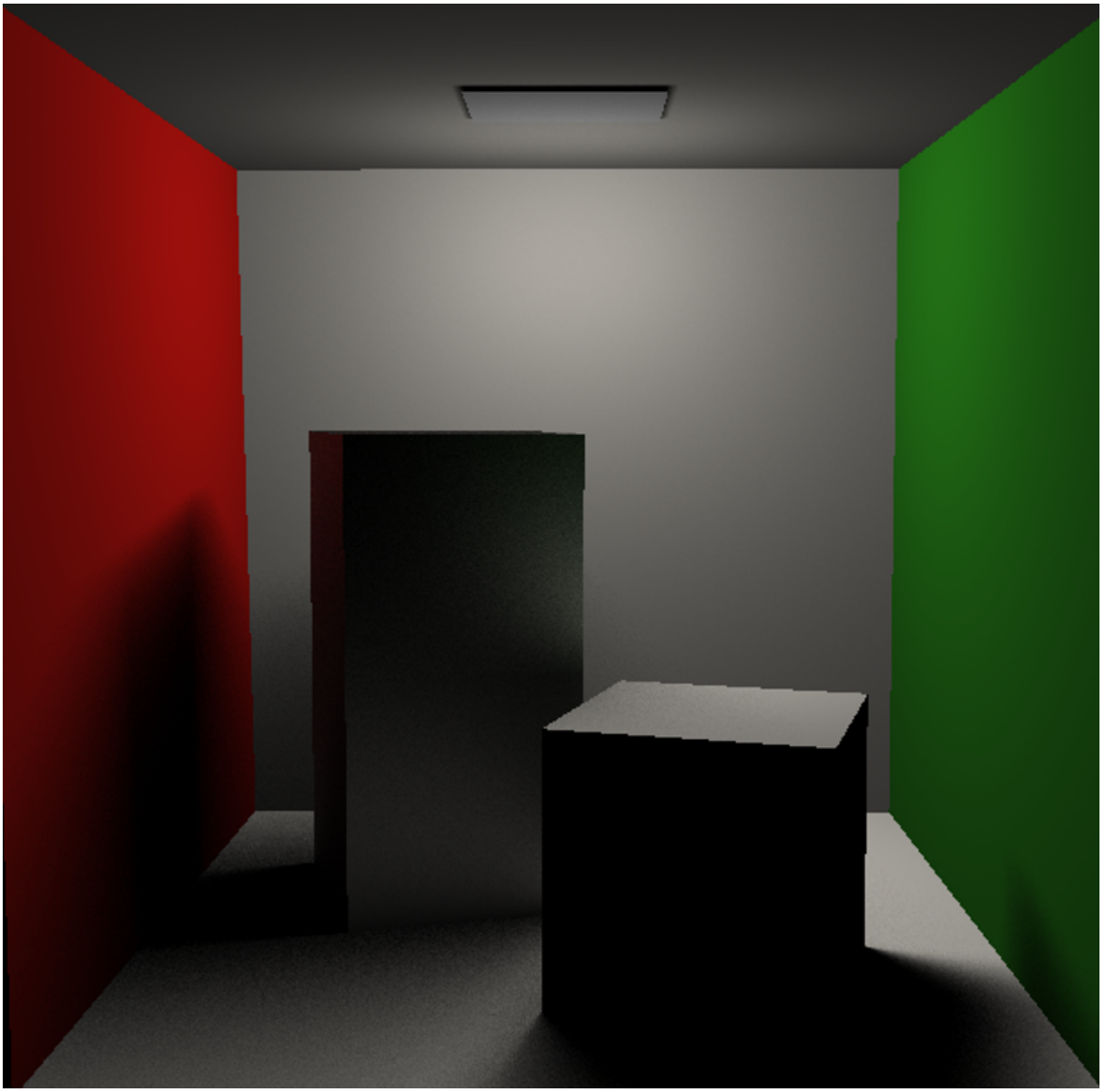
CG final project - group 42

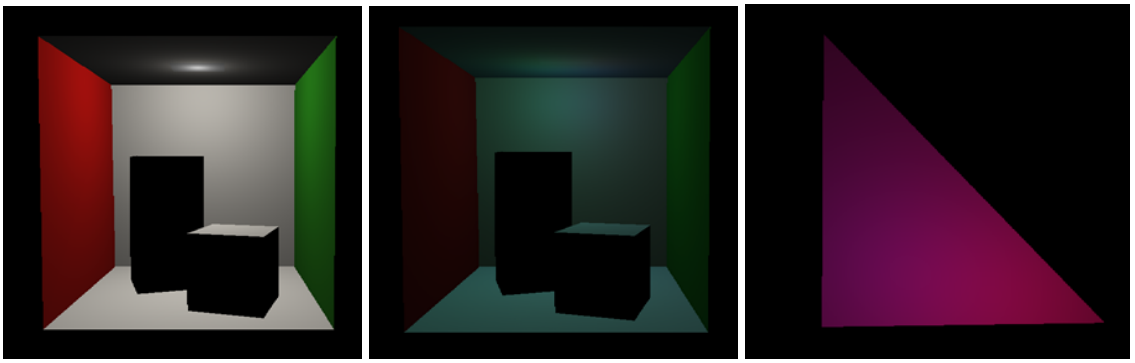
Tijmen Meijer
Andrei Dascalu
Alperen Guncan



3.1 Shading

For this feature the shading is computed in the ‘computeShading()’ function in shading.cpp. It will calculate the diffuse and specular and returns a vector3 with those 2 combined. This function is called in light.cpp, where we iterate over each light source. We then check if shading is enabled, and if that is the case we calculate it and add it to the total shading result. If shading is disabled, we return ‘materialinfo.kd’, since that is what the function returned on default before shading was implemented.

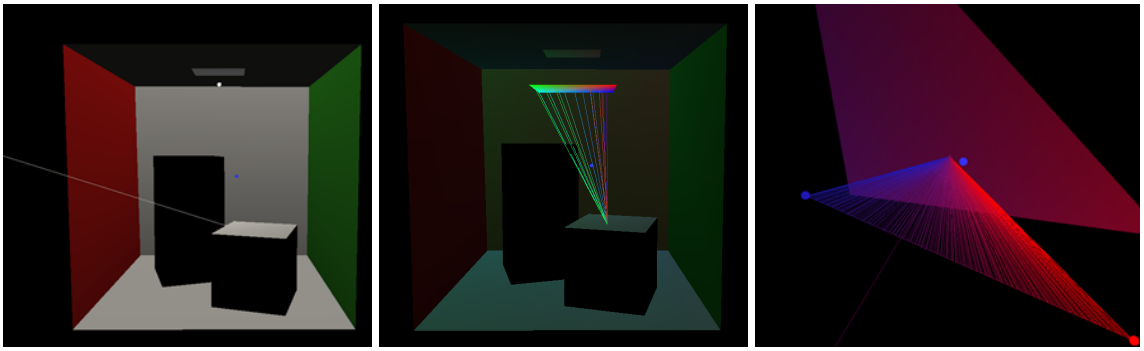
Rendered images



Point light *Parallelogram light* *Segment light*

Visual debug

The rays are colored based on the shading, this is done with drawRay() in getFinalColor():



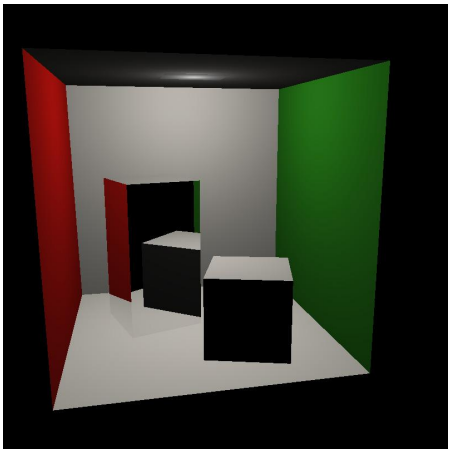
Point light *Parallelogram light* *Segment light*

3.2 Recursive ray-tracer

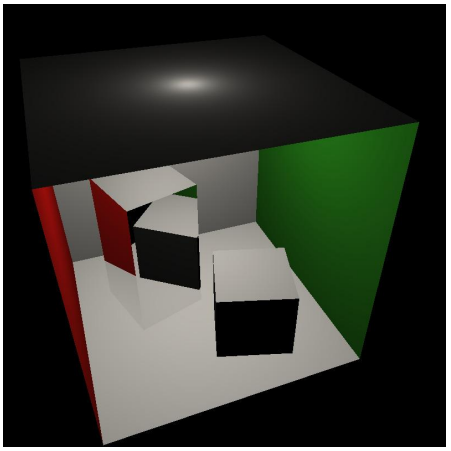
This feature is implemented inside the function 'getFinalColor()' of render.cpp. When a ray intersects a specular surface, the reflected ray is computed and traced. This reflected ray is then used recursively by calling 'getFinalColor()' with the new ray, while also contributing to the specular component with every recursion by multiplying with the material.ks of the hit surface. On every recursion level, we

decrement the rayDepth, which accounts for the maximum number of reflected rays. This process stops when either rayDepth becomes 0, meaning that we reached the maximum amount of reflected rays, or when we intersect a surface that is not specular.

Rendered images



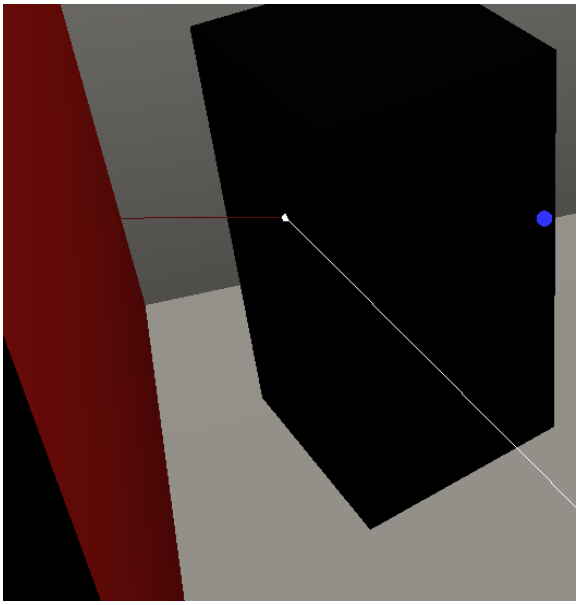
Recursive tracer with shading



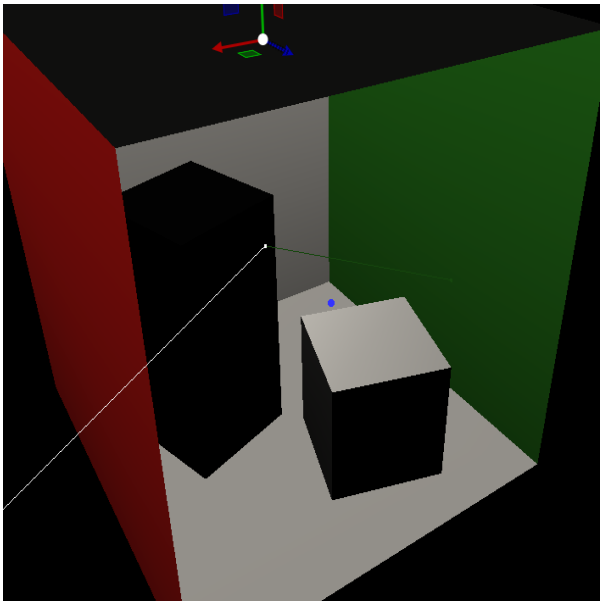
Different angle

Visual debug

We draw the reflected rays while also combining with the visual debug of shading by colouring the rays. We added a slider, under Debugging, in order to adjust the rayDepth parameter, with values ranging from 0 to 7.



Reflected ray from specular surface

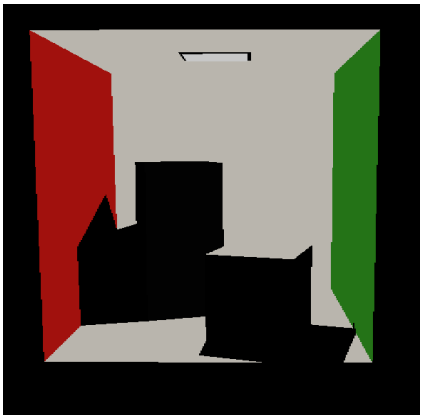
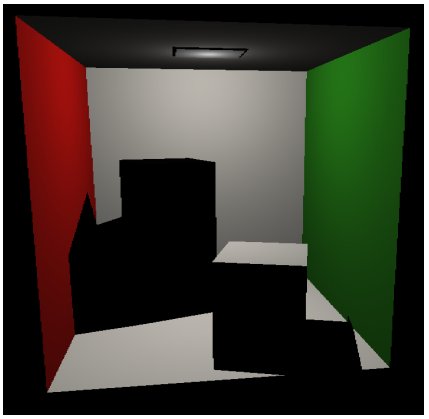


Different angle

3.3 Hard shadow

For this feature the ‘testVisibilityLightSample()’ method is called, returning a float which is either 0 or 1. A ray is shot from the light source to the intersection position to check if it intersects with any object before, which means it is inside the shadow. Based on this it returns 0 or 1. Then, in ‘computeLightContribution()’ we multiply the returned float with the shading (or materialInfo.kd, if shading is disabled), meaning that if we multiply the shading with 0, it won’t contribute, a.k.a. it is inside the shadow from that specific light source, and multiplying with 1 will keep the shading as it is. Hard shadow is only done for point lights, since segment/parallelogram lights can have multiple samples. We will explain this in ‘3.4 Area light’.

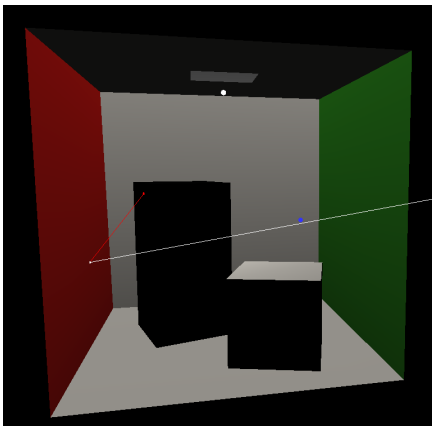
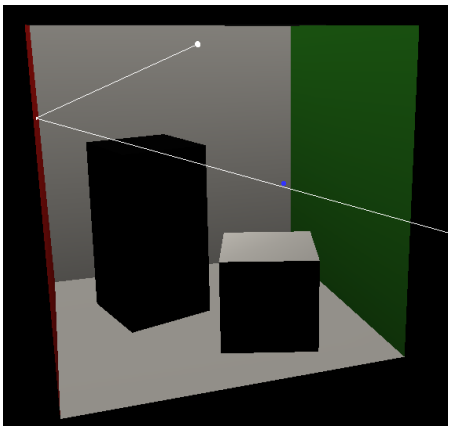
Rendered images

Hard shadow without shading *Hard shadow with shading*

Visual debug

We shoot a ray from the point light (white dot) to the intersection point. If it intersects with any objects before, we colour it red to indicate this. This means that the point will be in the shadow of that light. Otherwise the ray will be the colour of the light, in this case white.

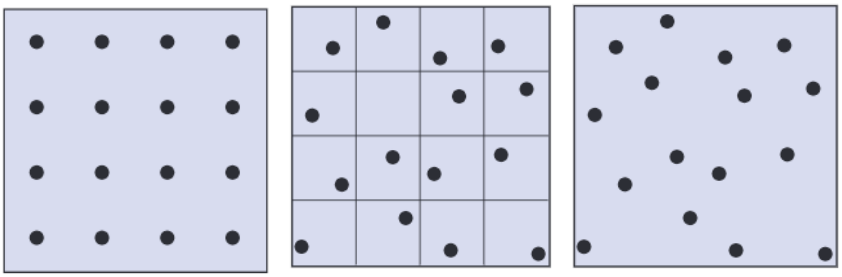



Point in shadow (red ray) *Point not affected by shadow*

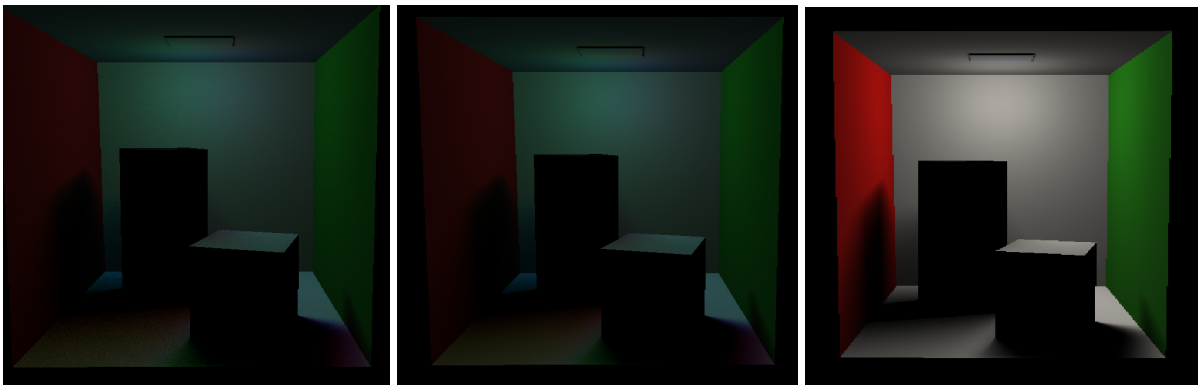
3.4 Area lights

For this feature we need to make soft shadows for both parallelogram and segment light. Soft shadow is calculated by creating a bunch of samples between the endpoints of each light source, and for each sample we apply the same methods as hard shadow, so we get a soft shadow for the light source. In order to do this, I created 2 sliders, one holding the amount of samples for parallelogram light, and one for segment light. To determine the total amount of samples, these values are multiplied with the distance between the light endpoints. The 'sampleSegmentLight()' and 'sampleParallelogramLight()' functions (in light.cpp) are responsible for calculating the positions and colour of the samples. These functions are called when we iterate over each light source in 'computeLightContribution()'. Here, we also iterate over the amount of samples, and for each sample we calculate the position by calling these functions. With a random offset the sample's position is determined to prevent it from being exactly uniformly distributed. Then, with this position, interpolation is performed to calculate the colour of the current sample. For parallelogram light we perform bilinear interpolation. The colour and position will be updated and shading will be computed. Since each sample is a fraction of the total light, we have to divide the final light contribution result by the total amount of samples.

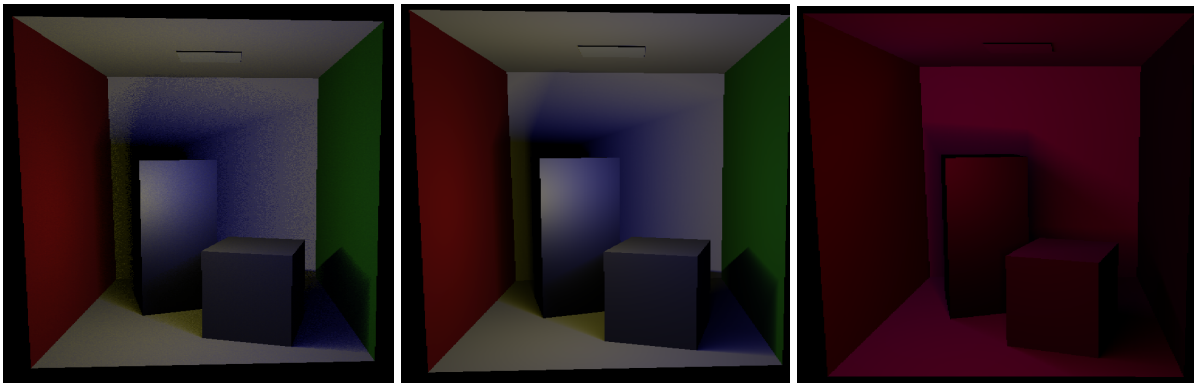
Sampling method: First we determine the amount of samples, by creating a grid based on the sliders. They are uniformly distributed. Then we apply jitter, by adding a random offset to each sample. I used the book 'Fundamentals of Computer Graphics' page 330 as source for this. The following images show a visualisation. More samples means less noise, since the grid will have smaller boxes, so the randomness offset will reduce. For segment light we do the same, except on a line instead of a grid. This randomization is added because, as stated in the book : 'One potential problem with taking samples in a regular pattern within a pixel is that regular artifacts such as moire patterns can arise'. These artifacts turn into noise by adding randomness.



Rendered images



Cornell box with parallelogram light. Left image has 100 samples and the second image has the same settings with 400 samples. If you look closely the noise will disappear by adding more samples. The last image is the default parallelogram light Cornell box but white.



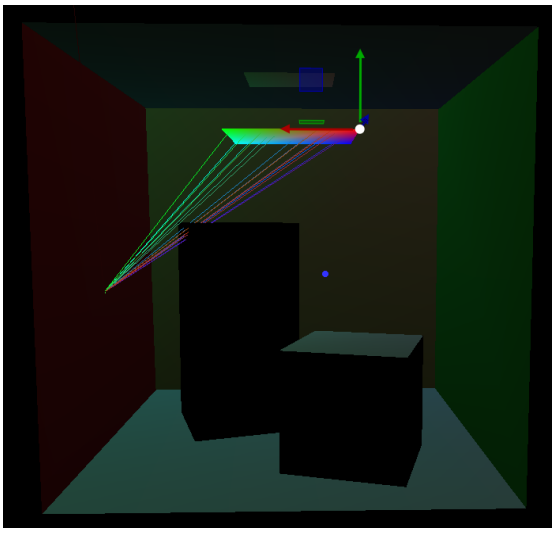
First 2 images are blue-yellow Segment lights, shading is enabled. First one with 10 samples, second image 200 samples. The last image is a red-blue segment light with 100 samples, located on a slightly different position, where the blue endpoint is outside the box.

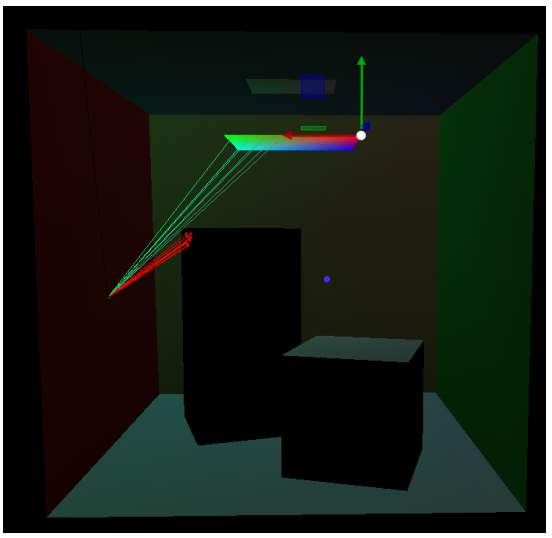
Visual debug

This visual debug will show the amount of samples per light source, randomly distributed. For the visual debug only, the random positions are determined with a seed, so the rays will always have the same position and it will be easier to see all of them. If soft shadow is enabled, it will do the same as hard shadow: If the light ray hits an object, it will turn red, indicating that the ray's position is inside the corresponding sample's shadow. With the following 2 sliders you can change the amount of samples per light source, for parallelogram light the amount of samples is in both directions:

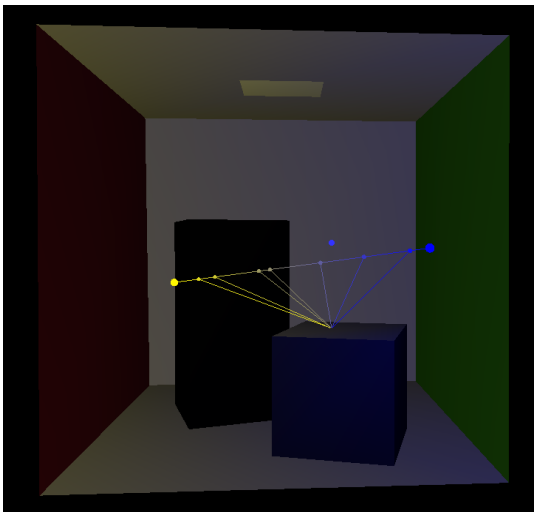
Slider to change the amount of samples per light source; More samples, better result.

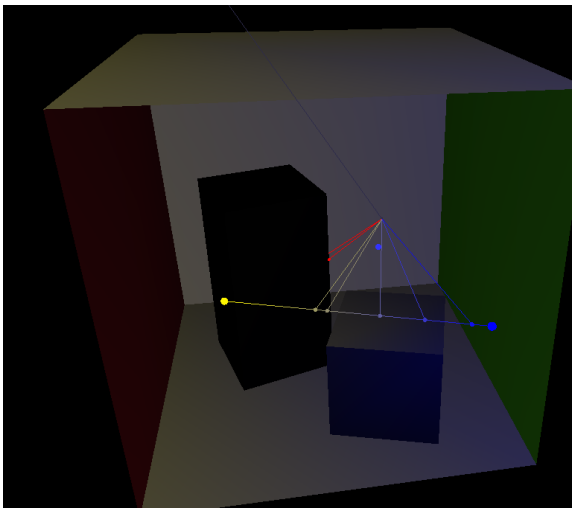
Visual debug for both parallelogram and segment light, 4 examples:





Parallelogram light; no shadow *Parallelogram light with soft shadow*





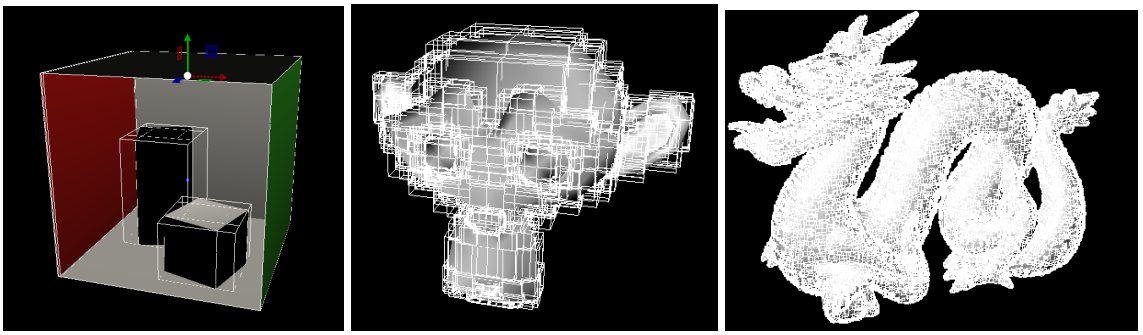
Segment light; no shadow *Segment light with soft shadow*

3.5.1 BVH Generation

This feature requires the full generation of a Bounded Volume Hierarchy for all meshes on a scene. This is done in the *bounding_volume_hierarchy.ccp* constructor method. We first build a vector that stores all the primitives in the scene (using the custom struct Prim), which reduces the time that it takes to generate it, but increases memory usage by a bit. Both triangles and spheres are assigned onto the vector, and for each we store the centroid, its Axis-Aligned Bounding Box, its triangle/sphere id, and the mesh id it's contained by. After we construct the primitive vector, we recursively build the hierarchy with the *ConstructorHelper()* method. For each call stack we store the **index and level** of the current node being processed, the **ids of the prims** that the current node should split to its leaf nodes (or it should contain if it's a leaf node itself), **the Nodes vector,** and its **parent node index** (while it can be derived from the level if we were using a full tree representation of the Node

vector, we lease that assumption for time and space performance, so it does not follow that the child node of a parent is 2*[its index] and 2*idx + 1). Then at each call we either assign it all the remaining primitives if the current node is a leaf or only 1 primitive has remained, or in the case of an interior node, we sort the prim_ids vector by a revolving axis (starts at x and goes through all with repeat), split at the middle to get the median, and continue the recursion with the two split vectors. Of course, at the end of each call stack we push the current node with all the needed fields: the AABB that is calculated with the helper method *calculateAABB()*, a boolean for whether it's a leaf node or not, its level in the hierarchy, and the ids of the next nodes or final primitives. The primitives are split on a right-ended median, thus the case of odd number of primitives or equal centroids is resolved by 'over'-filling the right child node.

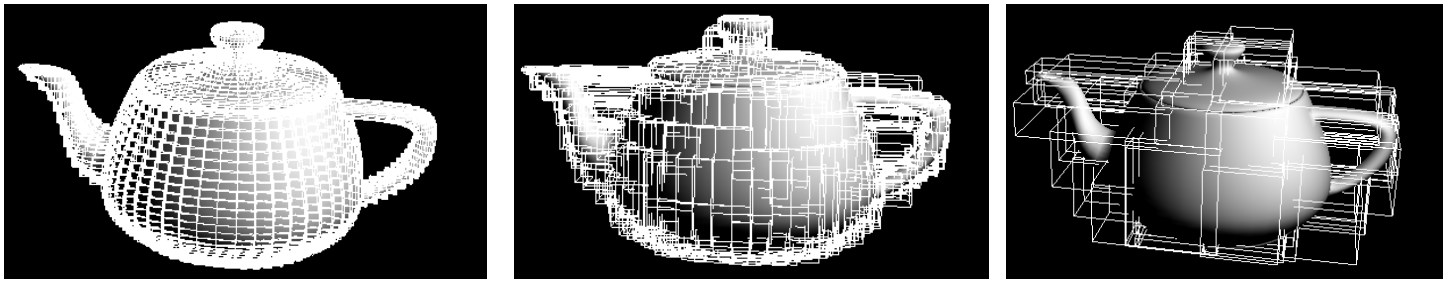
Rendered images



Last level of the BVH for the Cornell Box (6 levels including root level), Suzanne Monkey (11 levels), and Stanford Dragon (18 levels).

Visual debug

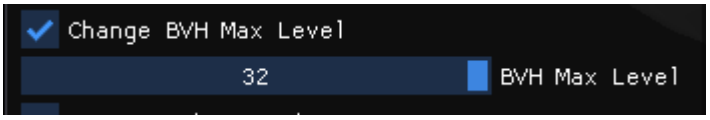
There are three visual debugs to ensure the BVH generator's correctness. One is already used to show the BVH renderings above, namely the level visualizer. This one draws all the AABBs of all nodes on the level that is selected with a slider on the UI.



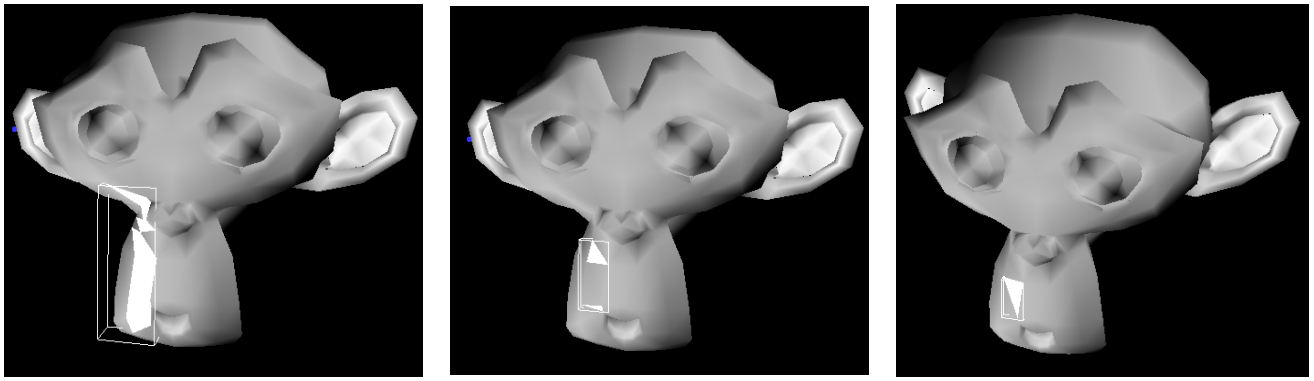
Utah teapot with visualized (from left to right): 14th/last level, 10th level, and 6th level

The second one is a leaf node visualization, which highlights the selected leaf node by id and all its contained primitives (the traversal for finding the matching id is the same as the way the BVH is being generated, namely a pre-order traversal). While this debug is not by itself all that useful, it pairs well with the third debug we have for BVH, a slider for changing the maximum allowed level of the

BVH. This way one can reduce the max level to force more primitives per leaf node, and check if all primitives are contained in the node properly.



Max Level BVH slider



Combining the last two visual debugs (leaf node and max level changer), to display different nodes and their triangles of the Suzanne Monkey. From left to right: 37th leaf node on max level 6 (~15 triangles/node), 290th leaf node on max level 9 (2 triangles/node), and 544th leaf node of fully expanded BVH (1 triangle/node).

3.5.2 BVH Traversal

For the traversal of the acceleration data-structure, we start off in bounding_volume_hierarchy.ccp, inside the ‘intersect()’ function. If BVH is enabled, we make a call to the 'traversal()' function, which is used to recursively traverse the nodes of the acceleration structure. We always traverse the closest intersected node first, so when calling the function recursively, we always start with the closest node intersected so far. In order to optimise the traversal, we keep track of a value called absoluteT. This value represents the closest distance to a triangle found so far in a leaf node. If any of the AABB's are located further away when compared to absoluteT, we can be sure that no other triangle that is closer will be found in the respective AABB's, so we simply ignore them and continue traversal.

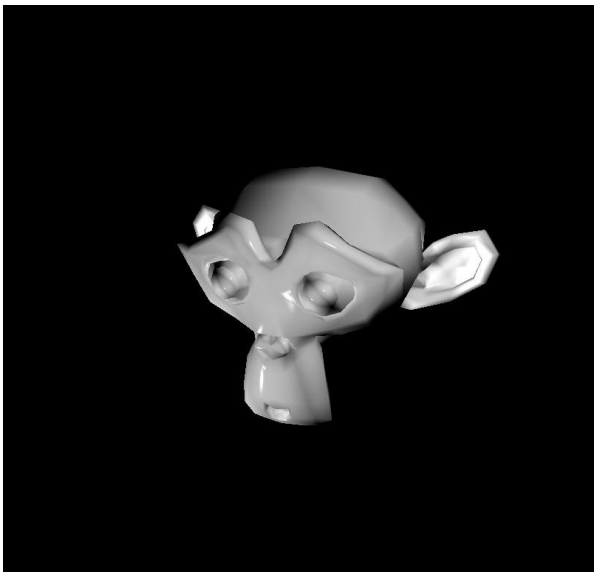
Rendered images

Time to render image: 57132.3 milliseconds

*Teapot rendered **without** BVH enabled*

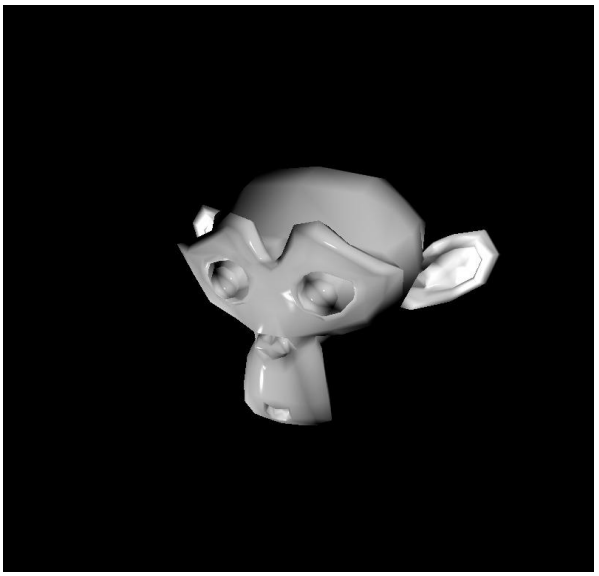
Time to render image: 3247.91 milliseconds

*Teapot rendered **with** BVH enabled*



Time to render image: 3146.48 milliseconds

*Monkey rendered **without** BVH enabled*

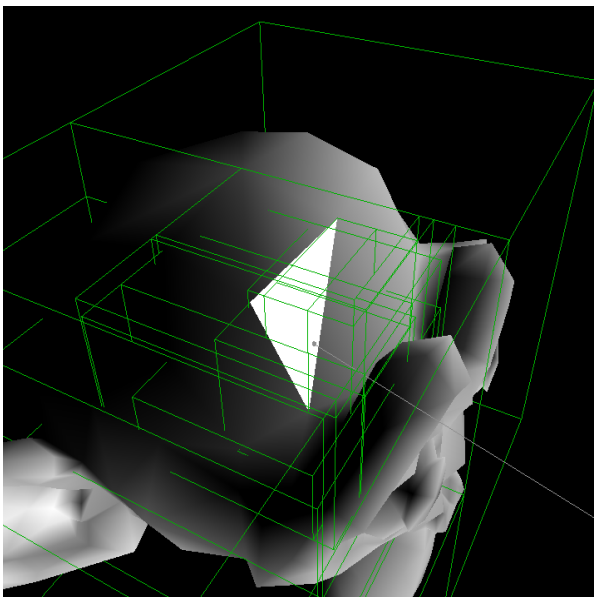


Time to render image: 580.812 milliseconds

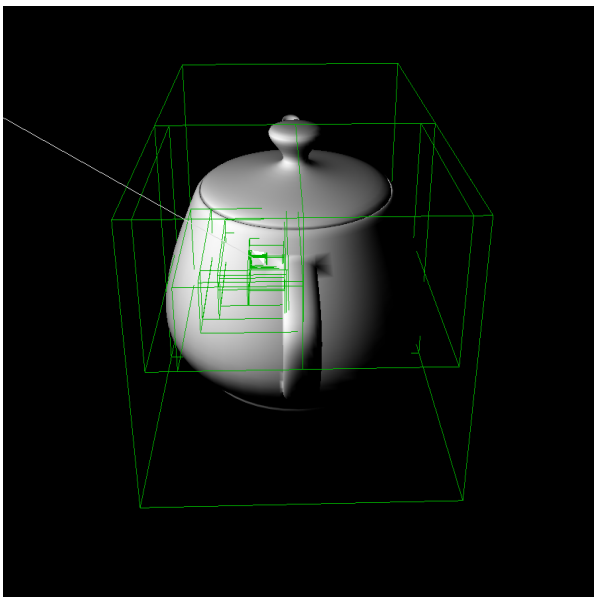
*Monkey rendered **with** BVH enabled*

Visual debug

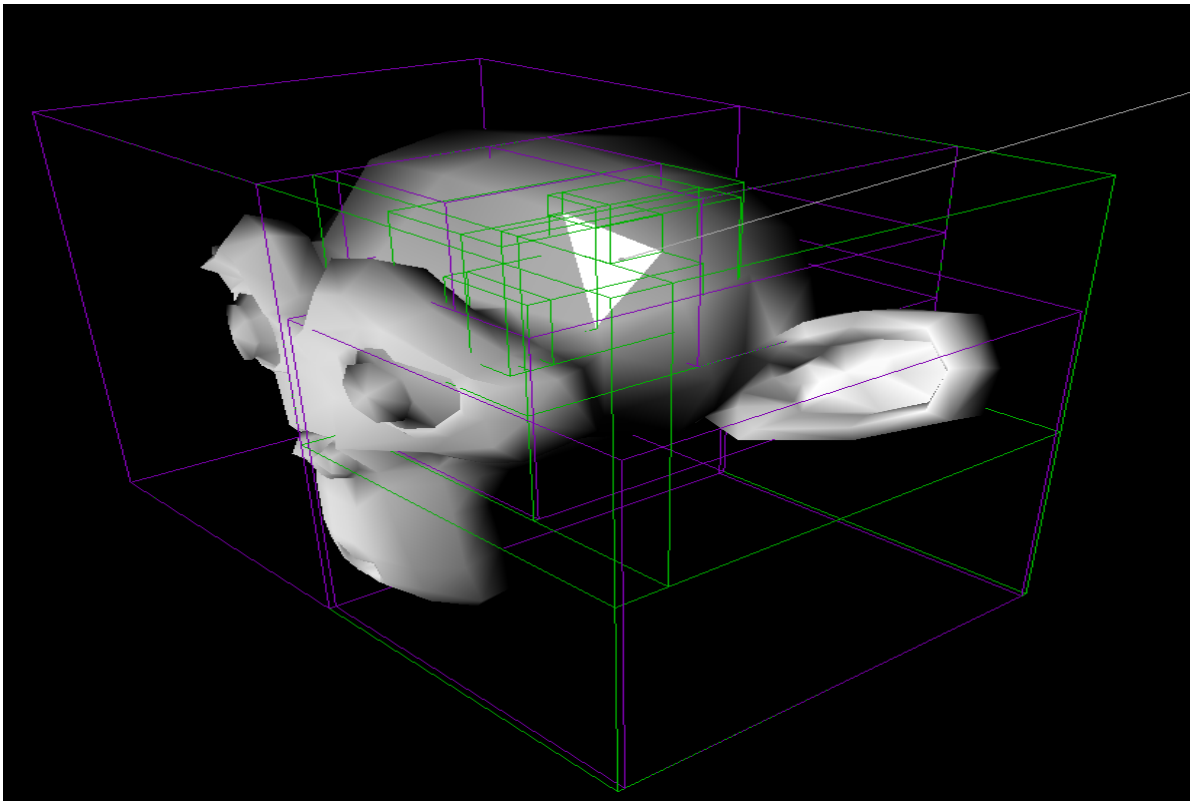
In order to visualise the process, we shoot a ray at any particular model. AABB's that are intersected are outlined with the colour green, followed by the highlighting of the final intersected triangle. When enabling the 'Intersected but not traversed' feature, under Debugging, AABB's that were intersected at one point but further discarded due to the optimization will be highlighted with the colour purple.



Intersected AABB's outlined with green on monkey



Intersected AABB's outlined with green on teapot

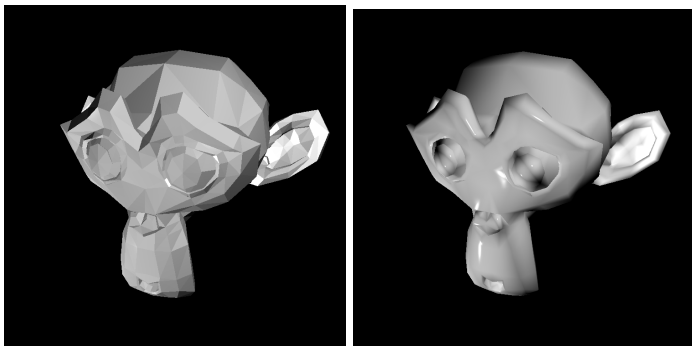


Intersected and traversed AABB's outlined with green, while intersected but not traversed AABB's outline with purple

3.6 Normal interpolation with barycentric coordinates

In order to achieve a “smoothing” effect over models, we interpolate over the normals of vertices for a particular model. This is done inside the ‘intersect()’ function located inside Bounding_volume_hierarchy.cpp. Once we find the desired triangle, we use the ‘computeBarycentricCoord()’ function to compute the barycentric coordinates of its vertices, and then proceed to use the obtained coordinates to interpolate over the normals of the vertices using ‘interpolateNormal()’. The normal of the hitInfo is then updated correspondingly to the interpolated normal.

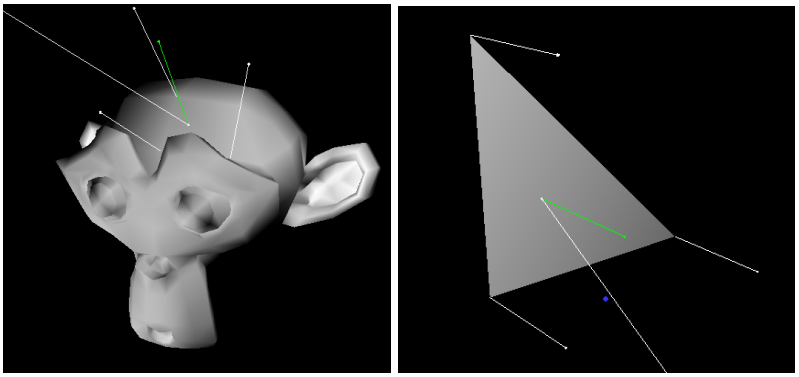
Rendered images



Monkey without and with interpolation.

Teapot without and with interpolation.

Visual debug

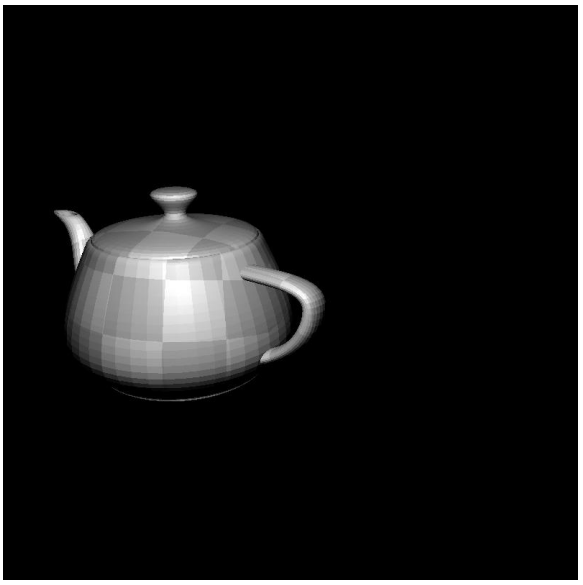


The green ray is interpolated by the vertices in both images.

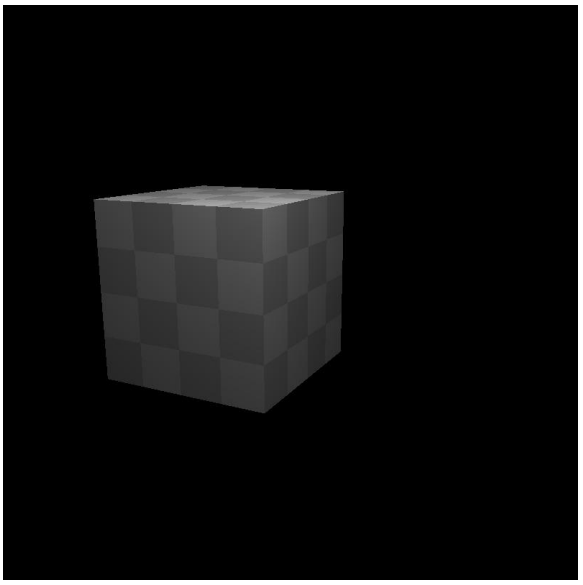
3.7 Texture Mapping

For this feature, a check is made in bounding_volume_hierarchy.cpp, inside of the 'intersect()' function to see if a mesh contains a texture image (if texture mapping is enabled). In the case in which a texture image is present, we interpolate the three texture coordinates of the corresponding triangle using the 'interpolateTexCoord()' function, and we then use the interpolated texture coordinates in 'acquireTexel()' in order to get the corresponding texel from the image, which will then be used for the material diffuse coefficient. We account for coordinates that are outside of the texture image range by using repeat mode.

Rendered images



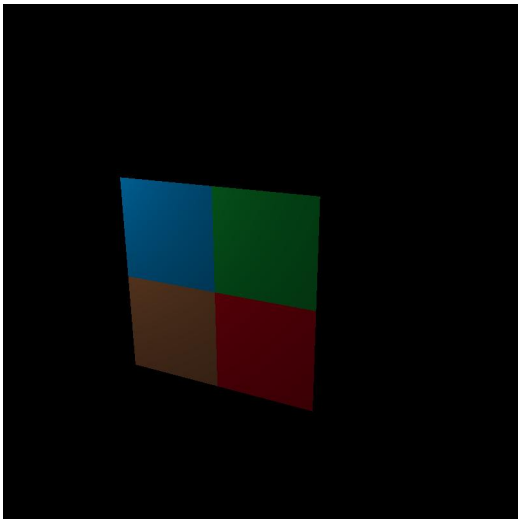
Texture mapping and shading on teapot

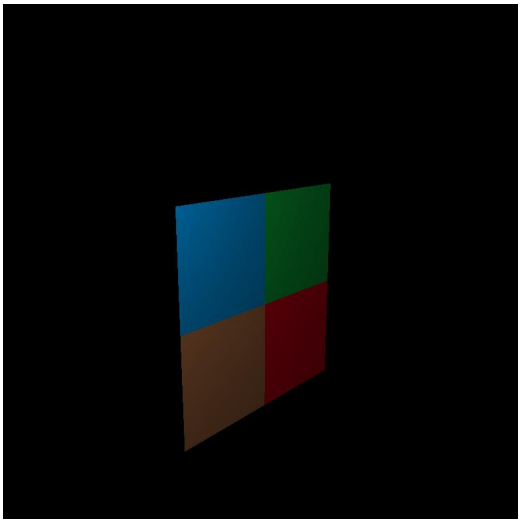


Texture mapping and shading on cube

Visual debug

In order to visually debug this feature, we created a quad textured with a 2x2 texture with 4 colours, each corresponding to a single pixel: red, green, blue and brown.





4.1 Environment maps

Rendered images

Visual debug

4.2 SAH+Binning criterion for BVH

Used sources: Lecture slides on Acceleration Data Structures

This tasks requires for two optimizations to the Bounding Volume Hierarchy generation, namely SAH (surface area heuristic), and Binning, which is used to speed up the creation of BVH with SAH. All the needed changes to adapt the BVH generation for SAH are integrated into the original constructor methods, by modifying the *calculateAABB* method, and by adding a helper method for calculating the surface area of an AABB. Now we will discuss each one of the optimizations separately:

SAH

We differentiate by the two generation methods using a boolean flag as an argument to the constructor. If SAH is enabled, then we have to change our split criteria from the original. The way it is done is by using a modified formula of the general Surface Area Heuristic, which for our purposes of minimization actually leads to the same result, but quicker. The simplification works on two assumptions: one is that the program takes nearly the same amount of time to check whether a triangle is intersected by a ray for all types of triangles, which is a fair assumption given our implementation. The other is that the rays are equally likely to intersect an AABB no matter the angle at which they are incident to the box. The latter is used to derive p_A and p_B as being nothing but the ratio of surface areas of the split bounding volumes to the entire volume:

$$\begin{aligned} c(A,B) &= t_{trav} + p_A \sum_{i=1}^{N_A} t_{isect}(a_i) + p_B \sum_{i=1}^{N_B} t_{isect}(b_i) \\ &= t_{trav} + p_A t_{isect} N_A + p_B t_{isect} N_B \\ &= t_{trav} + t_{isect}(p_A N_A + p_B N_B) \end{aligned}$$

Since we are working with an optimization problem, we can safely drop some of the variables in the above formula. The intersection time is constant for any triangle, hence for minimizing the cost function it is a redundant variable. Since we are not working with a look-ahead SAH optimizer, the time it takes to traverse the node is the same for any split permutation we test for a given depth level of the hierarchy, hence why that one can also be dropped for the sake of optimization.

$$
\begin{aligned}
min(c(A,B)) &= min(t_{trav} + t_{isect}(p_A N_A + p_B N_B)) \\
&= min(p_A N_A + p_B N_B) \\
&= min\left(\frac{SA(A)}{SA(A+B)} N_A + \frac{SA(B)}{SA(A+B)} N_B\right)
\end{aligned}
$$

At the end, we are left with only having to find the surface areas of each split, with the possibility of caching the surface area of the current node for all future calculations (its reciprocal will even be better, though that is quite the small micro-optimization). Finally, the total surface area requires the doubling of each unique face's area, but since we divide two total surface areas, we can omit the multiplication and just find the ratio of the two half-surface areas.

Finding the surface areas of an Axis-Aligned Bounding Box can also be optimized quite nicely. Since all sides are aligned of the box, it is very easy to find their lengths, namely it is the difference between the upper and lower bounds of the box. Furthermore, finding the dot product of that vector with its one-shifted dimensional vector leads to the formula we wish to use:

$$
\begin{pmatrix} v_x - u_x \\ v_y - u_y \\ v_z - u_z \end{pmatrix} \cdot \begin{pmatrix} v_y - u_y \\ v_z - u_z \\ v_x - u_x \end{pmatrix} = \begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix} \cdot \begin{pmatrix} y_d \\ z_d \\ x_d \end{pmatrix} = x_d y_d + y_d z_d + z_d x_d = \frac{SA(AABB(u,v))}{2}
$$

[where u and v are lower and upper bounds of AABB, and $[x|y|z]_d$ are the side lengths of the AABB.]

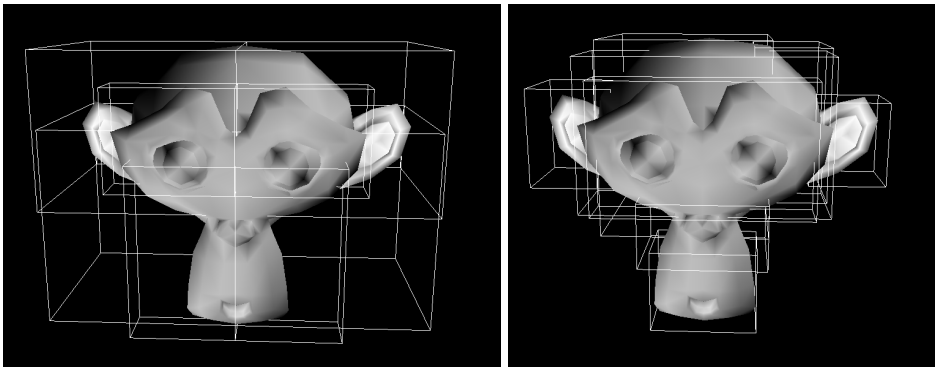
Other than minimizing this metric for each possible split of an AABB, there's not much else to the heuristic: You iterate through all possible axes, sort the ids for each axis, calculate the cost for each split (where *each split* is defined as checking a split on every primitive's centroid), while saving the index of the primitive with best split and the best axis. Then we split the ids at that point and everything else is the same as with the base BVH generation.

Binning

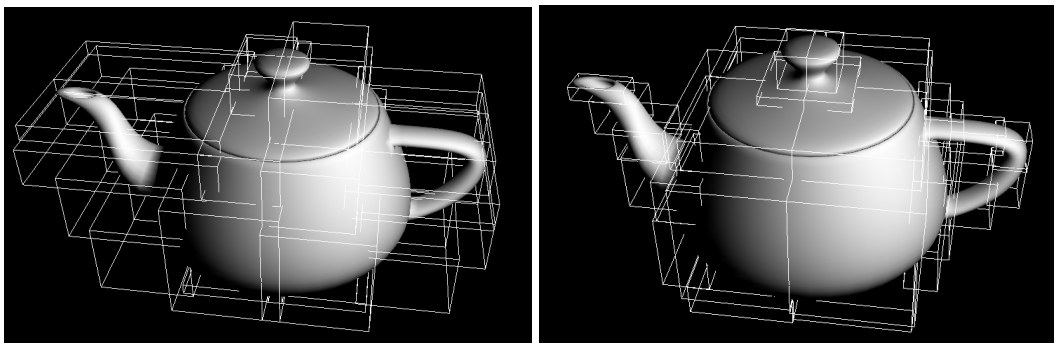
The above heuristic would have been more than enough… only if it was not painfully slow for most models since you basically iterate through the entire list of primitives three times for each axis, and that's not including the stacking effect of repeatedly having to iterate through the same ids. This is where the binning optimization comes into play. The more primitives there are, the more diminishing the returns are of iterating through each possible split of the bounding volume, hence why we limit that to a certain number of bins. This only applies, however, when the number of bins is less than the number of primitives, otherwise it is more performant to do regular SAH with checking every centroid separately. When it can be done, however, it is quite the noteworthy speedup. What we have to do is to split the volume that is between the two furthest centroids to several uniformly spread bins and check the centroids for each bin grouped as a whole. The number of bins is configurable in the UI, defaulted to

64. The splitting procedure is implemented with the better performing sweeping version of the SAH - instead of creating a new AABB for each split, we incrementally add new primitives to the left AABB, so that for each iteration we only have to check the min-max of a single centroid to the box, instead of the min-max of all centroids below the current bin boundary index, effectively saving up most repeated calculations. Similarly to regular SAH, we do this for every axis and save the best performing split.

Rendered images



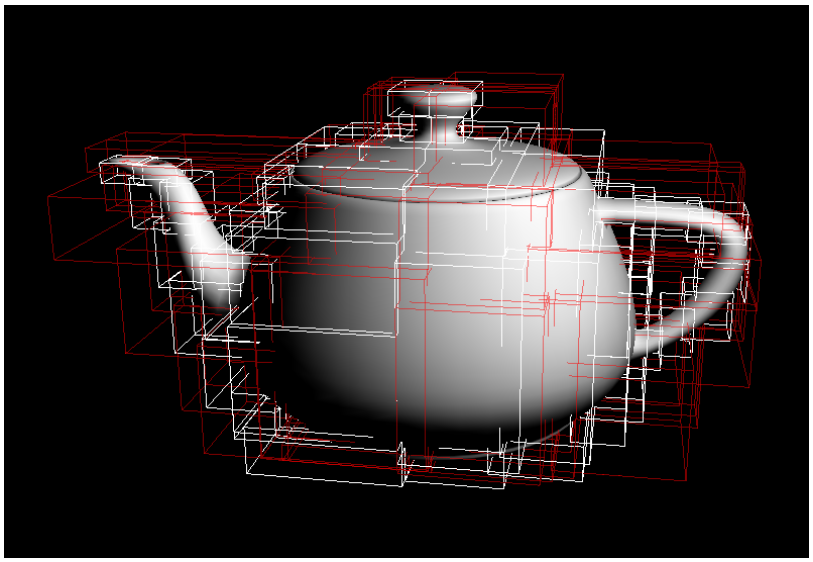
SAH- and regular BVH side-to-side for Suzanne Monkey (level 3), and Teapot (level 5). Note the savings on empty space between the two generation models.

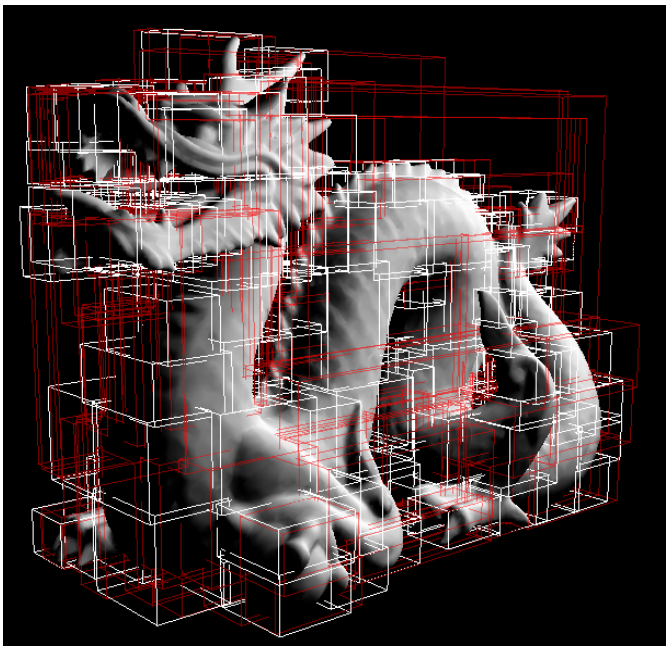


It is quite clear that while it might take a bit longer for SAH-BVH to generate, it is definitely worth the wait, as it cuts down on the empty space quite well.

Visual debug

The extra debug for this feature is the visualization of the savings on SAH compared to BVH. Since regular BVH generation is much quicker than SAH for medium to a large number of bins by orders of magnitude, it does not hurt for the debug to generate a second BVH in the background and overlay it with the one of the SAH BVH. This way the red parts will only show the parts of the AABB that are not included in the optimized BVH, giving a good idea of how much this algorithm speeds up intersection tests.





Showing savings of using SAH over BVH for Utah Teapot (on level 5), and Stanford Dragon (level 7).

4.3 Motion blur

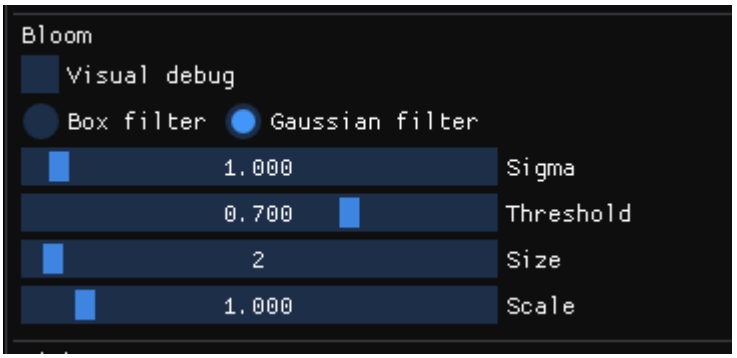
Rendered images

Visual debug

4.4 Bloom

Used sources: https://www.opencv-srf.com/2018/03/gaussian-blur.html
https://codereview.stackexchange.com/questions/169655/create-a-two-dimensional-gaussian-kernel

For bloom I added a lot of features to the UI. Since there are multiple ways to apply a filter to the thresholded colours, I decided to add both box filter and gaussians distribution filter. You can switch between these in the UI. Gaussians filter will give a relatively smoother result, but is a bit harder to compute. I have chosen to add Gaussian's filter because its result is more realistic than the box filter. When gaussian's filter is selected there will appear a slider where the sigma can be adjusted to change the spread. The other 3 sliders are for the threshold, size and scale of the bloom filter. There is also a checkbox for debug mode.



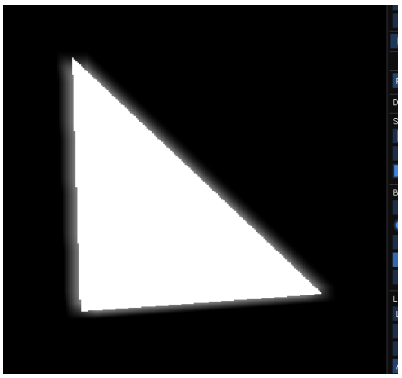
In render.cpp after calculating all pixels, the 'addbloom()' method in 'bloom.cpp' is called. Here we create a new array containing all pixels having an RGB value above the threshold. Next, in 'filterPixel()' we either calculate the average of all pixels in the box (for box filter), or we create a gaussian kernel and apply that filter to the pixel. This kernel is created with the 'gaussianKernel()' function. The standard deviation is based on the slider's input value, same for width and height. Since we have a set amount of pixels, the gaussian kernel will calculate discrete values for each pixel, shown in the image below. Normally we would multiply the function with 1/(2*pi*sigma^2) but since it is discrete we will divide it by the sum of the boxes to make sure they all sum up to 1. After the filter is applied, we update all pixels on the screen and multiply this with the scale. If debug mode is enabled, we set the pixels to the filtered pixel, so we can show how the applied filter looks. Otherwise we add each filtered pixel to the original pixel.

$\frac{1}{273}$

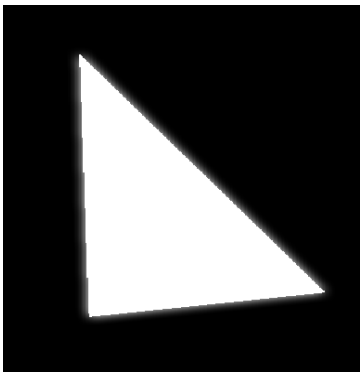
1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

$$g(x,y) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/(2\sigma^2)}$$

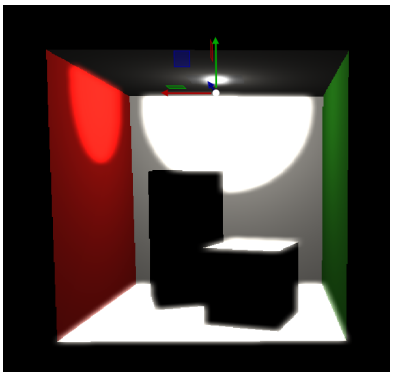
Rendered images



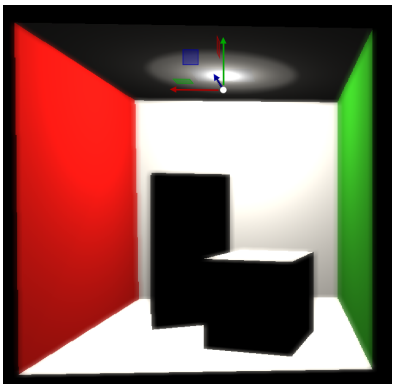
Box filter + threshold 0.7 + size 10 + scale 1



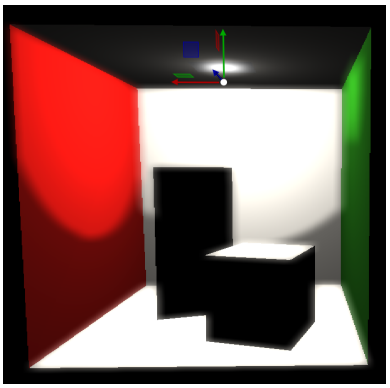
Gaussian filter same settings + sigma 5



Gaussian filter + sigma 5 + size 6 + scale 2 + threshold 0.55



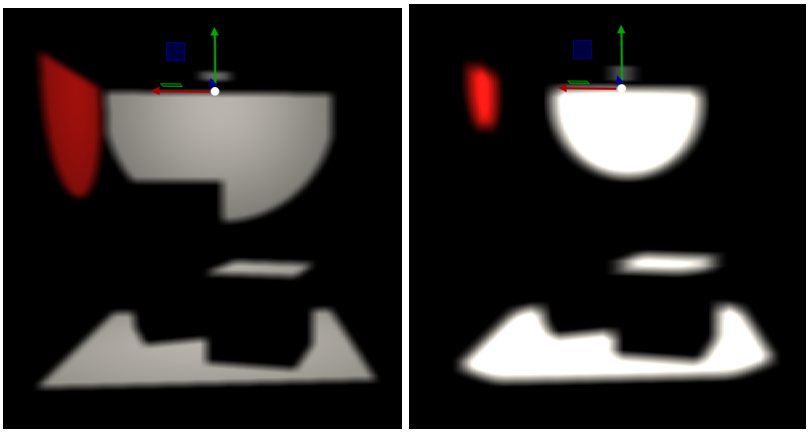
Box filter + threshold 0.2 + size 6 + scale 1



Gaussian filter + sigma 10 + size 10 + scale 1 + threshold 0.4

Visual debug

For this visual debug, there is a special checkbox, since it is easier to show this in ray tracing mode. It will show the filter applied to the image after thresholding the colours. This way you can see where the bloom will be applied.



Left: Gaussian filter + 0.5 threshold + sigma 10.0 + size 6 + scale 1.0
Right: Box filter + 0.6 threshold + size 10 + scale 2.0

4.5 Texture filtering: Bilinear Interpolation

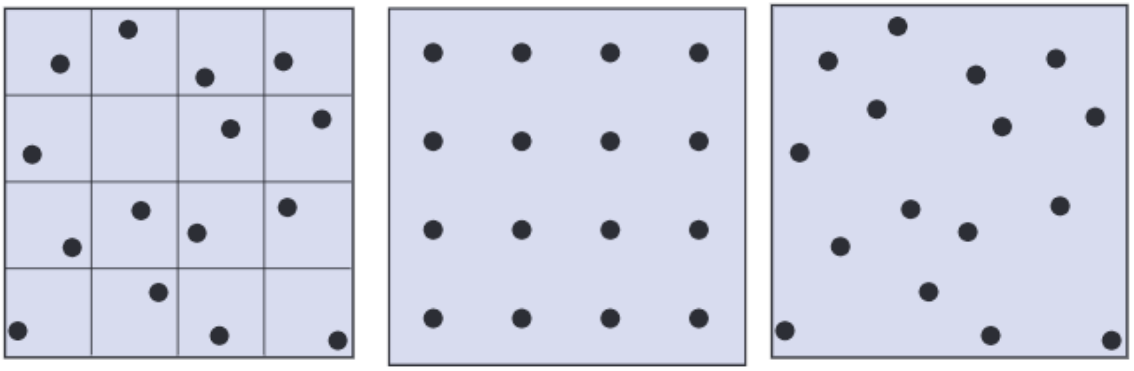
Rendered images

Visual debug

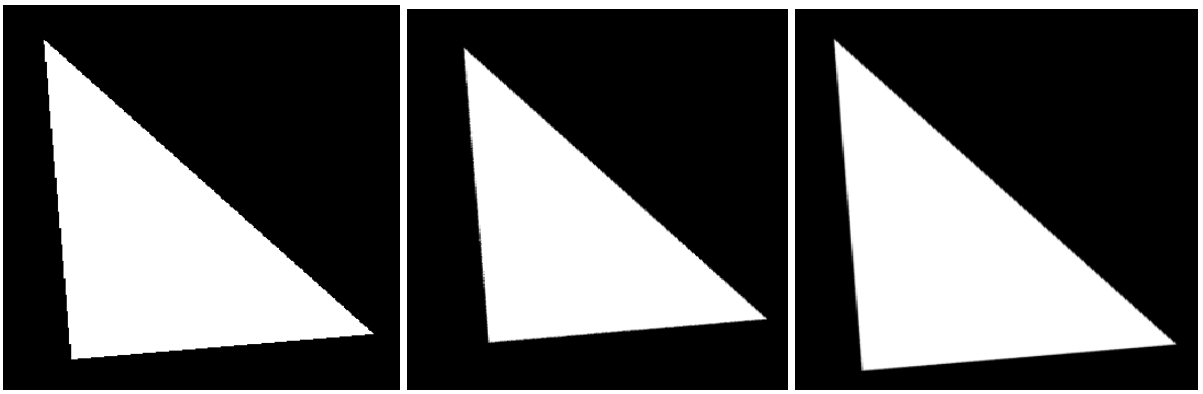
4.7 Multiple rays per pixel

Shooting multiple rays per pixel will make the images look less pixelated by averaging the colour from all rays per pixel. That is exactly what happens in this feature. In render.cpp the pixel colours are calculated by shooting a ray in ‘renderRayTracing()’. For this feature, I added a line of code which calls the ‘calculateColour()’ function in ‘multipleRays.cpp’. In this function the average ray colour is calculated based on the pixel’s location, by shooting multiple sampled rays at the corresponding pixel. There is a slider named ‘ray multiplier’ which determines how many rays we will shoot. The rays are uniformly distributed over the pixel in a grid, and a random offset is added. All rays are parallel for the corresponding pixel. The function returns the average colour of all rays.

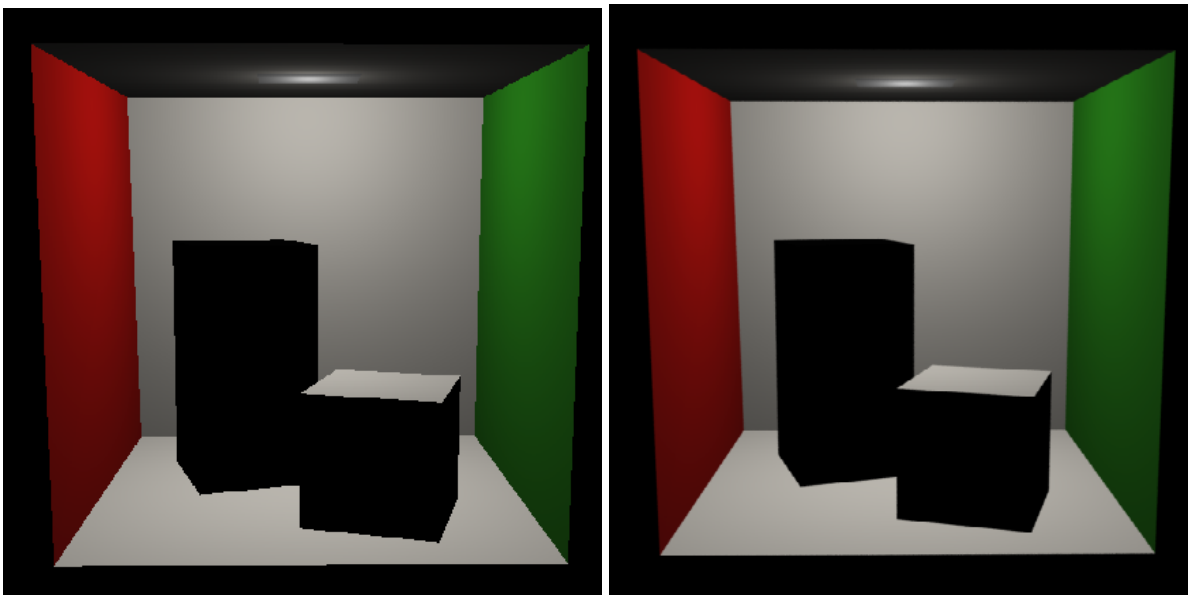
Sampling method: First we determine the amount of rays per pixel, by creating a grid based on the slider. The rays will be uniformly distributed. Then we apply jitter, by adding a random offset to each ray. The following images show a visualisation. More rays means less noise, since the grid will have smaller boxes, so the randomness offset will reduce. The colours of each ray in the pixel’s grid will be averaged. This randomization is added because, as stated in the book : ‘One potential problem with taking samples in a regular pattern within a pixel is that regular artifacts such as moire patterns can arise’. These artifacts turn into noise by adding randomness. Here is a visualisation for a single pixel.



Rendered images



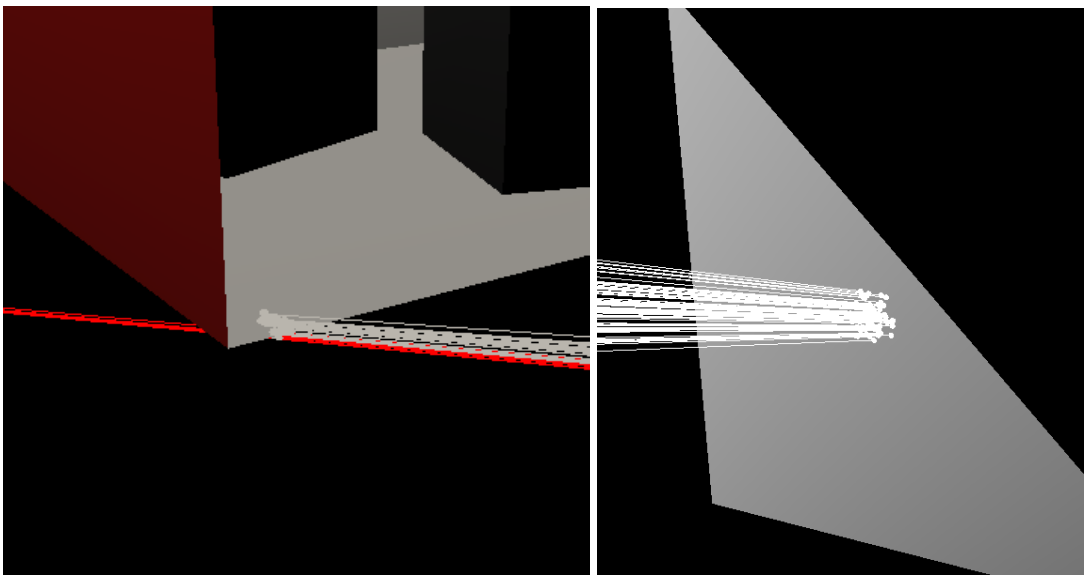
1 ray per pixel (normal) *3 rays per pixel* *10 rays per pixel (smooth edge)*



1 ray per pixel (normal) *10 rays per pixel (notice the smooth edges)*

Visual debug

For the visual debug, I decided to draw all the rays for a certain pixel. The slider indicates the amount of rays. This is done by changing some lines of code in main.cpp. Whenever the user presses ‘R’ to shoot a ray, the function ‘calculateColour’ in ‘multipleRays.cpp’ is called.
We pass 2 extra optional parameters, one indicating if we are in debug mode, and one containing the window size. The latter one is used to calculate the rays based on the mouse position. Since this function is only called when we press R, all rays are stored in a vector so we can draw them every frame. This vector will reset every time when we shoot a new ray. Each iteration in main all rays in this vector will be drawn if the feature is enabled.



