CG Final Project Report (Group 144)

Eddy Fledderus (student nr: 5049989)
Renāts Jurševskis (student nr: 5098688)
Thomas Sjerps (student nr: 5058287)

Work distribution

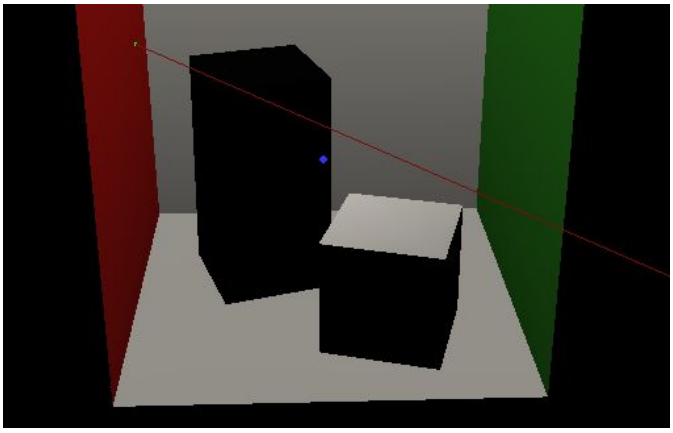
Feature	Eddy	Renāts	Thomas
Shading using Phong Illumination Model	100%	0%	0%
Recursive ray-tracer	100%	0%	0%
Hard shadows with visual debug	100%	0%	0%
Soft shadows with spherical lights	0%	0%	100%
Acceleration data-structure	0%	100%	0%
Report	33%	33%	33%
Soft shadows for other light sources	0%	0%	100%
Motion blur	0%	0%	100%
Anti-aliasing	0%	0%	100%
Cast multiple rays per pixel	0%	0%	100%
Glossy reflections	100%	0%	0%
Interpolated normals	0%	100%	0%

Features

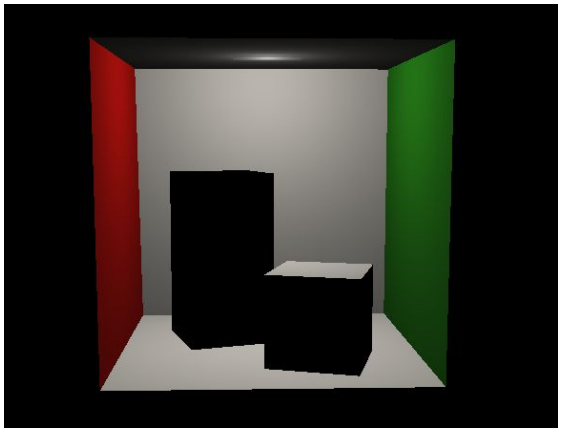
Shading using phong Illumination Model

To shade the objects with the phong illumination model we need to calculate to calculate a diffuse and specular component. To calculate the diffuse component we multiply the diffuse component of the object with the angle between the normal of the object, the vector to the light and the color of the light. The specular component is calculated by multiplying the specular component with the angle between the reflected vector from the light and the vector from the surface and the camera raised to the power of the shininess of the material.

Visual debugger



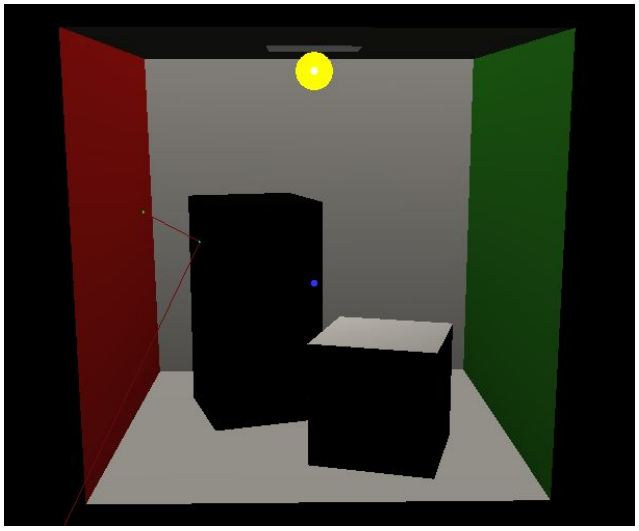
Rendered image



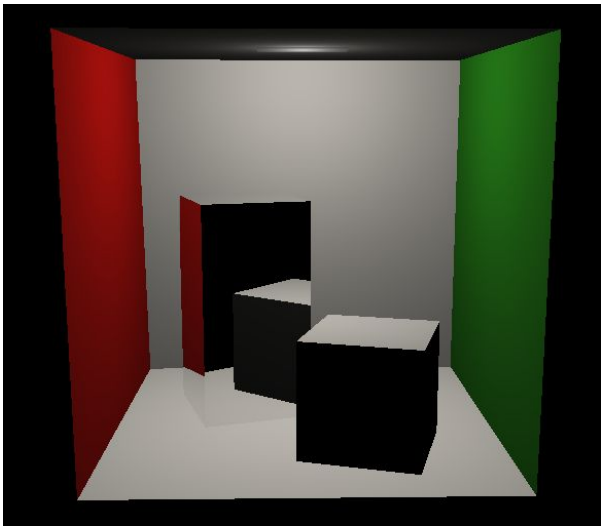
Recursive raytracer

For the recursive tracer the original getFinalColor gets the sole reason of calling a recursive function which in turn can call itself to get the color of a following ray if the surface of a material is specular. We decided to make the maximum depth of recursion to be 5.

Visual debugger

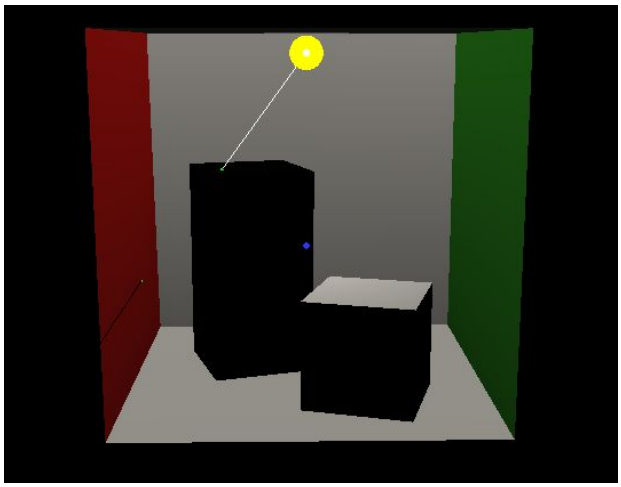


Rendered image

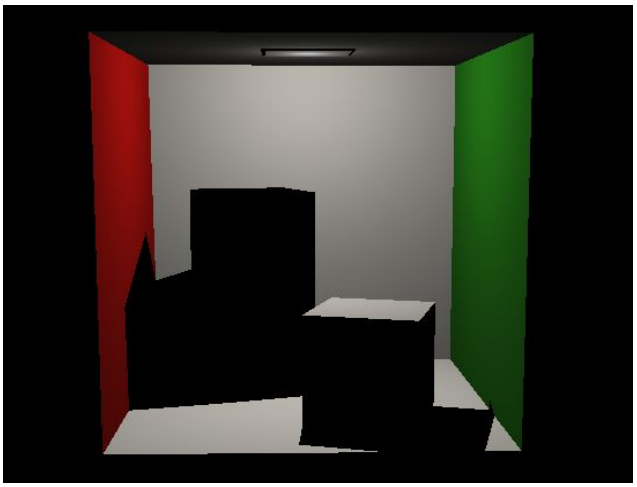


Hard shadows with visual debug
For the hard shadows we send a ray to the point that's hit by the original ray.
If the hitpoint doesn't 'see' the lightsource, the ray returns a black color.

Visual debug



Rendered image



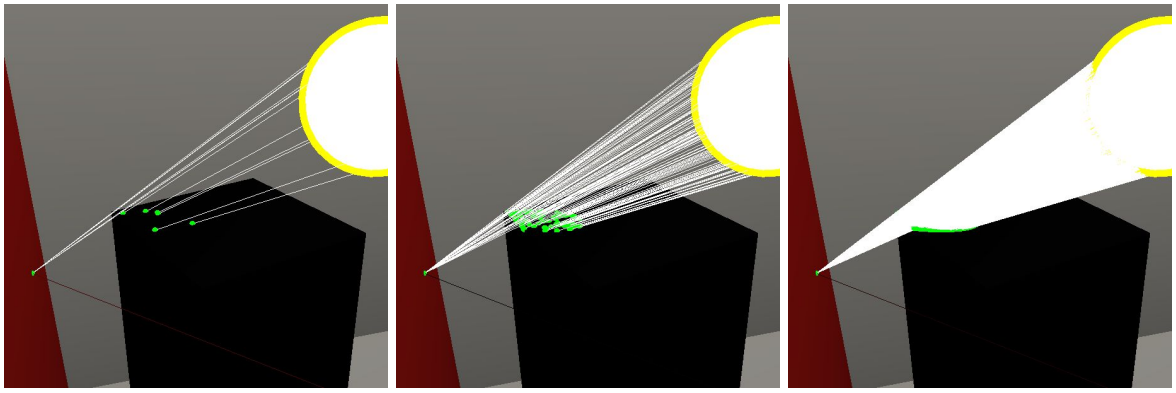
Soft shadows with spherical lights with visual debug
To create a soft shadow, a point P is sampled on the spherical light.
Our sampling strategy is as follows: first, generate a point on the unit sphere by obtaining a longitude and latitude (λ, φ) (uniformly distributed along [0,1]).
Then compute the rectangular coordinates (x, y, z) using the longitude and latitude.[1]
Obtaining P is done by translating and scaling the space such that the unit sphere now lies on the spherical light (translate by $[x_{light}, y_{light}, z_{light}]$, then scale by $size_{light}$).

Next, we check if P is on the side of the sphere that faces the vertex.
To do this, a ray is cast from P to the vertex, and if this ray intersects with the spherical light, we abort this sample of P and try again, repeating until a point that faces the vertex is obtained.

After this, light is simulated as if P were a point light, using the same hard shadow functionality.
The procedure described above can be repeated many times, and the user can select this by using the 'rays per light' slider in the user interface.

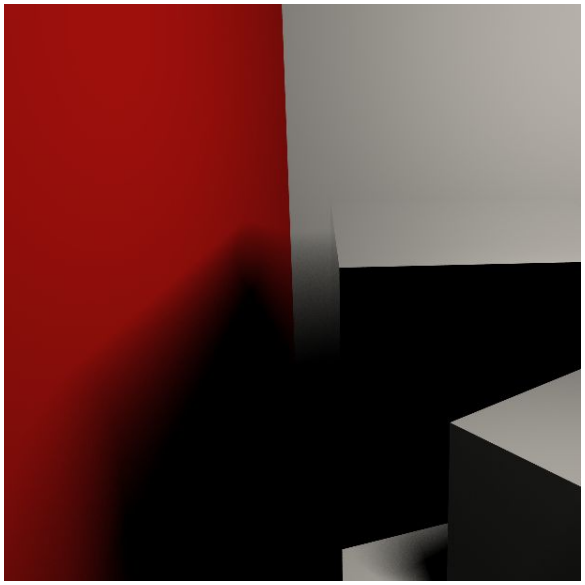
[1] point on unit sphere procedure via: https://math.stackexchange.com/a/1586185

Visual debug



(as we add more rays per light per pixel - shown here from left to right - we better approximate the true fraction of light intensity)

Rendered image



(rendered with approx. 500 light rays per light per pixel)

Acceleration data-structure

The acceleration data-structure is initialized by creating axis-aligned bounding boxes (AABBs) and recursively splitting them to decrease their total area. Each bounding box contains some triangles from the original scene. Lets denote a triangle's lowest value on some axis as its lower bound. The splitting is done by taking the longest axis of the AABB (largest distance from minimum to maximum value) and calculating the average of the lower bounds for all triangles on this axis. Then if some triangle’s lower bound exceeds this average, it is assigned to the first child AABB, and if it is smaller or equal to the average, it is assigned to the second child AABB. I selected this subdivision criteria, as its performance ended up being the best compared to other methods (calculating centroids instead of lower bounds, taking the median instead of the average, etc.). The maximum depth of the recursion is automatically calculated by the base 2 logarithm of the total amount of triangles in this scene plus a constant. From experimenting

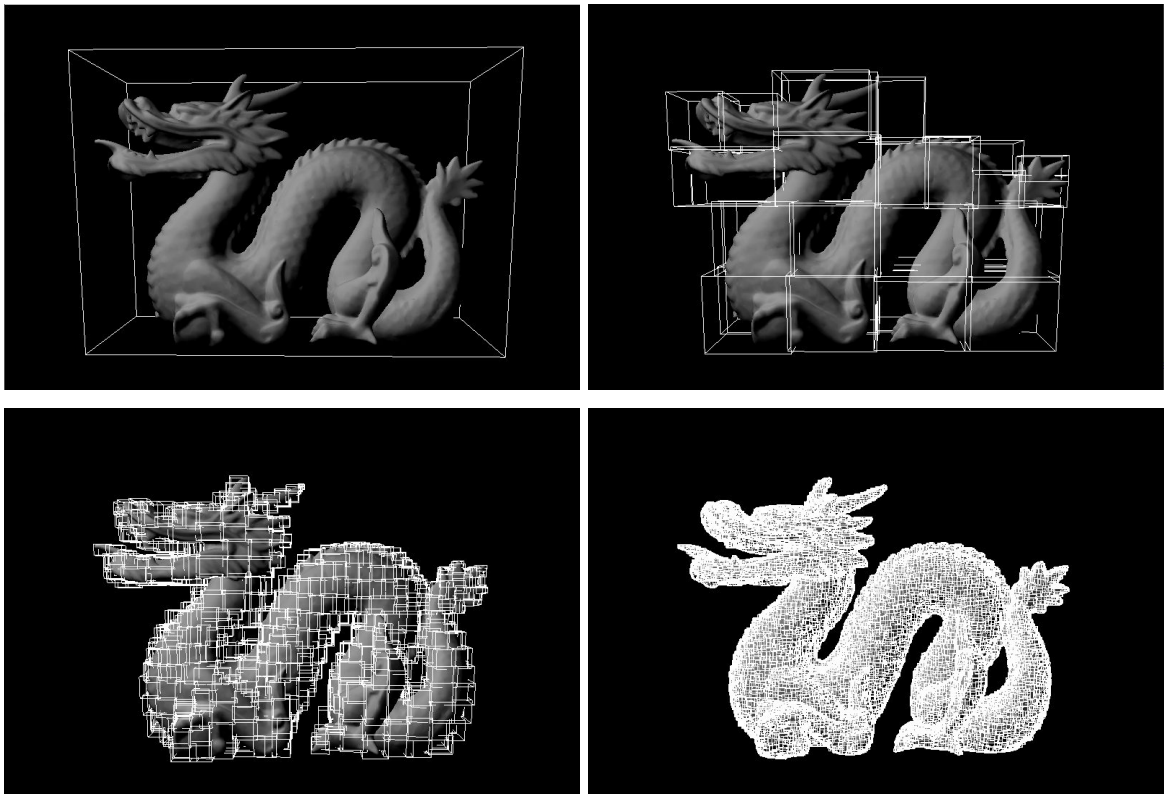
with different constant values, I ended up choosing 7, as it performed the best. In addition, any node is considered a leaf node if it contains less than 4 triangles. In this case no additional subdivision is necessary.

For drawing the debug AABBs I decided to slightly modify the requirements. Instead of displaying only the nodes that are on this current level, I decided to also show leaf nodes that were on a lower level. Since my subdivision does not always divide the triangles evenly, the leaf nodes can be on different levels. I feel like this better represents the use case of debugging, as all of the triangles in the scene are always contained in some AABB.

For calculating the intersections I made a function that recursively intersects the ray with AABBs. The recursive call is made only if the ray intersects the child AABB, and the distance to this AABB is smaller than the current closest triangle intersection. To do this efficiently, I added a function in ray_tracing.cpp that calculates the distance to some AABB without modifying the value of ray.t, while returning 0 in the case where the ray origin is already inside the AABB. Another performance optimization that was implemented is the ordering of the child AABBs. Before making the recursive calls the child AABBs are sorted based on their distance to the ray origin. This is useful, as if we already find an intersection in the closest child node, we do not need to make the recursive call for the other child node.

Some additional optimizations that I added include adding a Ox compiler flag, pre-calculating the inverse ray direction to reduce the number of times division is performed, optimizing the pointInTriangle function, and creating a custom function to get a random vector. This (together with some of the optimizations mentioned in the construction and intersection section) brought the total rendering time of the dragon model down from the initial implementation’s 800ms to roughly 57ms (calculated with the default camera location).

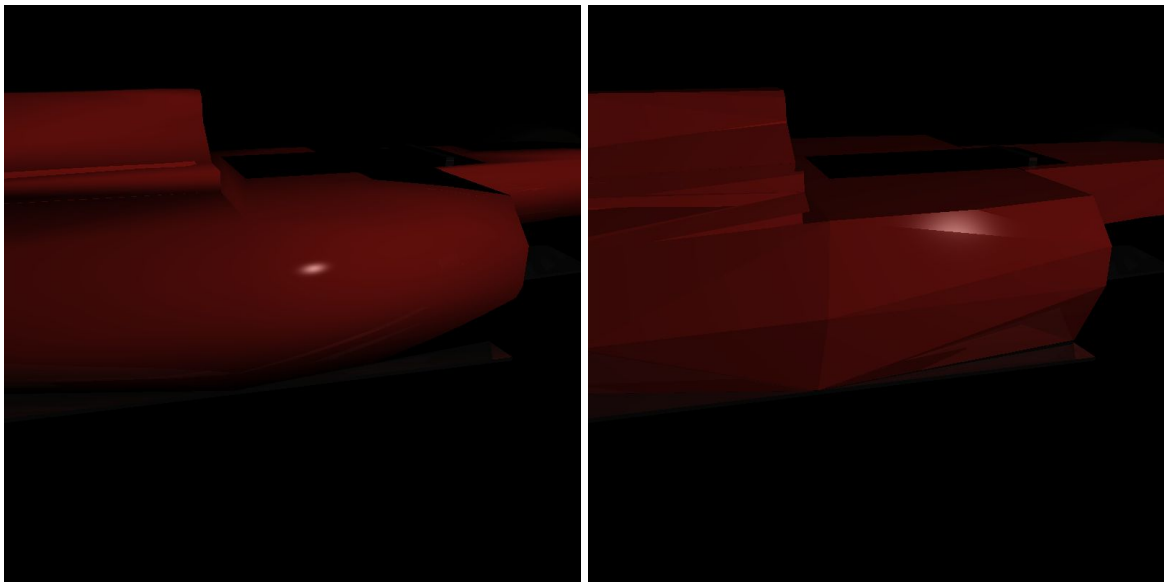
Visual debug with levels 0, 5, 10, and 20:



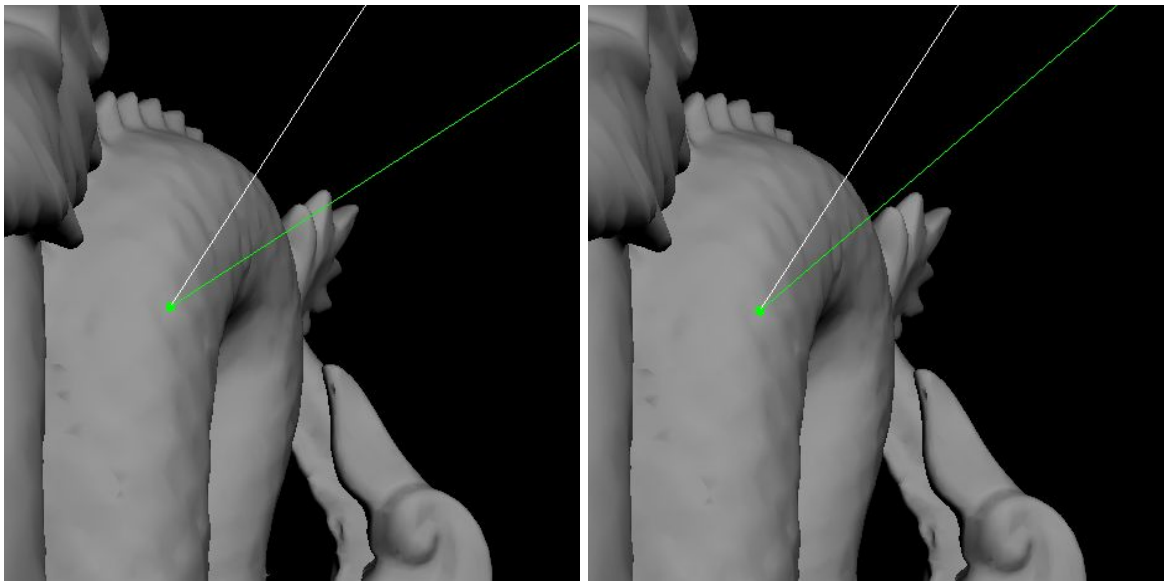
Interpolated normals

The interpolation of the normals is done when calculating the intersection of some ray with a triangle. It calculates the barycentric coordinates of the intersection point and interpolates the normals based on these coordinates. I added an additional attribute “interpolatedNormal” to the HitInfo class that stores the interpolated normal alongside the “normal” attribute. Then in the menu I added a checkbox that can be used to turn this feature on or off. It is also used in all visual debug options when this feature is turned on.

Rendered images with and without interpolated normals:



Visual debug with the same intersection point with and without interpolated normals:



Dynamic rays per pixel / Motion blur

These three methods are summarized in the Imaging panel.

When raytracing, the user can choose from 3 imaging modes:

Continuous mode

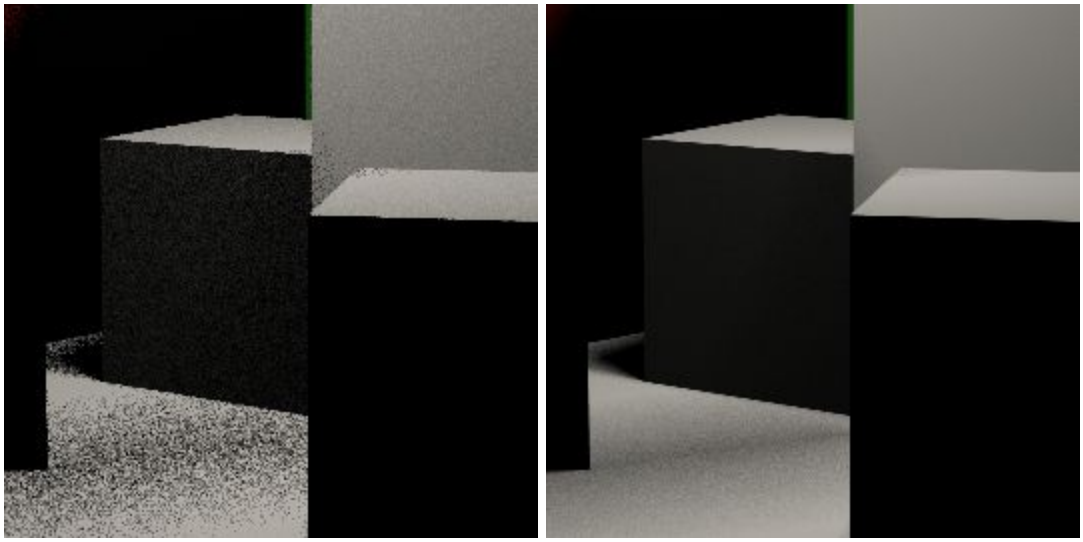
Start every frame with a blank screen and draw over it. The next frame, the result from the previous frame is discarded.

