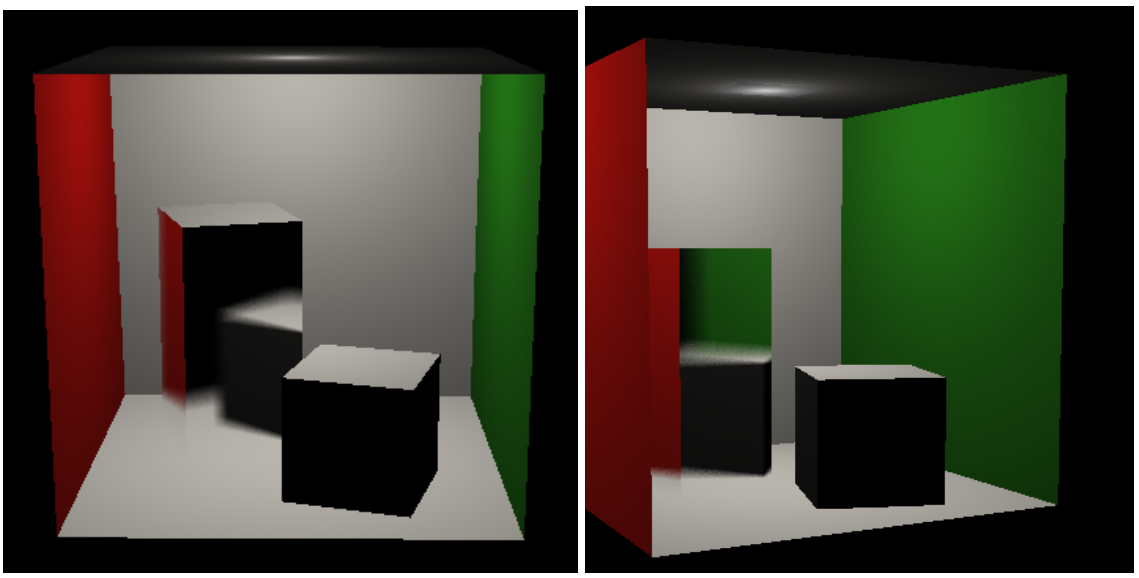
- **4.8 Glossy reflections - Jeroen Kok**

Code can be found in:
Shading.cpp line 9-10,45-79
Render.cpp line 47-54

Glossy reflections are implemented by creating a sample set of reflection rays that have been slightly perturbed to simulate an imperfect mirror surface. This has been done using formulas as described in 2.4.6 from Fundamentals of computer graphics. This creates an orthonormal basis around the intersection point with which we can create the new slightly offset rays. The basis ensures that the new rays that are created are only slightly offset and still point in the general right direction. Then together with the formulas described in 13.4.4 from Fundamentals of computer graphics, we slightly alter the direction of our reflection ray. Each time this is run, the amount by which our calculated orthonormal vectors are multiplied is randomly selected by a uniform distribution that generates a value between 0 and 1. The uniform distribution used comes from the random class in c++ and the generator used is a Mersenne Twister generator with a rand() seed. This further helps to make the generation of these small multipliers actually random, which will help to prevent patterns from showing up. Within the calculation a multiplier is used to increase hitInfo.material.shininess(shading.cpp line 72 and 73), I chose to do this because the variable could be quite small and it would result in the glossy reflections not showing up well. With the new and slightly perturbed ray we can call getFinalColor with that ray, to find the color on the intersection. This is done over x rays and each one is added together. At the end the result is divided by x and then returned as the final color.

The visual debug shows all the perturbed rays and each of them show what color they are adding to the whole

Pictures of the glossy reflections in use:



Pictures of the glossy reflections visual debug: