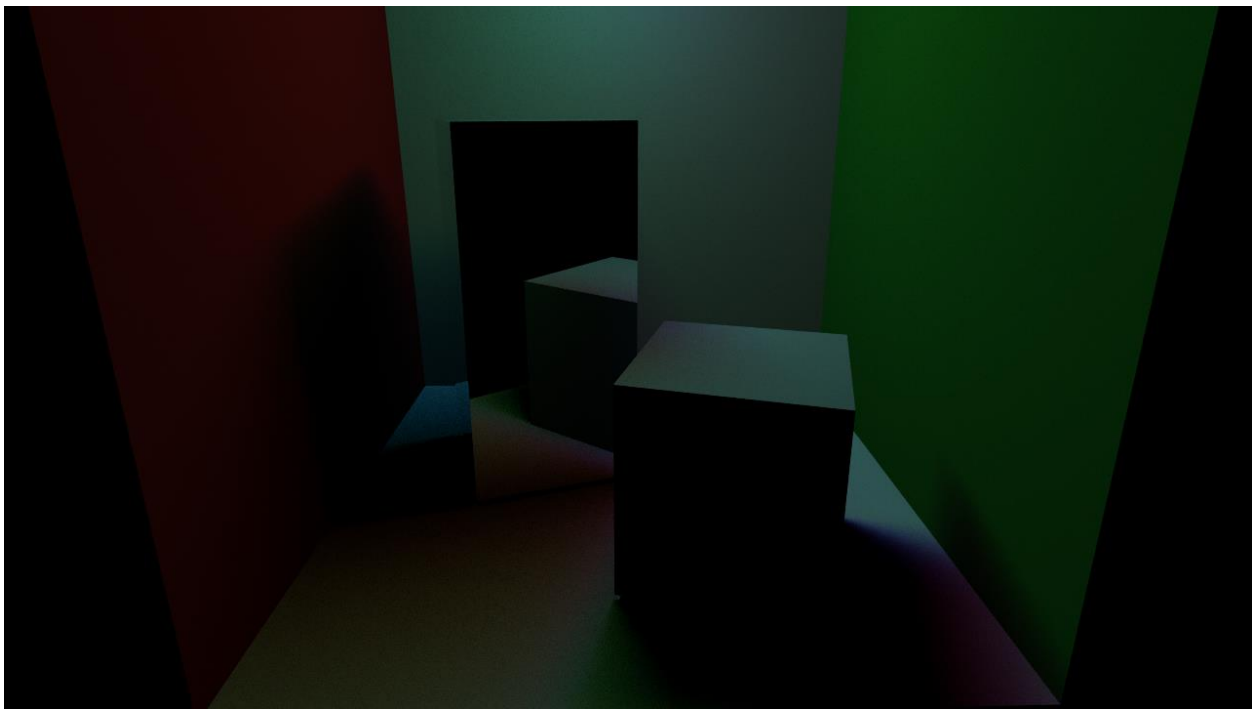
Alexandru Bolfa | Lennart Verstegen | Dorian Erhan

Computer Graphics

REPORT
Final Project



Basic features

Note: In our project we did not use the default library intersection functions, instead, we used our own intersection functions that can be found in intersect.cpp.

Acceleration data-structure generation

Problem addressed: *How to (quickly) calculate the triangle hit by a given ray ? Part 1*

Idea: Divide the scene into box nodes, ordered in a tree so that nodes in a level are half the size of the nodes in the level above it. This way, generally speaking, only one or a few triangles are within a leaf node and the time to calculate which triangle is intersected by the ray gets significantly reduced.

Description: To correctly divide the scene into the nodes, all the triangles in a certain node get split evenly by splitting at the centroid of the median triangle. In 3 dimensions, the direction of this split alternates between the x, y and z axis. The creation of the boxes is done in a recursive manner by calling the `buildAxisAlignedBox()` on the two newly created boxes by splitting the triangles of the current box.

We store the nodes with their box dimensions in an array, where the root is located at 0 and the two childs of each node are stored the indices *currentNodeIndex * 2 + 1* and *currentNodeIndex * 2 + 2*.

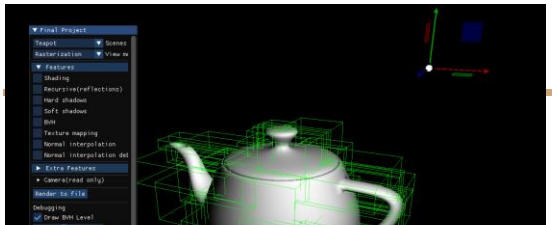
Visual Debug: For the visual debug two methods are implemented: `debugDrawLevel()` and `debugDrawLeaf()`.

In the first method we show all the boxes at a certain level of the tree, where the level can be selected by a slider in the interface. This method can be implemented by making use of the way the nodes are stored in the array. If we want to access level i, then all the nodes from starting index:

Up until the end index (startindex + 2 ^ i) are nodes from that level.

```
int startingIndex = 0;
for (int i = 0; i < level; i++) {
    startingIndex += pow(2, i);
}
```

With the second method only one of the leaf nodes of the tree gets visualised which you can select by adjusting the slider in the interface. When creating the tree, we store each node that is a leaf node in a separate array, so that when asked to show a certain leaf, the number from the slider gets passed to the leaf array and accesses it right away.



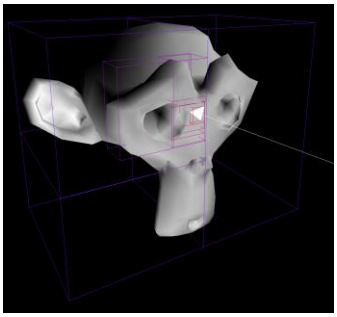
Performance Table:

	Cornell Box	Monkey	Dragon
Time to Render	**24.40 milliseconds**	**28.97 milliseconds**	**38.99 milliseconds**
BVH Levels	**5**	**10**	**17**

Acceleration data-structure traversal

Problem addressed: *How to (quickly) calculate the triangle hit by a given ray ? Part 2*

Description



Idea: Given the data-structure explained *above*, we can make use of split criterion to faster find the expected primitive.

Naïve approach: Traverse every primitive and choose the one closest to ray origin (smallest *ray.t)*.

Slightly faster algorithm: Start from the root node, then recursively traverse the child node closer to the ray origin. We calculate proximity to the ray's origin by getting the distance that the ray needs to travel to reach the node's AABB[1].

Optimal algorithm: Instead of recursively calling "traverse" on every node we keep keep a min-heap of the list of all nodes together with the distance to them. We iteratively get the top of the heap and compare its distance to the closest hit primitive so far. If the distance to the node[2] is bigger than the one to the closest hit primitive we stop the algorithm since every primitive that we can now find is further away than the one we found this far. This is assured by the min-heap.

Implementation: We implement the optimal algorithm explained before in the **BoundingVolumeHierarchy::intersect(...)** method. For the heap, we use **std::priority_queue<T>** with the custom **std::greater<T>** operator to maintain the min-heap order.

Visual Debug:

Idea: In order to better understand the code's behaviour, we need to check the nodes that the ray traverses and the triangle hit.

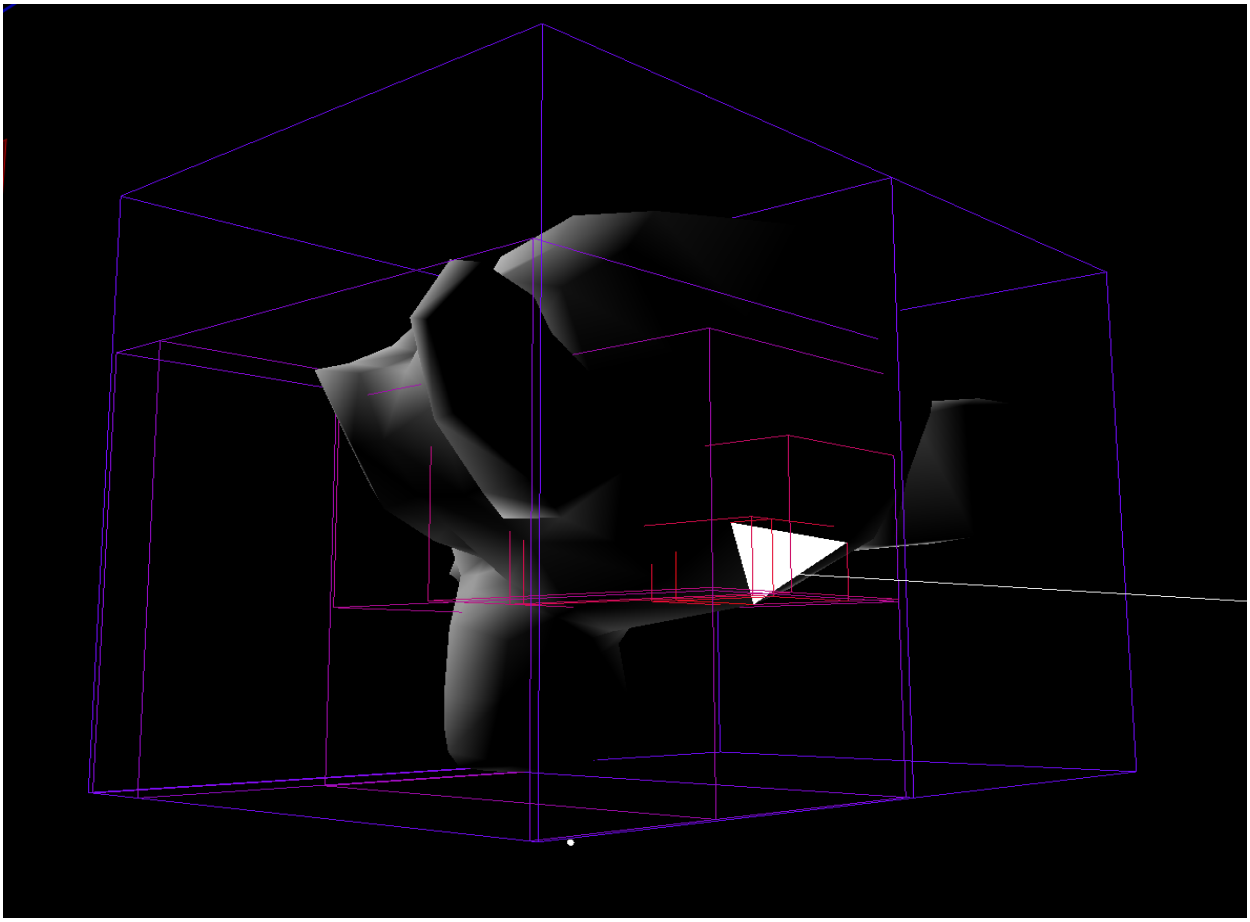
One way to do this is to draw all the *AABBs* 'hit'. We draw them in a colour that progressively becomes redder. This allows us to check the order of the traversal.

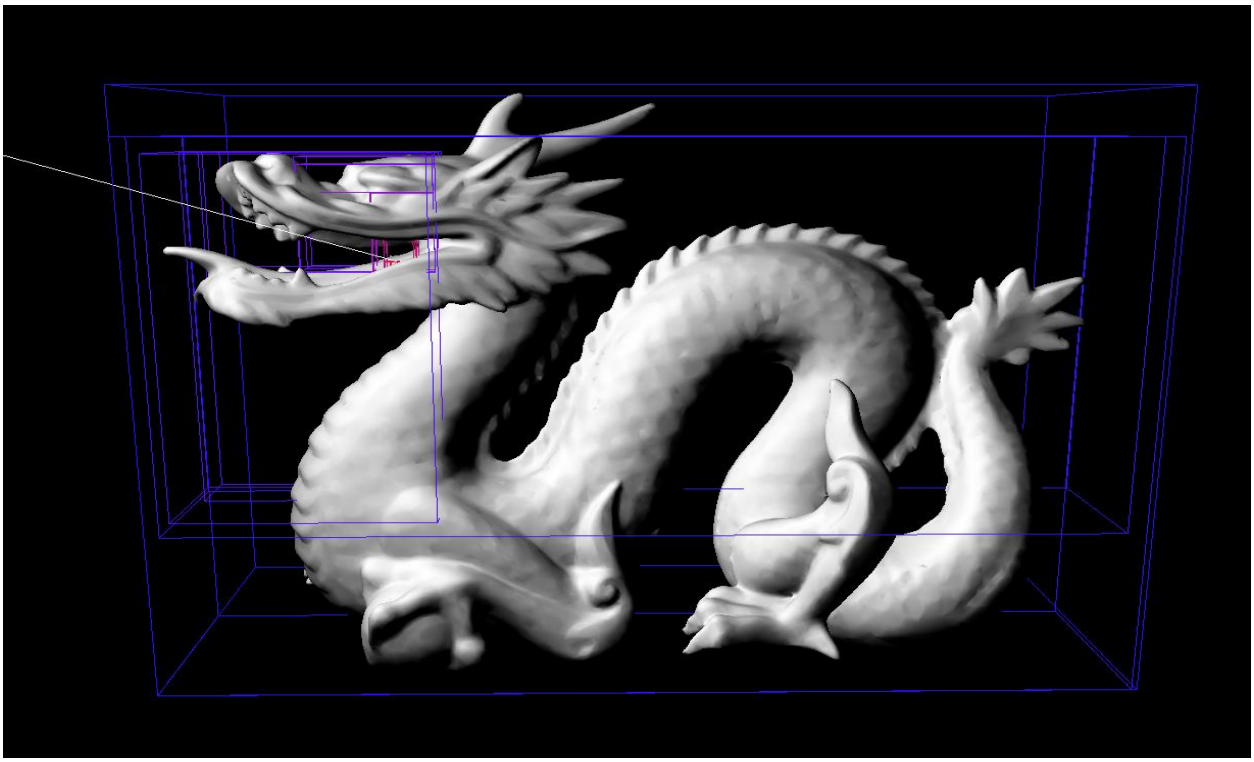
Finally, the triangle hit is coloured to check if the correct primitive has been chosen.

Implementation: After traversing the tree, we check if the *DEBUG flag is set to true*. In that case we draw the *hitTriangle* in the colour white. Then we start going *up* the tree and drawing the corresponding *AABBs*. The first one (that is a leaf node) is coloured red. The colour then gradually tints to blue by a constant factor of *0.05*. The implementation can be found at the end of the **BoundingVolumeHierarchy::intersect(...)** methods.

[1] Axis Aligned Bounding Box(es)
[2] The distance to the AABBs of the node

E.g.





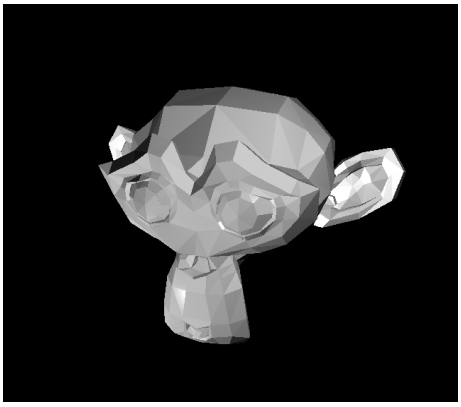
Shading

Description:

`computeShading()` – This method calculates the Phong model of shading for every pixel. It computes all the parameters needed: the surface point (`ray.origin + ray.direction * ray.t`), the vector going from the point to the light source, the view vector, and the reflected vector. For the angles needed, computing the dot product with normalized vectors is equivalent to computing the cosine of the angle. It is also important to check if this dot product is negative, in this case the pixel should not receive any light, thus we set the dot product to 0.

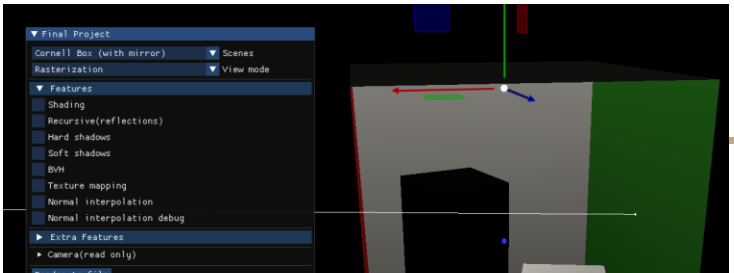
`computeLightContribution()` – For every light source in the scene we compute the shading of the pixel. The final color will be the sum of all shading done for each point light source. The behavior of this method for non-point light sources will be mentioned in the Area Lights segment.

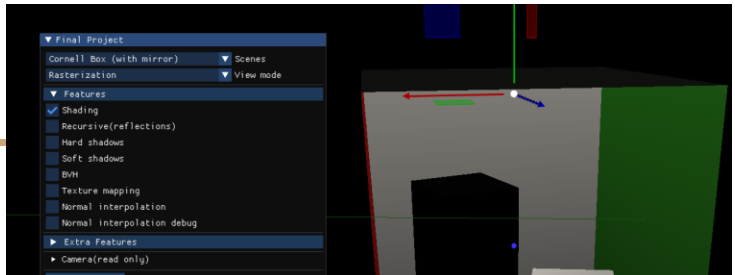
Cornell Box with only shading enabled.



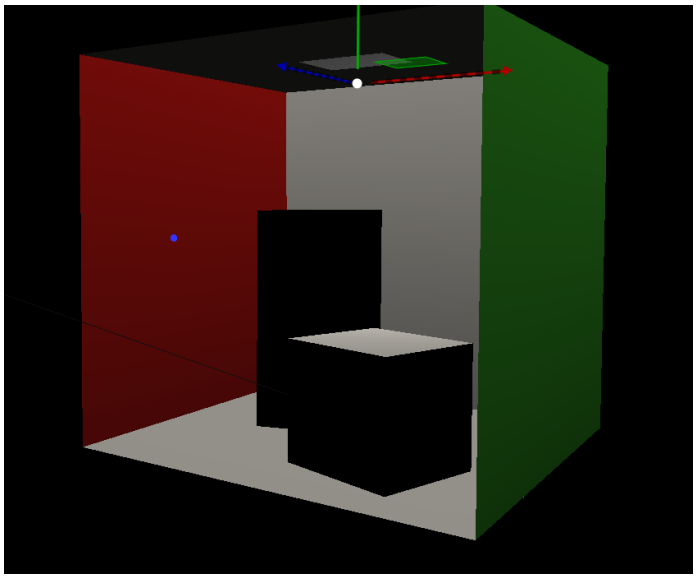
Raytraced monkey with only shading enabled.