Dynamic mode

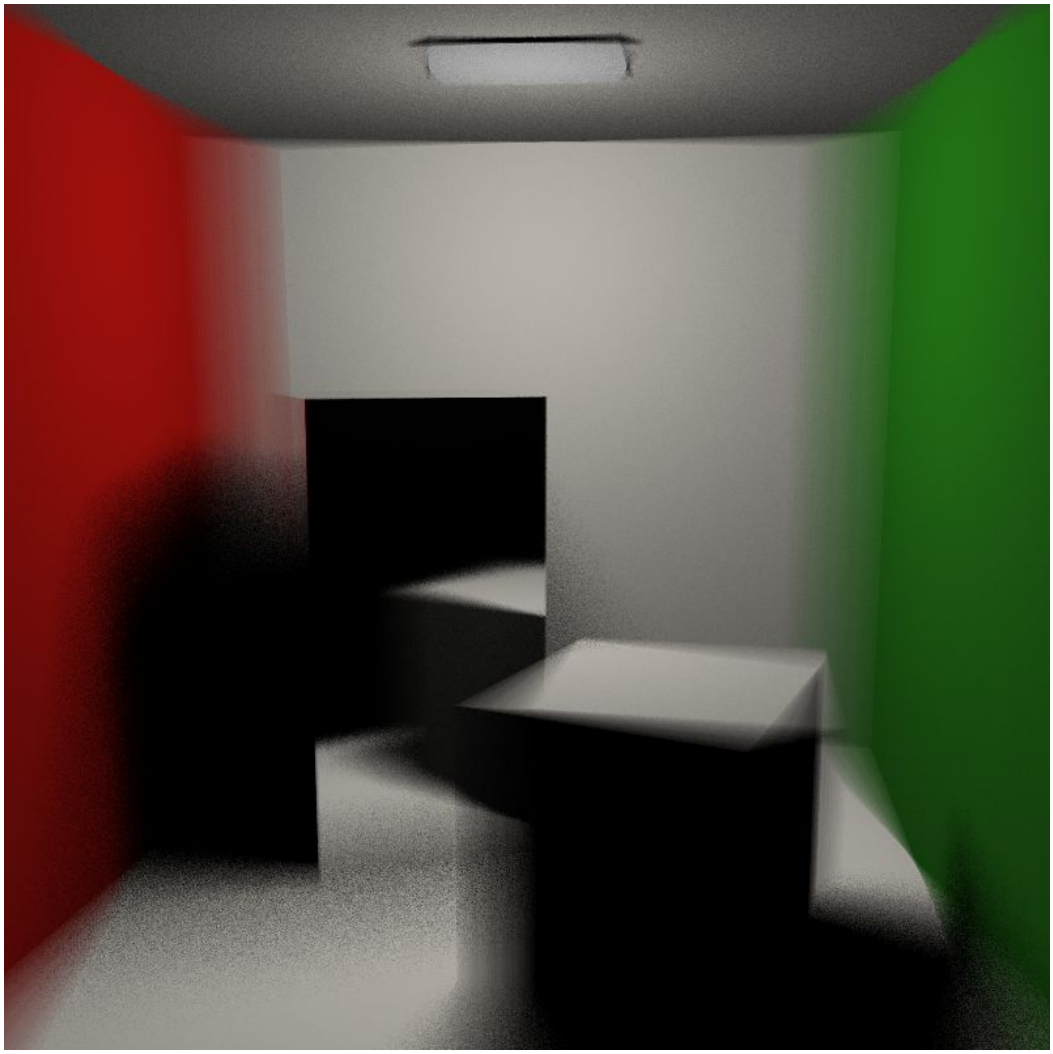
Maintain a rolling average of the results of the previous frames, and add to it every frame. When a parameter is changed or the camera is moved, clear the screen and start again.

Manual mode

First specify an exposure time, measured in frames. Then press the ‘Expose image’ button. During the exposure time, the screen will be updated with new calculations.
Moving the camera during this exposure time creates a motion blur effect, as shown in the figure.



(a stationary camera in dynamic mode produces high-quality results over time)



(moving the camera while exposing in manual mode creates a motion blur effect)

Cast multiple rays per pixel / Anti-aliasing

Anti-aliasing is implemented by casting multiple rays per pixel in a frame and averaging the resulting RGB values afterwards.

By randomly varying the starting position of a ray within its corresponding pixel, we approximate anti-aliasing.

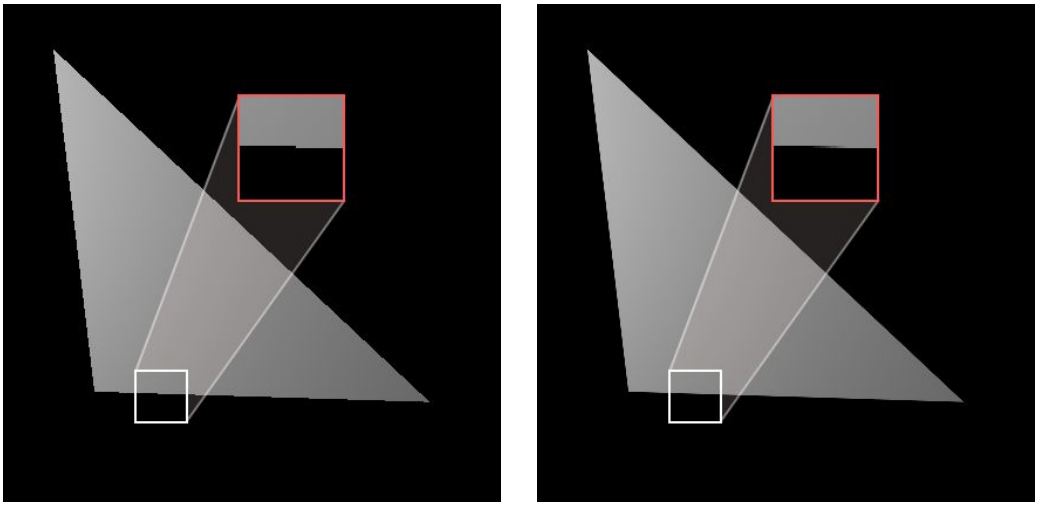
The sampling strategy is as follows: instead of shooting the ray from (x, y), add a random offset, uniformly distributed in the bounds of [-0.5, -0.5] × [0.5, 0.5].

If a user wants to turn on anti-aliasing, they can do so using a slider in continuous imaging mode, and a tick box in dynamic and manual imaging mode.