Multiple rays per pixel, half of the rays don't collide in the first image, so the colour will be darker since some rays will be coloured black like the background.

4.8 Glossy reflections

Used sources: Book, 13.4.4 on Glossy Reflection

This feature introduces a new look for metallic and non-perfect mirror-like objects, more specifically glossy reflections. In its essence, a glossy object behaves similarly to a perfect mirror, with the only difference being that its reflection needs to be blurred. The strength of this blur is inversely proportional to the shininess of the object (shinier objects are closer to a perfect mirror, while lower shininess is closer to an object with a rough surface). The way these reflections are implemented is by ultimately sampling many rays in a specific way, and then averaging all the color values for these rays. Most of the implementation is done in a separate file called *gloss.cpp,* but the actual application of the function is done next to the recursive ray tracer in *render.cpp*. We are basically applying a randomly generated and continuous version of the traditional Gaussian/Box filter blur.
We first calculate the reflected ray's origin and direction and form a basis with the reflection's direction to get a 'sampling square', from which we can sample all our rays needed for averaging. Then we use a random number generator (initially started with the built-in *rand()*, but then changed it to a *Mersen Twister random-number generator* for its speed and better pseudo-random number algorithms) and pass that through a normal distribution to give us a random displacement that is normally distributed, the origin of the 2D Gaussian distribution being the intersection point between the sample square and the reflection ray, using the two formulas given in the textbook:

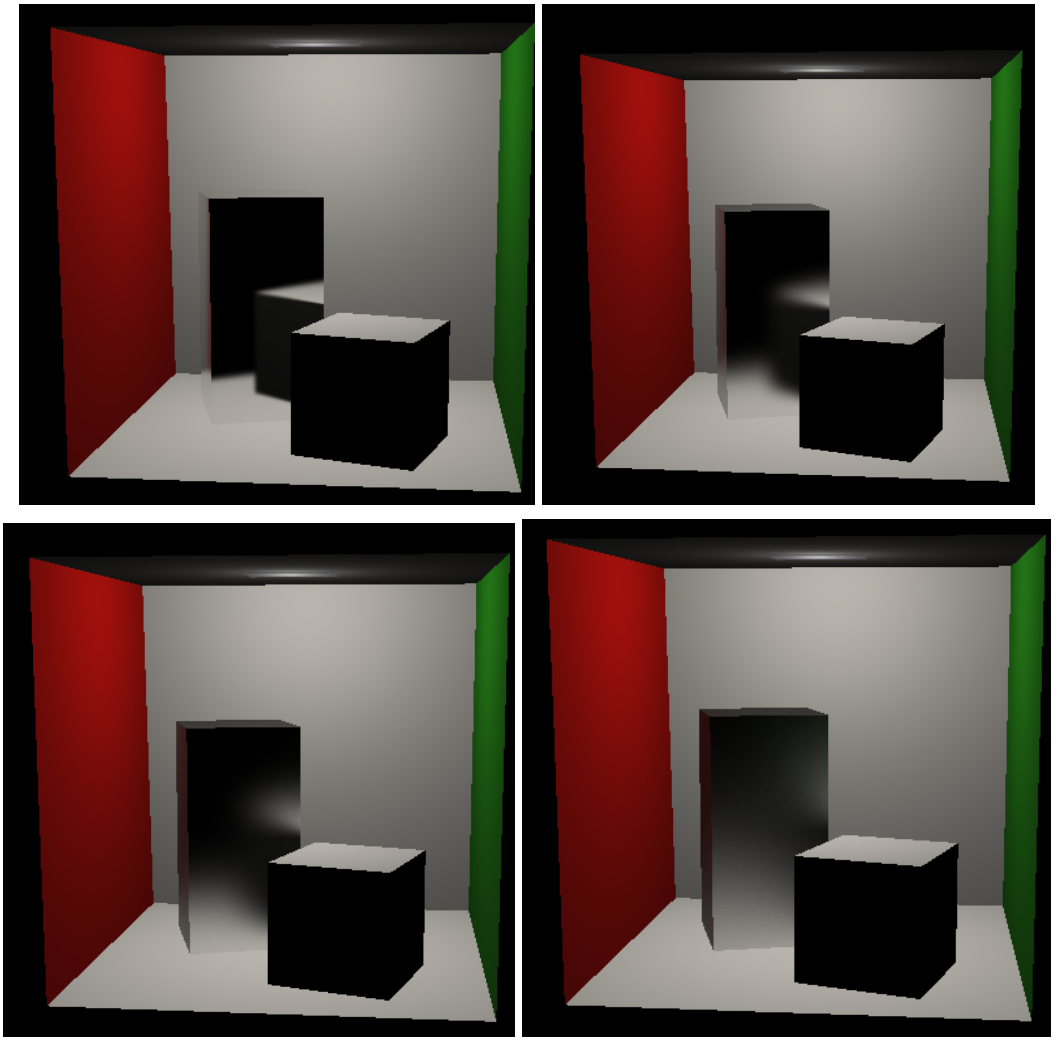
$$u = -\frac{a}{2} + \xi a,$$

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

The reason Gaussian is used instead of the jittered/uniform distribution given in the textbook is mostly because gaussian looked more balanced and overall a better blur in our testing.
We can then scale these displacements by the side of the sample square to get the final perturbation of the ray. The square is distanced **0.3 units away** from the reflection origin through its direction, and the final formula chosen for its size (same thing as blur's strength) is **σ / (3 * shininess)**. The sigma is a general term, whose main purpose is to add more configurability to the feature, so by default it is just 1, but can be changed through the UI to all values between 0 and 5 (where 0 means the term is a perfect mirror). The multiplication by 3 of the denominator also serves a purpose: it is a normalization term for demonstrating the feature in its best *light* (due to the relatively low shininess of the reflective mesh in our specific screen, the reflection is too blurry to be appealing if we do not scale the denominator - for real use scenes the shininess of the model should be configured instead of the hacky scalign done here), but also when paired with a square distance of 0.3 units, it makes it so that the square covers almost *3 sigmas-worth of rays* end-to-origin in all its directions, which covers *~99.7%* of all rays that will be fired from a given position (based on the probability of a value having a magnitude z-score of less than 3), which is more than good enough for our application. The filter_size/number of rays, is also configurable through the UI, the default being 64.

The implementation also handles the edge case of a perturbation making a ray go through the surface it's being reflected from (returns zero vector as per the textbook's recommendation), which will be seen in the second visual debug image as well. One final thing - while conceptually glossy reflections would require the recursive ray tracer to be left on (after all, you are averaging final color values of reflected rays), in our case the implementation of glossy reflections is isolated from the recursive ray tracer for a more independent testing of the features, for easier grading, and to avoid a situation in which one person's implementation is dependent on another.

Rendered images



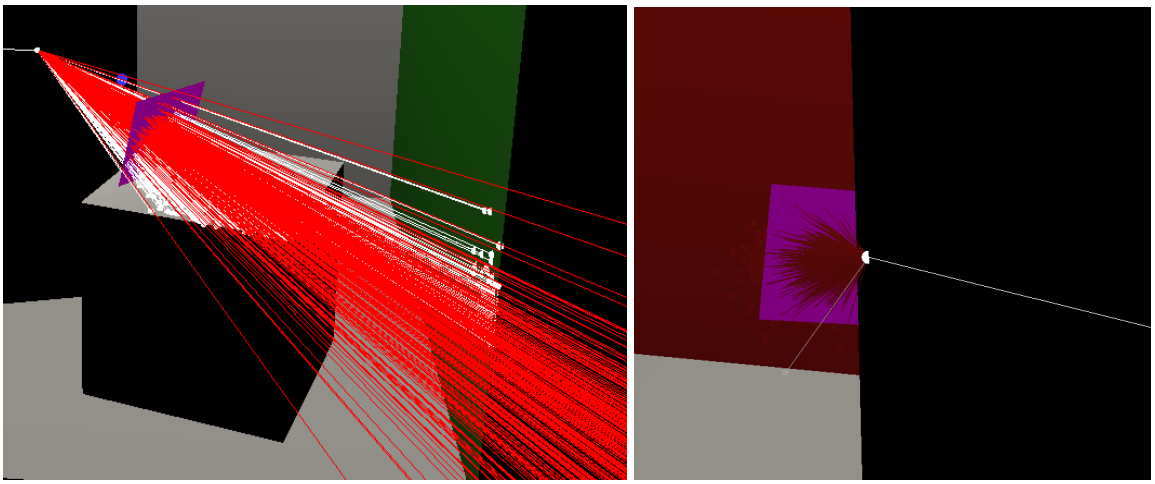
Rendered Cornell Box with sigma = 0.2, 1, 2, and 5, all with the maximum number of 2048 samples. Small values of sigma lead to a smaller sample box, hence all rays are clumped together, until at 0 it is effectively a single ray, like a perfect mirror. At sigma = 5 the cube becomes unseeable from the reflection.

Visual debug

Besides the multiple ray visualization from the recursive ray tracer and the ray coloring from the shading methods, the only unique visual debug to the glossy reflection is the sample

square that scales proportionally to the sigma and inverse-proportionally to the shininess. At the aforementioned distance from the ray.reflection origin, a pair of two purple triangles are drawn to visualize the location and size of the sample square.

UI tweaking of glossy reflections variables

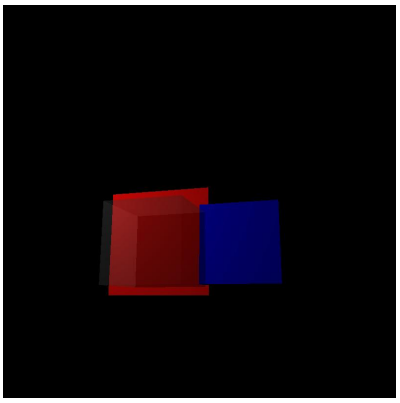


Left: Debugging 1000 rays through the sample square, most of them miss the green wall; Right: Debugging 1000 rays through a sample square, all colored to the red wall. As mentioned, all rays crossing the reflection surface are removed.

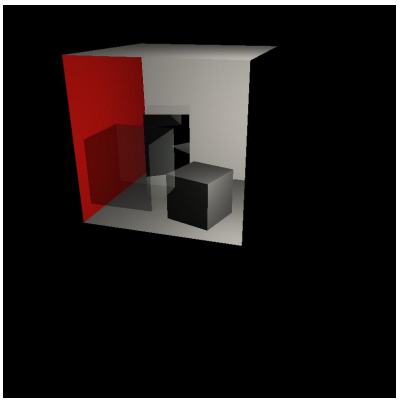
4.9 Transparency

For the transparency feature, we compute the final result inside render.cpp, in the 'getFinalColor()' function. If transparency is enabled, we check whether the intersected ray hit a surface that is transparent. If this is the case, we compute another ray in the same
direction as the previous ray, to check for further intersections, and we use this ray when recursively calling the 'getFinalColor()' function. As for the final colour of the pixel, we compute it using the concept of alpha blending: we take the alpha value(transparency)
of the intersected triangle and multiply it with the current colour, and add it to 1 - alpha value multiplied by the following intersected colour. This stops when we reach a surface that is fully opaque. By blending the colours of multiple surfaces for every computed pixel, we achieve the effect of transparency.