Visual Debug:





You can see here the effect of the visual debug. Without the shading flag (left) the ray has the default color of white, and with the visual debug (right) it gets the color of the pixel when shaded.



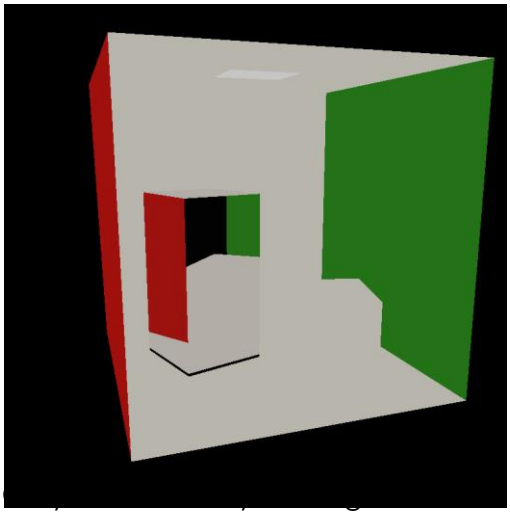
The visual debug for mirror surfaces returns black since it has no diffuse component.

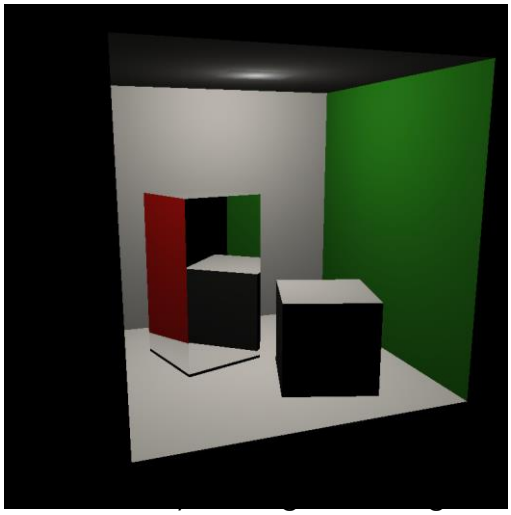
Recursive ray-tracing

Description:

`computeReflectionRay()` - This method takes a ray and a surface point with the normal vector of that point and computes the reflected ray. The direction is set by the formula : (reflectionRay = incomingRay - 2 * dot(incomingRay, surfaceNormal) * surfaceNormal). After that, you set the ray origin to the end point of the incoming ray and an arbitrary ray.t.

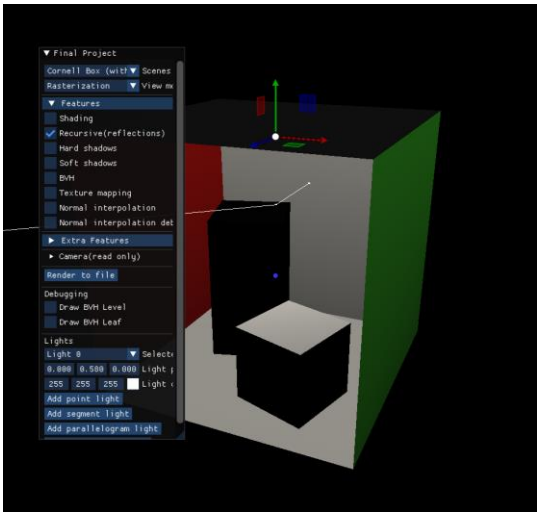
`getFinalColor()` - In this method we implement recursively adding the color together based on their specular component. If the specular component of an object we hit isn't zero, we add the color of the material that the reflected ray intersects with times the specular component to the current color. If the reflected ray doesn't intersect with anything, the get recursively called `getFinalColor()` is set to black and nothing gets added to the already computed color. As parameter we also pass a rayDepth and increase it by 1 every time `getFinalColor()` gets called from within itself, so that we can keep track of how many times the method gets recursively called and stop calling it if it has a depth of 4 or greater. This is needed if you cast a ray in a closed room where every object has a specular component for example, because otherwise the method would get infinitely called and cause a stack overflow.