In continuous mode, the ‘Anti-aliasing rays’ slider influences how many pixel samples are averaged, which improves AA quality, but linearly impacts performance.

In dynamic and manual mode, the ‘Anti-aliasing’ checkbox turns the feature on or off.

We don't need a slider for these modes, as the AA quality will be perfect, over time. A slider to determine quality would therefore be useless.

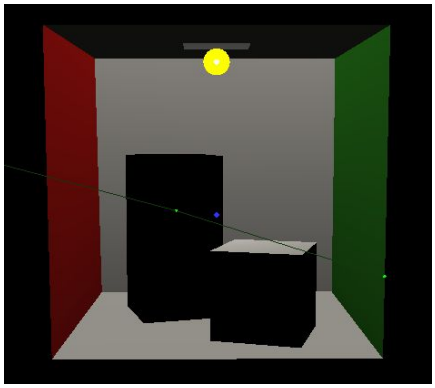


(1: no anti-aliasing leads to jagged edges; 2: anti-aliasing over time using dynamic mode leads to a smooth edge)

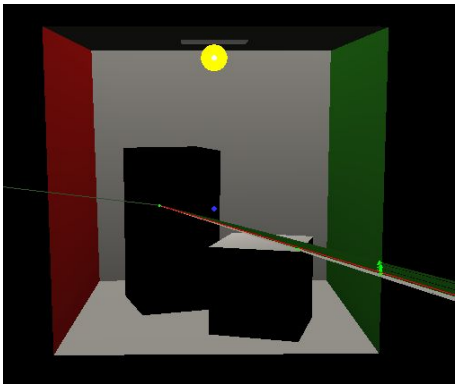
Glossy reflections

The glossy reflections work for a big part like the recursive raytracer, the difference is that the glossy reflection shoots out extra rays and takes the average color of all the rays with a weight which is the same as the dot product of the ray and the actual reflection ray. The random rays are calculated by adding or subtracting constants multiplied by two perpendicular vectors of the reflection ray.[2]

0 gloss rays (No glossy reflection)