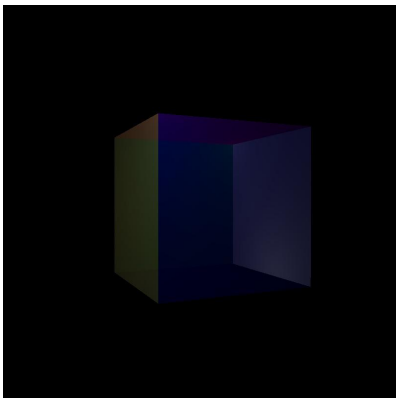
Rendered images



Grey transparent cube with non-transparent red and blue cubes



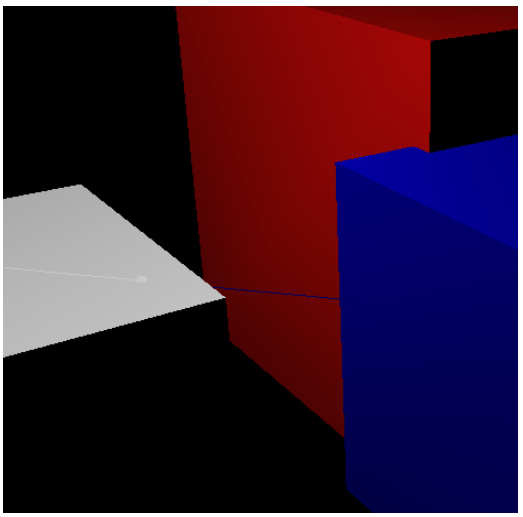
Grey transparent cube in cornell box



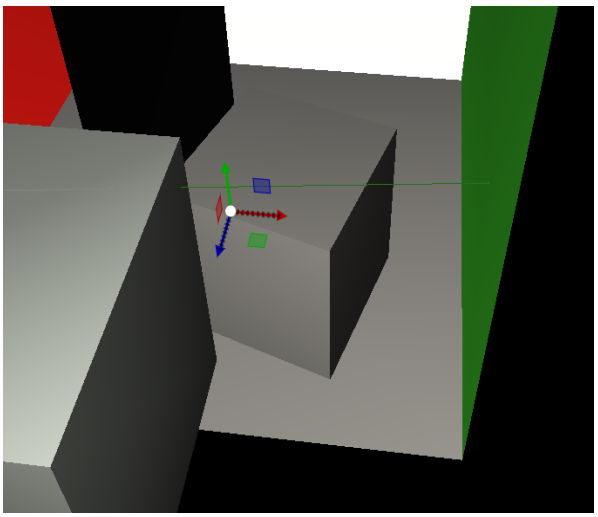
Transparent cube

Visual debug

For the visual debug, we visualise the intermediary colours that are used with alpha blending. Each ray colour represents the intersected mesh colour of the respective ray.



Ray passing through grey transparent cube onto blue opaque cube



Ray passing through grey transparent cube onto green opaque surface

4.10 Depth of field

Used sources: Book, 13.4.3 on Depth of Field

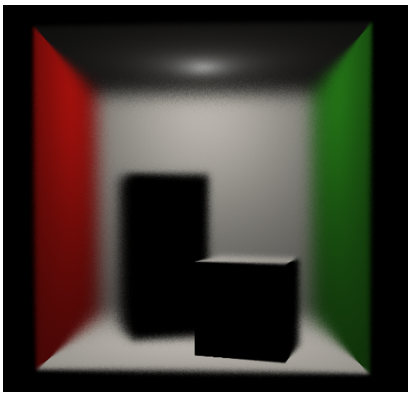
This feature replicates the lens of a camera, where we can adjust the lens to focus on certain objects. The focussed area will become sharper, while the rest becomes blurrier the further it is from the focus point. Our feature has multiple adjustable variables in the UI, which are used to replicate the lens.

If the feature is enabled, we will calculate multiple rays with different angles per pixel, in the direction of the focus point. This angle is determined by the aperture, adjustable in the UI. In *render.cpp*, we generate these rays for each pixel and calculate the average color by calling the *'getEyeFrame()'* function in *dof.cpp*. This means that rays who intersect with an object closer to the focus points will give a less blurry result, since their intersection point is closer together. Objects further from the focus point will be more blurry, since the rays are more spread out over the area. In *'getEyeFrame()'* the amount of samples are uniformly distributed with a random offset, so we use jitter for sampling. This is because we want to evenly spread the rays through the lens.

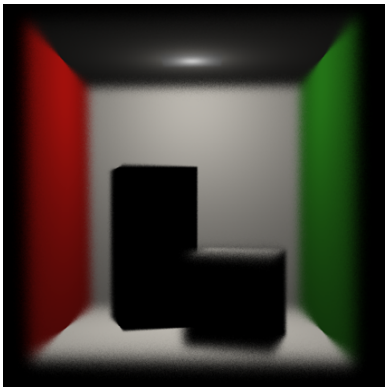
The base model of the depth of field is quite limited, as the blur of an object is linearly proportional to the position of the focal point to the T, meaning that even the slightest displacement from the focal point will result in a blur component. Clearly, however, in real-life the subject does not get blurred no matter the aperture size, so long as it is on the same focal length from the camera at all instances - that is the basis of portrait photography. Therefore, there needs to be an additional term that surrounds the focal point and acts as the *circle of confusion*, where we assume that circles smaller than the offset would be treated as sharpened, instead of slightly blurred with the regular model. Therefore we have added a checkbox to enable or disable this offset box and regulate its size in the UI. It is not on by default to have an easy way of distinguishing its impact on scenes, but also because it's effect is not always desirable, and is dependent on the use case. The size of that box will correspond to the additive volume in which all objects will look sharpened, or in other words, that box defines the Depth of Field (the distance between the two furthest sharp objects), with lower values equaling to a narrower volume, and vice versa. It replaces the traditional formula for calculating the depth of field for extra configurability dimension that is independent of other parameters.

In render.cpp in 'renderRayTracing()' before we shoot multiple rays, we first check if there is an offset enabled. If this is enabled we calculate if the intersection position is inside the area of the offset, and then we only shoot 1 ray. This is because all objects in the offset area will not be blurred. We then calculate the size of the frame from which we will get all our samples for averaging, using an f-Number formula scaled to properly be fit into the smaller models in this project's scenes and for a better visual look. This is made so that the aperture parameter can correspond to real-life apertures of cameras for easy reference, so higher aperture numbers will result in smaller openings, and thus, a sharper image.

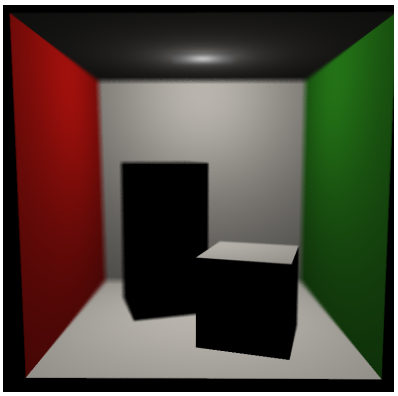
Rendered images



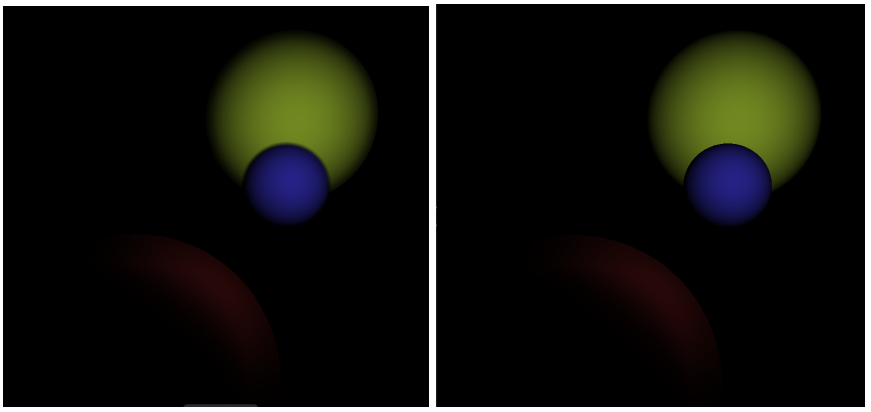
100 samples + 2.0 aperture + 2 focal length



100 samples + 2.0 aperture + 2,5 focal length



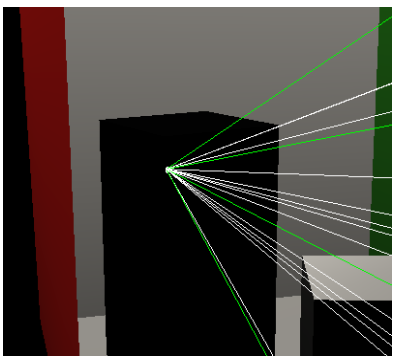
100 samples + 2 aperture +2.5 focal length + 0.04 offset



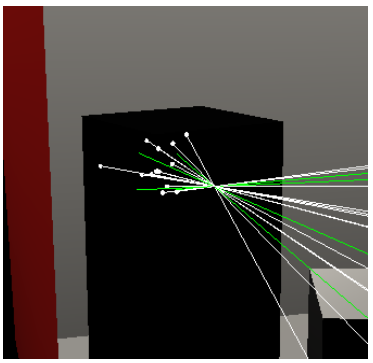
In the left image there is no offset added, in the right image you can clearly see that by adding offset, the blue sphere will become sharper, if you compare the 2 images.

Visual debug

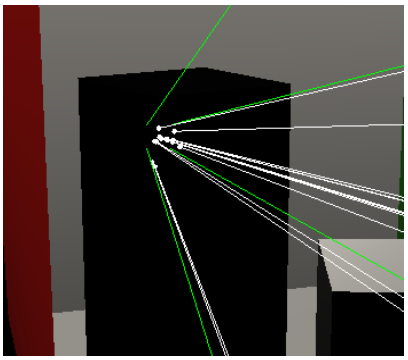
In main.cpp we calculate all rays for a single position if we press the ‘R’ key. The green rays are the rays from the camera plane to the focus point. They are calculated in ‘getEyeFrame()’ in dof.cpp. The white rays are all randomly shot rays in the area between the green rays. All rays are stored in vectors and converge at the focus point. The storing makes up for an easier debug experience, hence why they do not jitter while real-time debugging, but only on ray-traced mode and renderings. The offset visual debug is made by drawing 2 rectangles, and for all parts in the area between the 2 rectangles we only shoot 1 ray, to indicate that this place needs to stay sharpened. There are 2 checkboxes: 1 checks if we should draw the random rays, and 1 to check if we enable and draw the offset rectangles. The 4 sliders are for the amount of samples, the focal length, the aperture and the offset value.



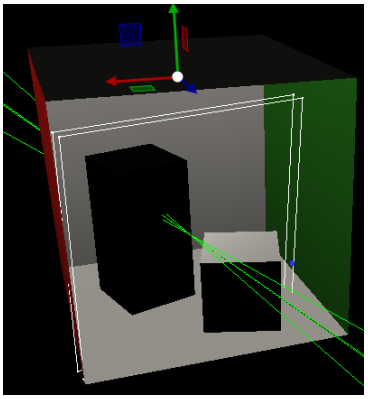
Focal length 2.0



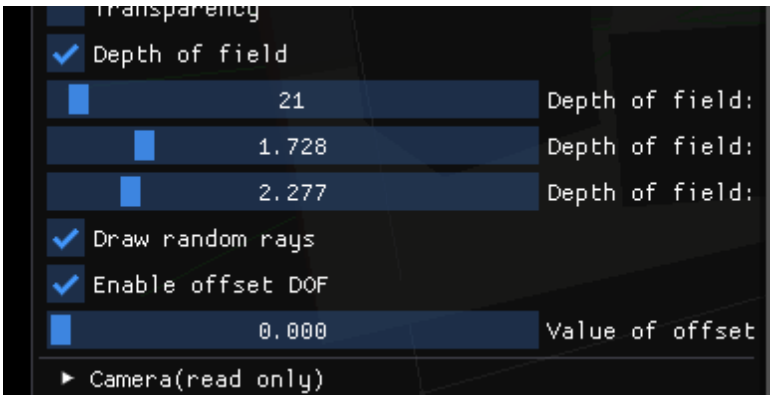
focal length 1.0



focal length 2.5



Random rays disabled + offset enabled



The variables from this feature

6.1 Performance metrics

All tests were performed on a desktop setup with an Intel i7-10700 with base clock and plentiful of RAM to cache all needed structures to avoid bottlenecks. The BVHs were generated with a ‘max_level’ set to 32 for maximum traversal performance.

	Cornell box	Suzanne Monkey	Utah Teapot	Stanford Dragon
Num triangles	32	967	15704	87130
Time to create (Base BVH)	0.040 ms	1.133 ms	22.142 ms	153.70 ms
Time to create (SAH+64 bins)	0.170 ms	16.06 ms	404.46 ms	2970.98 ms
Time to create (SAH, no bins)	0.204 ms	71.09 ms	9049 ms	89.33 s
Time to render (Base BVH)	387.4 ms	2381 ms	3907 ms	79.17 s
Time to render (SAH+64 bins)	163.8 ms	1444 ms	2428 ms	**76.61 s**
Time to render (SAH, no bins)	152.8 ms	1266 ms	***3660 ms***	***77.3 s***
BVH Levels (Base BVH, incl. root)	6	11	15	18
BVH Levels (SAH)	9	14	19	24
Max prims / leaf	1 prim / leaf	1 prim / leaf	1 prim / leaf	1 prim / leaf