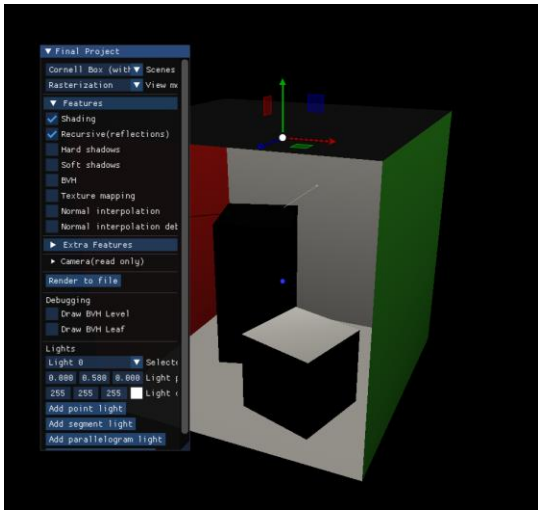


Visual Debug:

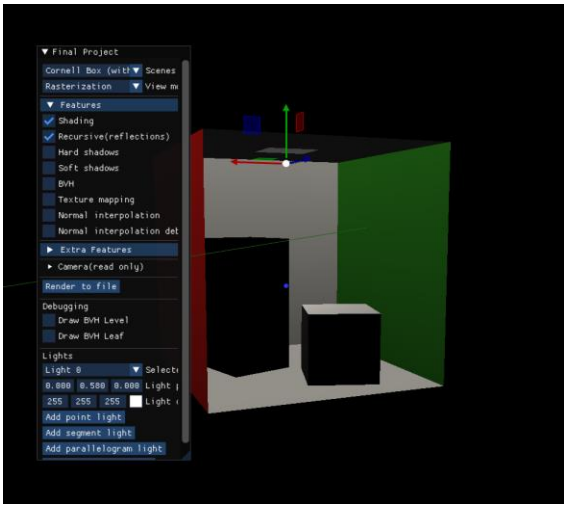
For the visual debug we draw the first ray and each ray that gets recursively called and intersects.



Debug with only recursive ray tracing



Recursive ray tracing + shading



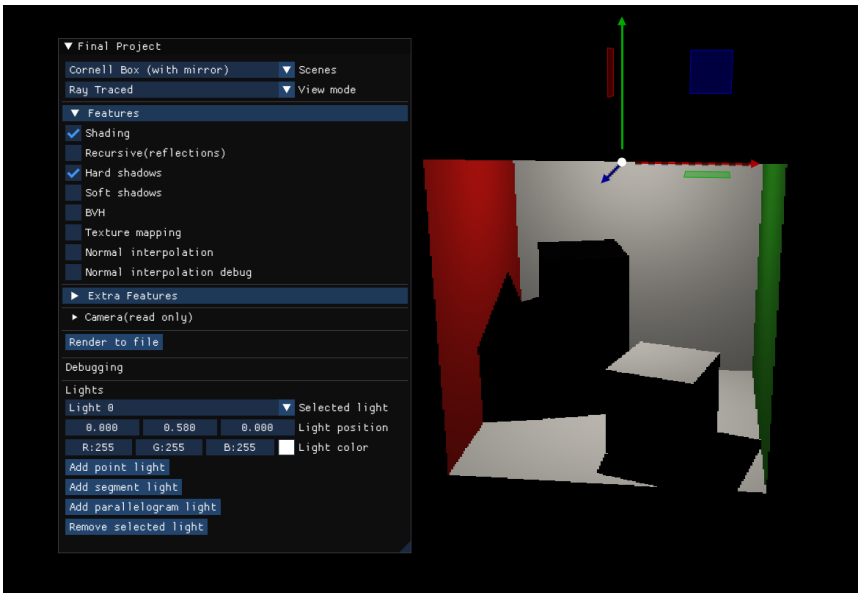
The ray doesn't get reflected when hitting a surface without a specular component

Hard shadows

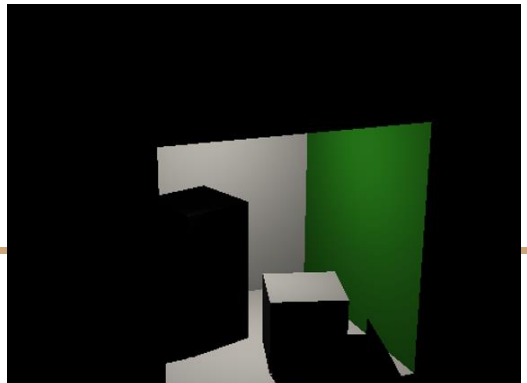
Description:

`testVisibilityLightSample()` - For a certain point light source we draw a shadow ray with origin on the point and direction `light.position - surface-point.` Set the shadowRay.t parameter to 1, since this is the distance to the light source. After that we call `bvh.intersect()` for the shadow ray. This will detect intersections and update the t value again. Finally, check if this new t value is smaller than 1. If it is, this means that there is an object between the point and the light source -> it should be in the shadow and we do not compute shading for it.

