Group 26

Computer Graphics Final Assignment

Group 26

Egemen Yildiz - 5716934

Christiaan Baraya - 5752132

Ewout de Dobbelaar - 5746957

Deliverable - Report

Work distribution

FILL IN THE BLUE BOXES ONLY

Fill in the percentages with integer values in range [0-100]

		member 1 5752132 Christiaan Baraya	member 2 5746957 Ewout de Dobbelaar	member 3 5716934 Egemen Yildiz	
basic features	**points**				total
generation and traversal of acceleration data-structure	6,00	0	100	0	100
implementation of shading models	1,00	100	0	0	100
implementation of recursive ray reflections	1,00	100	0	0	100
implementation of recursive ray transparency	1,00	0	0	100	100
normal interpolation with barycentric coordinates	0,75	100	0	0	100
implementation of texture mapping	1,50	100	0	0	100
implementation of lights and shadows	5,00	0	0	100	100
implementation of multisampling	1,75	100	0	0	100
total	**18,00**	**6**	**6**	**6**	
extra features	**points**				total
Environment maps	1,50	100	0	0	100
SAH+binning as splitting criterion for BVH	2,00	50	50	0	100
Motion blur	4,50	0	0	0	100
Bloom filter	1,50	0	0	100	100
Glossy reflections	1,00	100	0	0	100
Depth of field	3,50	0	50	50	100
total	**14,00**	**3,5**	**2,75**	**3,25**	
total (basic + extra)		**9,5**	**8,75**	**9,25**	

Standard features

BVH Performance test

	Cornell Box (with mirror)	**Monkey**	**Dragon**
Num triangles	32	967	87,130
Using regular BVH			
Time to create	1.38 ms	1.44 ms	70.13 ms
Time to render	53.4 ms	103.4 ms	185.5 ms
BVH levels	3	8	15
BVH leaves	8	256	32,768
Using SAH-binning			
Time to create	0.53 ms	3.48 ms	260.85 ms
Time to render	46.0 ms	58.2 ms	106.4 ms
BVH levels	8	25	42

BVH leaves	10	399	34,306

Release mode is used and no other windows are open outside of this document and vscode. The camera does not move from the starting position, and the average render time from one experiment over 100 frames is taken, rounded to a tenth of a milliseconds. The average construction time from one experiment over 10 builds is taken, rounded to a hundredth of a millisecond.

Light transparency sampling

Light transparency in our ray tracing model is a good representation of the real world but not as accurate as it can be. First of all, due to our approach towards light and shadows, and our intersection method, the light rays that meet/intersect a shadow disappear/get ignored afterward. This is not the case in real life since light that is on a shadow doesn't disappear like it's cut by a knife. Secondly, a light ray that hits and gets reflected from a surface doesn't go as a single light ray, it gets scattered into different amounts of rays. In our model, our reflection logic doesn't contain this real-life situation. Finally, the concept of refraction, especially when the rays go through a slightly transparent object is ignored in our project.

Extra features

Features descriptions

1. **Description.** A description of the implementation of about *50* words. Do not copy-paste code, but describe the feature and explain your approach. Cite external resources in case you used any. Your description should match what your code does!
2. **Rendered image (at least 2).** *Two* images that properly illustrate this feature. From the images, it has to be clearly identifiable. If the feature appears completely broken from the image, we may skip it during grading.
3. **Debug image (at least 2 if required).** *Two* images or sequences of images to illustrate the use of the visual debug feature. This can be a screenshot from the OpenGL window. For example, for the data-structure, you can show the different levels with multiple images and the intersected nodes of a given ray. It must be clear how the visual debugger helps to certify the correctness of your implementation.
4. **Location.** You should indicate which method(s) in which file(s) implement this feature, even if you have not deviated from the shell methods provided by the framework as described in Appendix C.

If any elements are missing, you may not receive all points for a feature, even if the implementation is correct.

Environment map

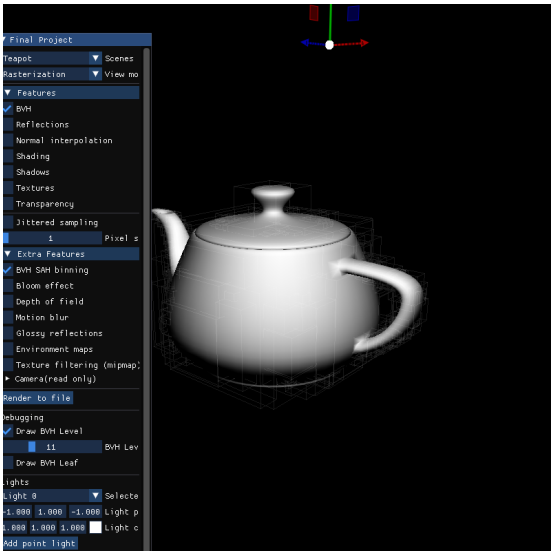
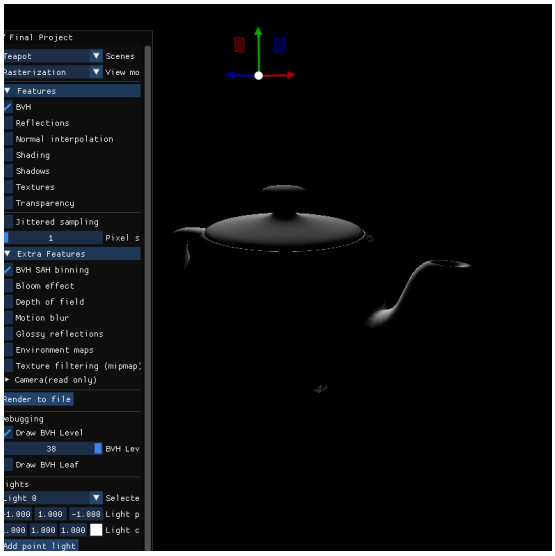
1. The environment map uses a cubemap of six different files. The method described in 11.2.1 from the book is used to convert the ray direction into texture coordinates. All the faces of the cube are stored as Image instances in a std::vector. Depending on the sign and the biggest absolute value from x, y and z, the faces to apply the texture coordinates on is chosen. This uses sampleTextureNearest() from the standard feature texturing.
2.

	
Example of an environment map	Environment reflected in a mirror.

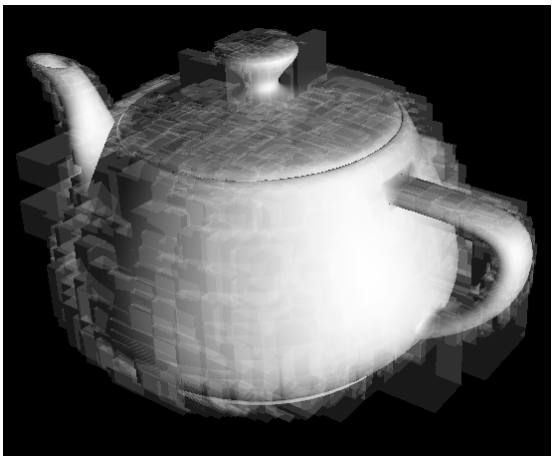
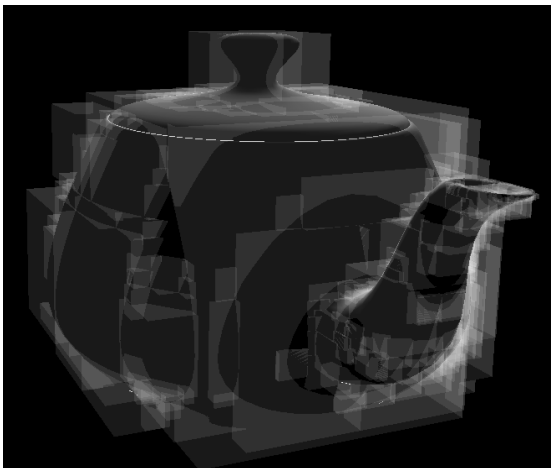
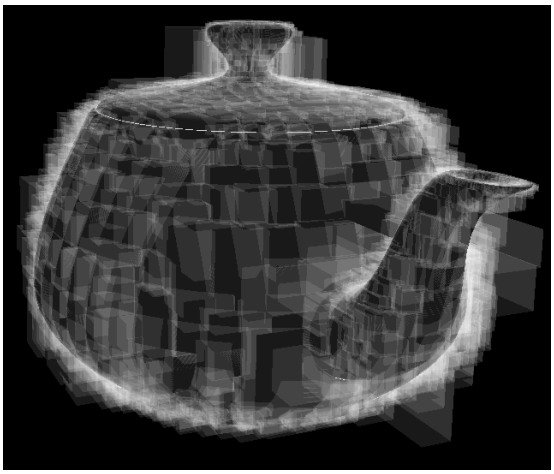
3. X
4. The method is in **extra.cpp** under **sampleEnvironmentMap**. It uses the method **sampleTextureNearest** from **texture.cpp**. In **scene.h** a std::vector with Images is added as an attribute to the struct. On **main.cpp**, below the ImGui::checkbox for the environment, the example files are loaded into the vector, these images can be replaced by changing the path. But images of a different cubemap should be added in the same order (-x, -y, -z, x, y, z).