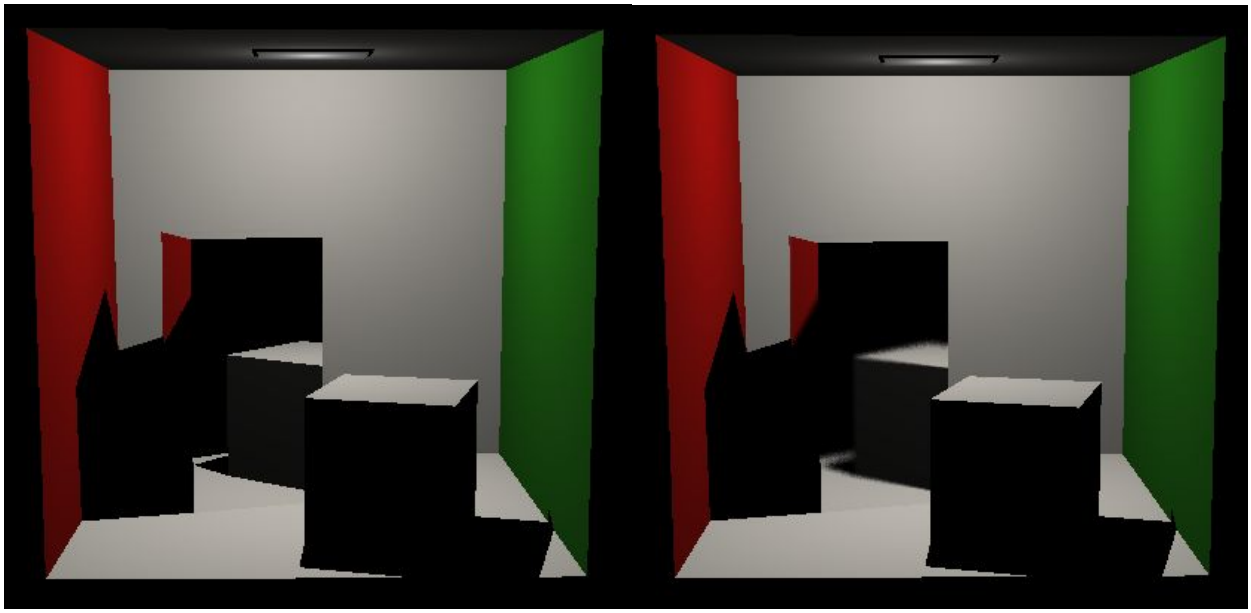


50 gloss rays



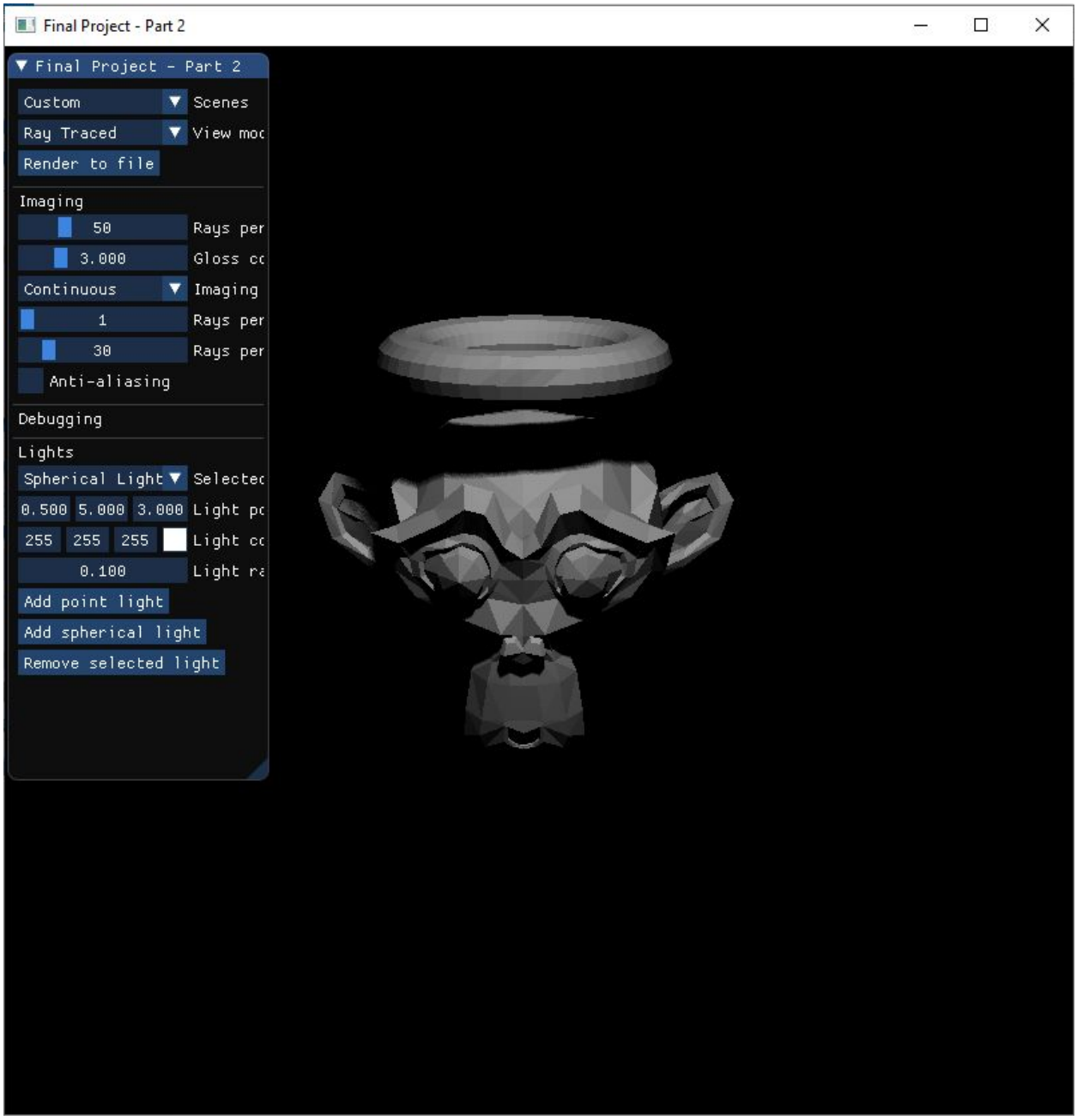
[2] Source used for glossy reflections: http://web.cse.ohio-state.edu/~shen.94/681/Site/Slides_files/drt.pdf

0 gloss rays (No glossy reflection) **50 gloss rays**



Models

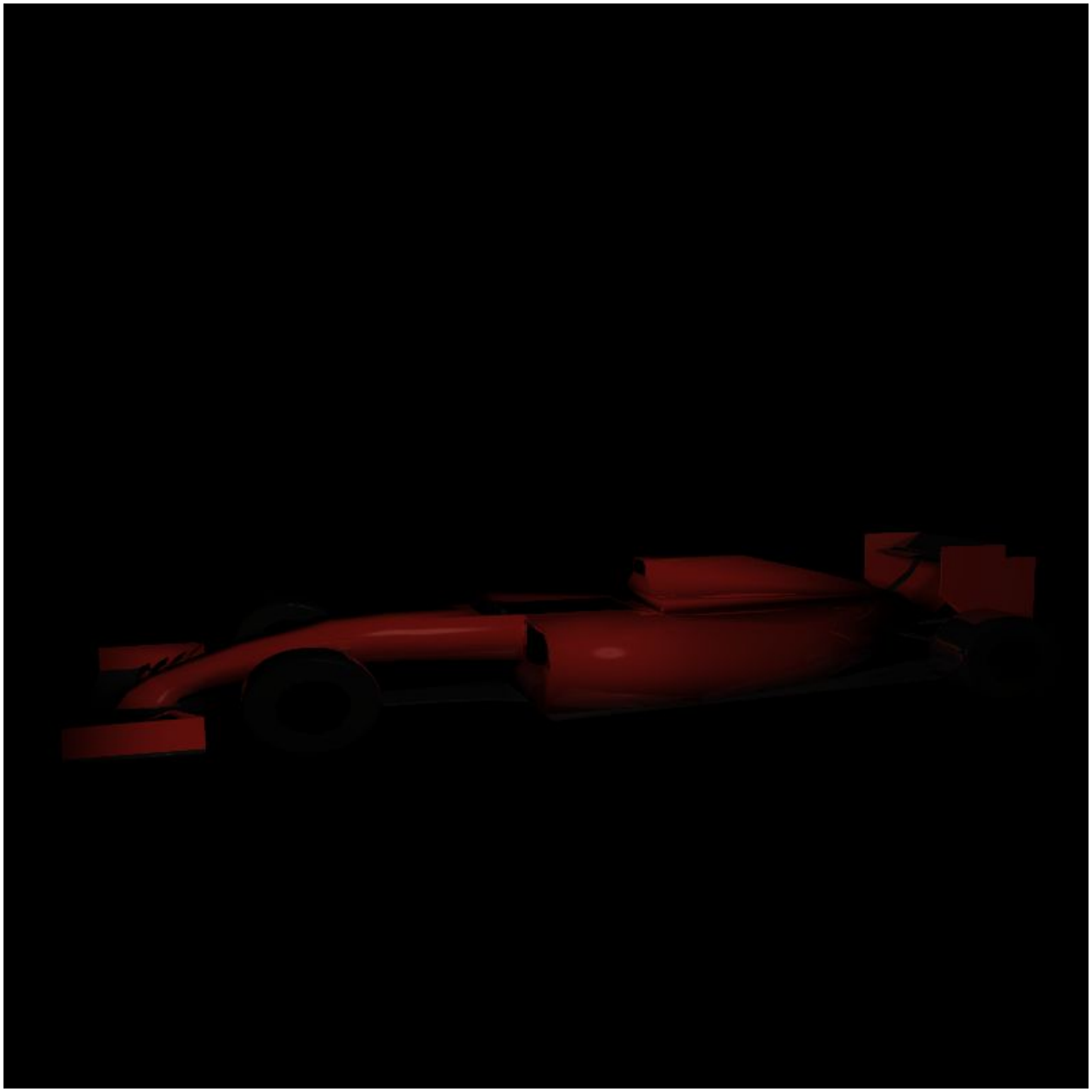
Eddy (monkey.obj)