Note: a small offset is added to shadowRay.origin to reduce noise.



Cornell box with hard shadows + shading. Note: since the role of hard shadows is to evaluate for each pixel whether they should receive shading or not, to see the effects of hard shadows, the shading flag should also be enabled.

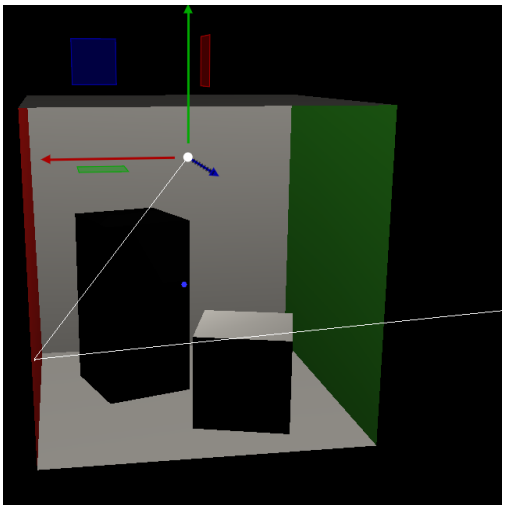


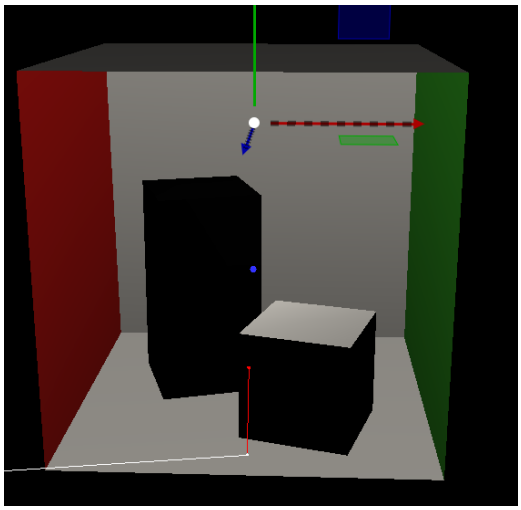
Can also see in this example that the exterior walls are completely black, because no light reaches them. This is done by inverting the normals for triangles facing the opposite direction of the light.

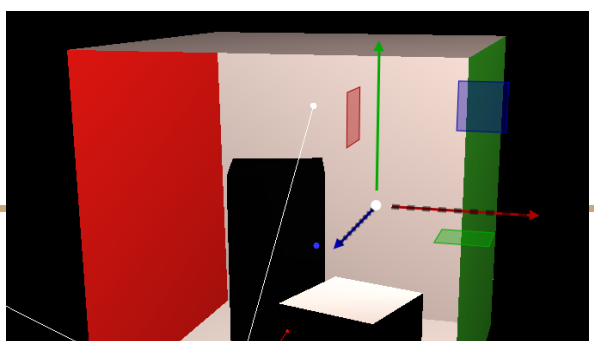
Visual Debug:

For this visual debug multiple auxiliary methods were created. `hardShadowsDebug()` is the general method that utilizes `getPointLights()` to get all the point lights in the scene and `drawShadowRay()` to draw all the shadow rays.

For points seen by the light (left case) the shadow ray is white and for cases where the light is occluded (right case) the shadow ray will be red.







Visual debug working for multiple light sources, for one light source the point is in the shadow and for the other it is not.

Area lights

Description:

For area lights a slider is added below the checkbox to change the number of samples taken for each area light. The method `getNumSamples()` in light.cpp is an auxiliary method added to retrieve this variable from main.

`computeLightContribution()` - For segment/parallelogram lights each sample we retrieve gets treated as a point light and we call the `testVisibilityLightSample()`, previously defined in hard shadows, to test its visibility and see if we compute shading for that particular sample. In the end all the shading values are averaged by adding them up and dividing by the number of samples.

Sampling:

Random uniform sampling was used in these methods, making use of random variables that follow an uniform distribution from 0 to 1. This method of sampling is used because it

is very simple and efficient, besides, being random is favorable because it won't produce patterns (unlike regular sampling).