SAH-Binning:

1. First The code sorts the array of primitives to be able to work with it. Then the code will test for 9 different splits with a for loop. Every time the value of the split gets updated and the primitives will get tested if they are before the split or after it. If the primitive is past the split the program will divide the primitive array up and calculate the area of both subspans. Then it will get compared to the lowest area yet, and if it is lower the index of the last primitive will get saved. If all splits are calculated the program will return the index of the split that had the least area.
2.

	
Somewhere in the middle of the BVH	At the end of the BVH

3.

	
Level 12 (the highest) in regular BVH	Level 12 using SAH-binning as well
	
Parts with more triangles, like the spout, have more nodes.	With normal BVH where the amount of nodes is almost the same everywhere.

4. The method is still in the same place. There are two helper methods, one for sorting and one for calculating area of a AABB, those two methods are directly above the main method.

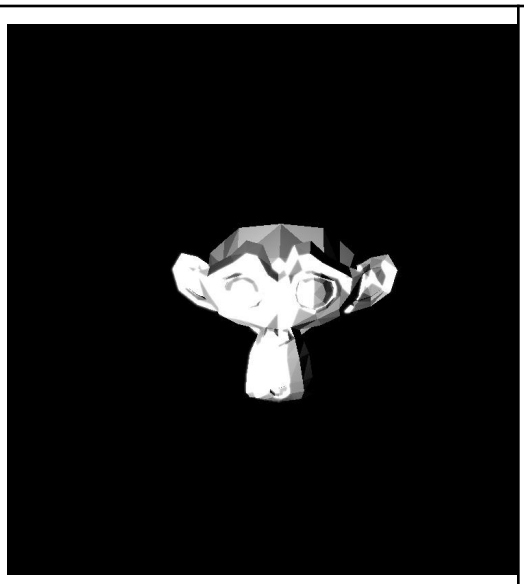
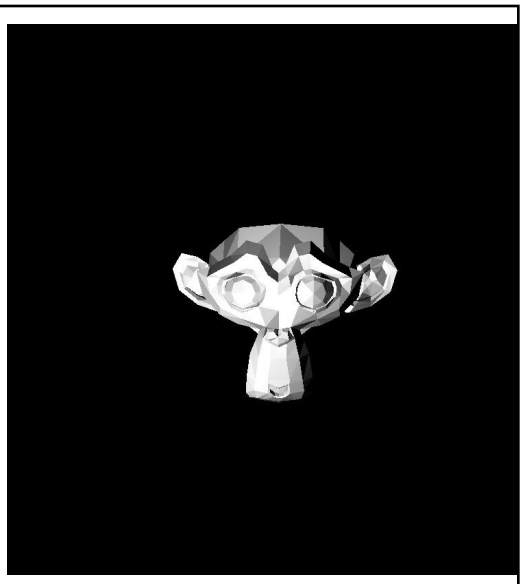
Bloom Filtering:

1. A bloom filter brightens the already “bright” (pixels with an RGB value higher than or equal to 0.8) parts of a picture. In the code, the postprocessImageWithBloom function applies a bloom effect to a rendered image. It checks if the bloom effect is enabled. Then