Final Project - Part 2
Final Project - Part 2
Custom
Scenes
Ray Traced
View mod
Render to file
Imaging
50
Rays per
3.000
Gloss co
Continuous
Imaging
1
Rays per
30
Rays per
Anti-aliasing
Debugging
Lights
Spherical Light
Selected
0.500
5.000
3.000
Light po
255
255
255
Light co
0.100
Light ra
Add point light
Add spherical light
Remove selected light

Thomas (flower.obj)

(colors achieved through lighting; rendered result copied/brightened/hue-shifted/composited in Photoshop)

Renāts (F1.obj)

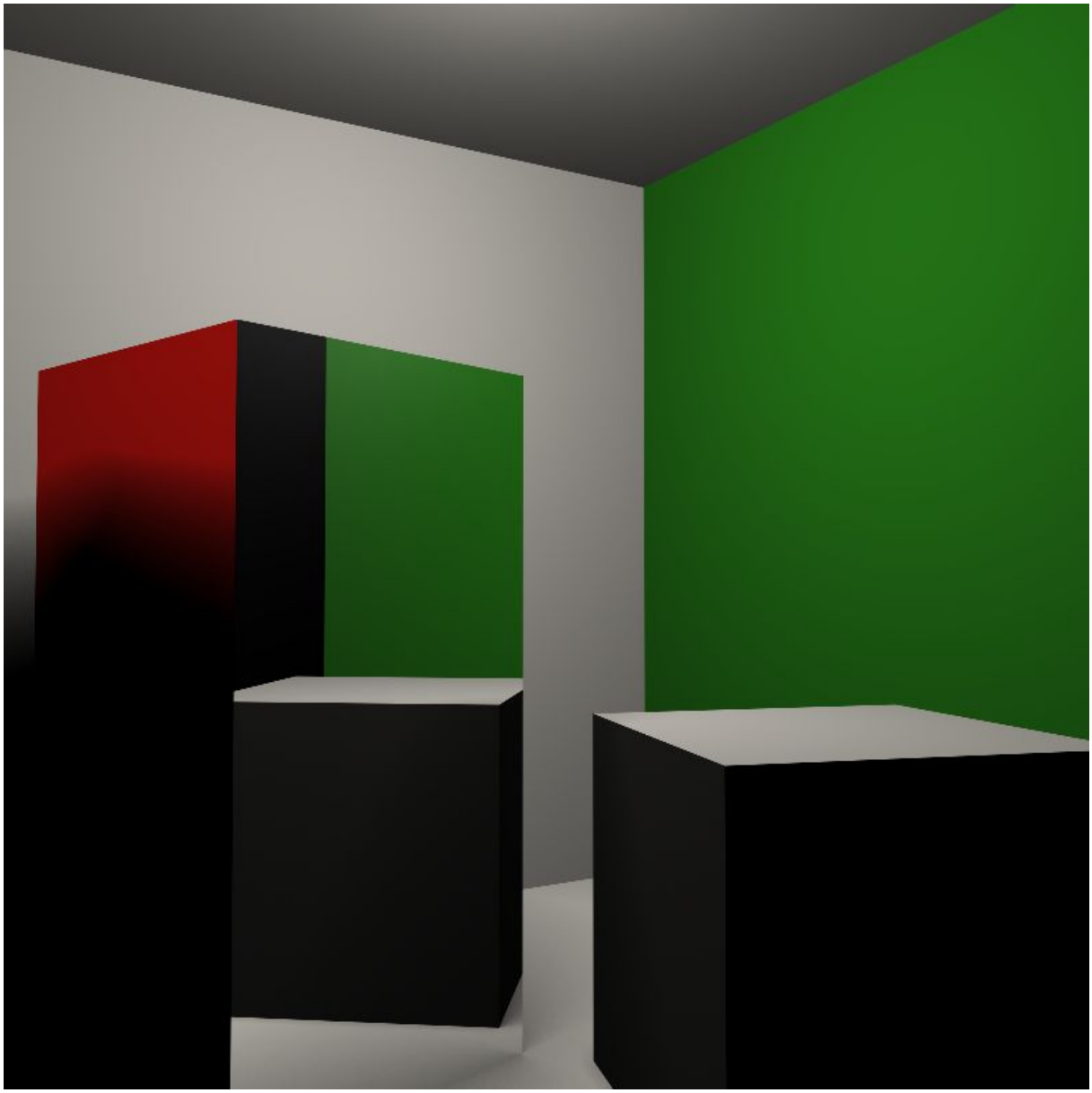


Performance test

For evaluating the performance we used the default camera location in the scene (for consistency) and performed 10 renders to file. In the table below we have included the lowest time achieved from these 10 renders. The BVH level amount is changed automatically based on the number of triangles in the scene (as is already explained in the BVH feature description). All rows in bold are rendered without any additional features, while rows in italic include some additional features..

	Triangles	Time	BVH levels
Single Triangle	**1**	**9ms**	**1**
Cube	**12**	**20ms**	**6**
Cornell box (point light)	**32**	**40ms**	**9**
Cornell box (spherical light)	**32**	**66ms**	**9**
Monkey	**968**	**65ms**	**16**
Dragon	**87130**	**55ms**	**23**
Teapot	**15704**	**96ms**	**20**
Blender Renāts	**22456**	**185ms**	**21**
Blender Eddy	**2120**	**69ms**	**15**
Blender Thomas	**15418**	**46ms**	**20**
Dragon (interpolated normals)	*87130*	*55ms*	*23*
Cornell box (spherical light, 50 shadow rays)	*32*	*1250ms*	*9*
Cornell box (spherical light, 50 shadow rays, 10 anti-aliasing rays)	*32*	*13500ms*	*9*
Cornell box (spherical light, 50 shadow rays, 50 glossy rays)	*32*	*6300ms*	*9*

APPENDIX: a nice picture



(we had this lying around; why not show it?)