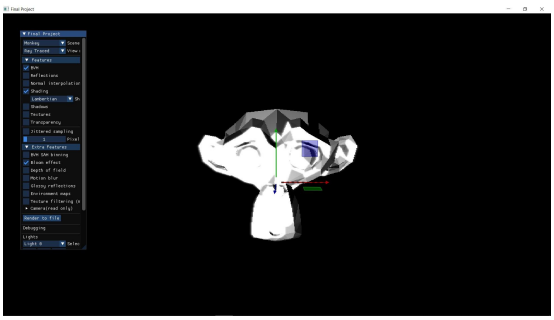
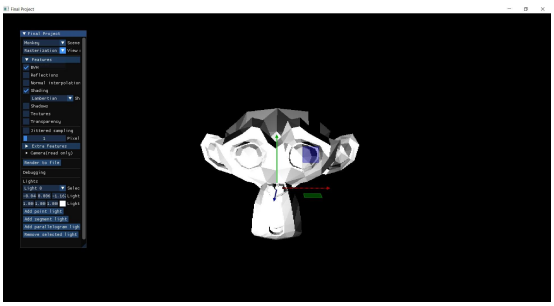
it identifies bright areas in the image and creates a bloomy version. Then, Gaussian-like blurs (through a filter that is normalized) are applied both vertically and horizontally, enhancing the brightness, and the results are added to the original image to achieve a bloom effect.

2.

	
With Bloom	Without Bloom

	
With Bloom	Without Bloom

3.

	
With Bloom	Without Bloom

4. Everything is in extra.cpp, under the name **postprocessImageWithBloom**. It also has a small factorial helper method called **fac** above it.

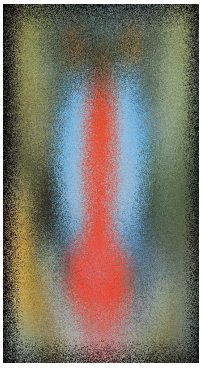
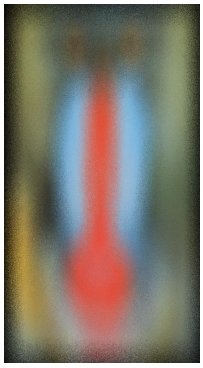
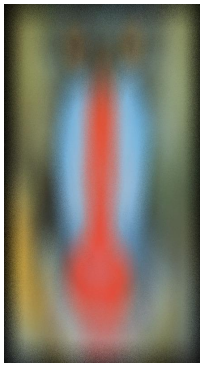
Glossy Reflections:

1. With a perfect mirror, all the rays get reflected perfectly. To get glossy reflections, a lot of samples are taken and the reflected rays are deviated with an offset sampled uniformly on a disk. The average of all these samples is taken to determine the color. Offset rays are taken in the following way:

   $\vec{r'} = \vec{w} + shininess/64 * \sqrt{\xi_2}(\vec{u}\sin(\xi_1 2\pi) + \vec{v}\cos(\xi_1 2\pi))$

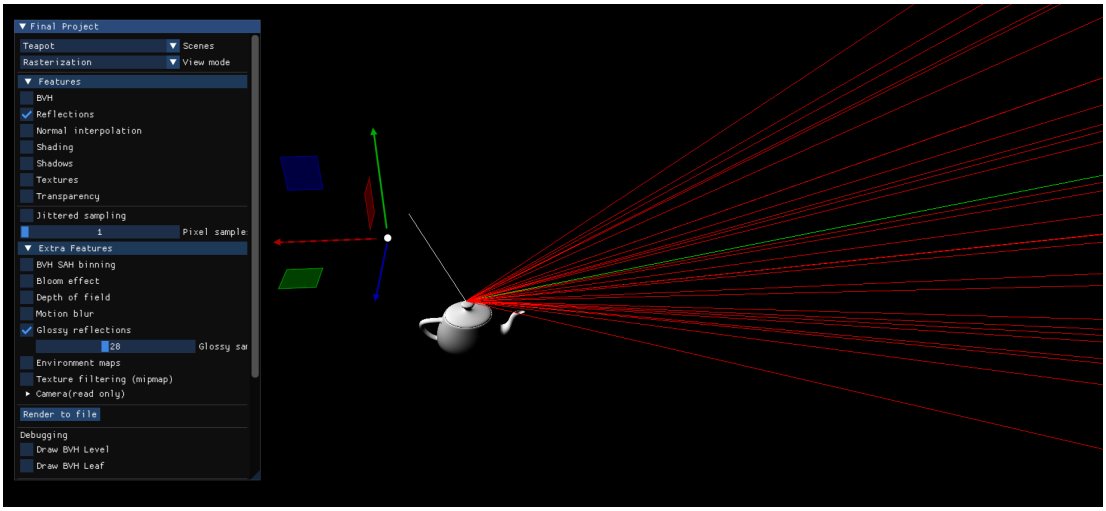
   Where $\vec{u}$ and $\vec{v}$ are a orthonormal basis to $\vec{w}$ which is the direction of the reflected ray.

2.

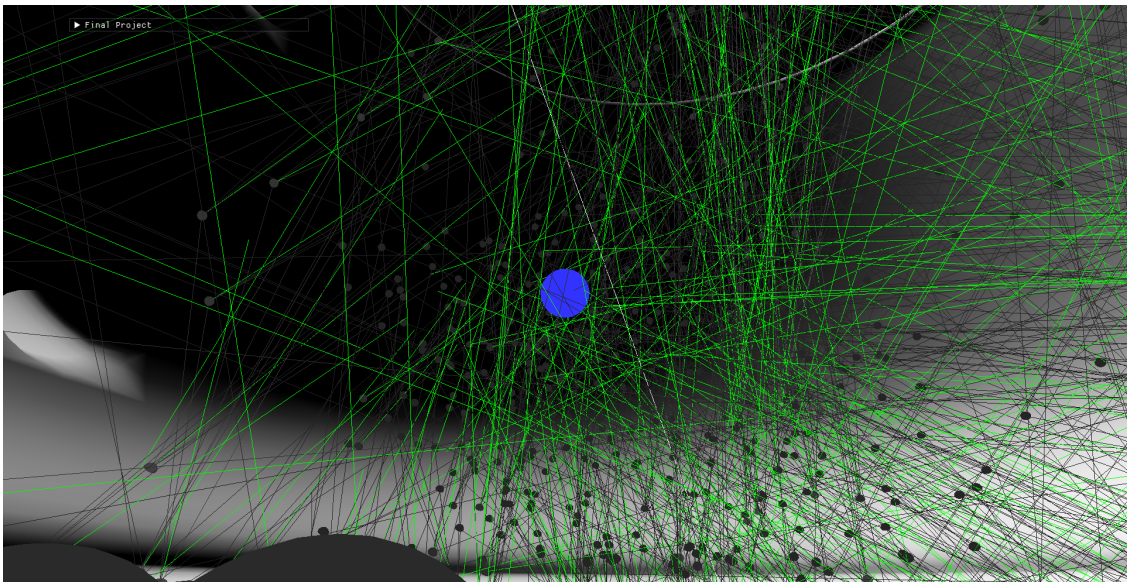
			
No glossy effect	Glossy effect of 1	Glossy effect of 16	Glossy effect of 64

These are renderings of a reflective surface that reflects a textured rectangle (see the custom scene).

3.



The green line is the reflected ray. The 20 red lines have a random offset from this green line.

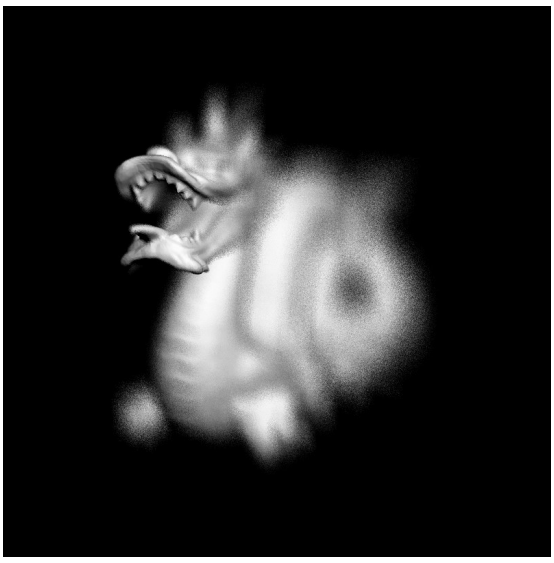


The inside of the teapot. Every time a gray ray hits a surface it splits into 3 more rays. This happens up to 6 times. For a total of 729 (3^6) rays and 243 green lines.

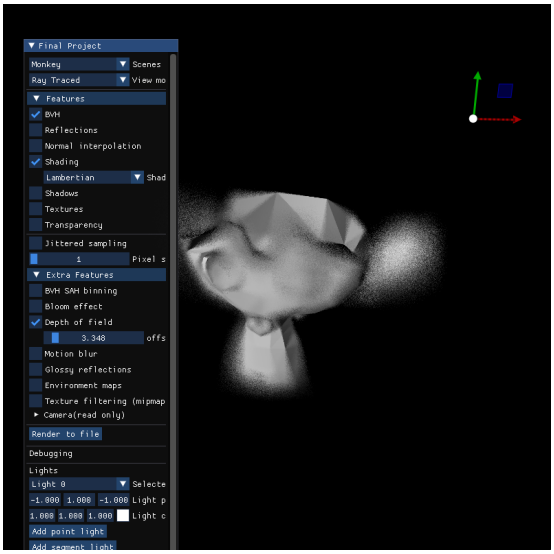
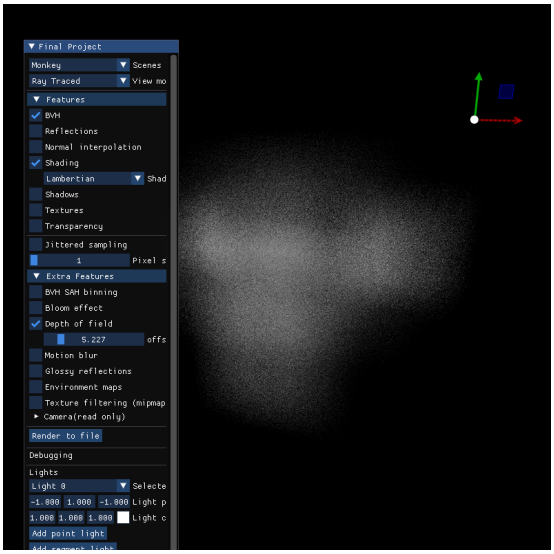
4. The method is in **extra.cpp** under **renderRayGlossyComponent**. It uses the methods **generateReflectionRay** and **renderRay** from **recursive.cpp**.

Depth of Fields:

1. Pre-determined sample and offset values are initialized. A loop is used to apply the depth of field effect for each pixel. For each pixel, RenderState object is created with scene, features, and BVH. The vector “color” stores the colors of samples for the current pixel. This is saved because of the chance that jittering samples is applied, in that case the avg of the jittering samples is first needed and then over all the different ray. We chose 20 samples to iterate over. Then, random offsets are generated that move the position and the angle of the which gives the depth of field effect. generatePixelRays creates rays from the camera with the effect included. For each ray, the origin and the direction is offset which gives the defocused effect. renderRays render the rays and calculate the color where rays intersect the scene. Then the components of the vector “color” are used and averaged for the final color. This creates the effect we observe. Finally, colors are matched with the corresponding pixels.
2.

	
closer point of focus	Further point of focus

3.

	
With a low offset(focal point)	With a higher offset(focal point)

4. Everything is in extra.cpp, under the name **renderImageWithDepthOfField**. A simple helper method called **avg** is used for taking averages.

References

Environment mapping:

Example image used created by Sergej Majboroda

https://polyhaven.com/a/small_empty_room_3

Used for converting the HDRI to a cube map.

https://matheowis.github.io/HDRI-to-CubeMap/

Depth of Field:

Fundamentals of Computer Graphics, Fourth edition, p333