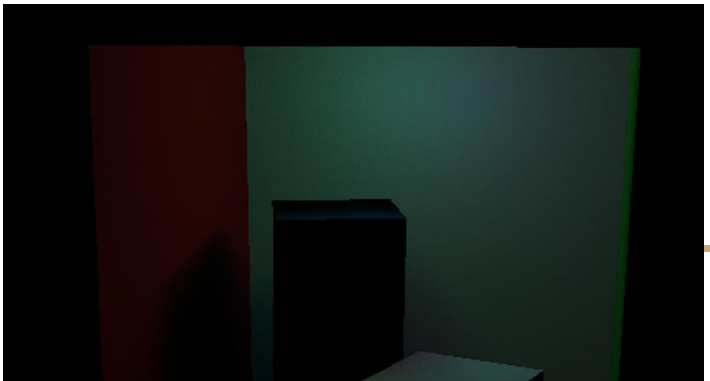
`sampleSegmentLight()` – Here one random variable (x1) is used to compute the position of the sample. First the endpoints are subtracted (`segmentLight.endpoint0 - segmentLight.endpoint1`), then we scale this vector with the random variable. Finally, we have to add the result to the first point, in order to align it with the segment light vector (`(segmentVector * x1) + segmentLight.endpoint1`). To compute the color simple linear interpolation is used - `x1 * segmentLight.color0 + (1.0f - x1) * segmentLight.color1.`

Source: [javascript - How can I generate a random point on a line segment? - Stack Overflow]

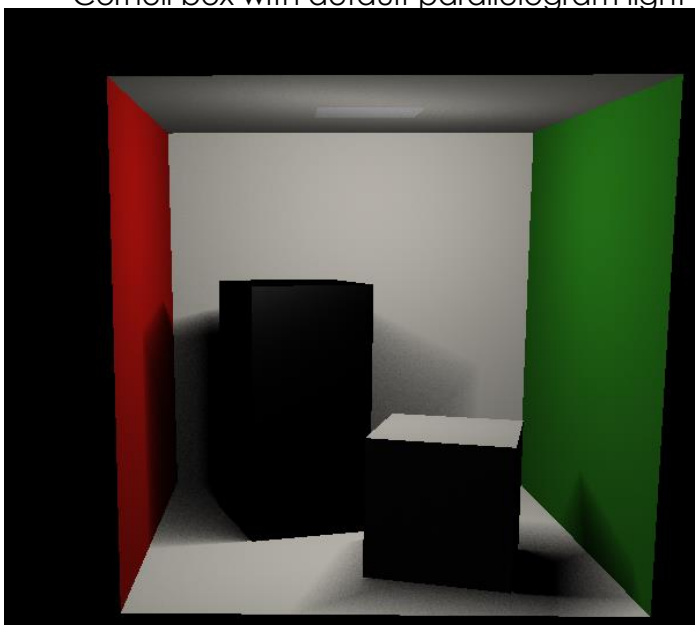
`sampleParallelogramLight()` - Here two random variables are used (x1 and x2) to compute the position of the sample. The formula to calculate the position is directly taken from the book:

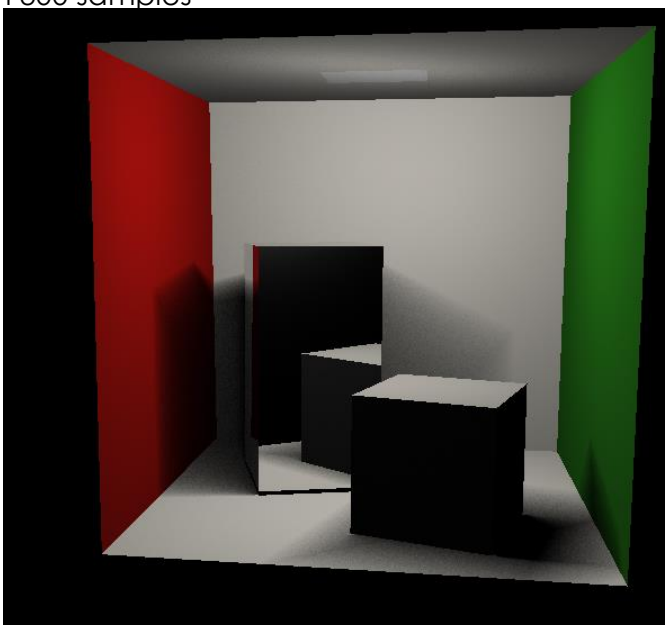
$$\mathbf{r} = \mathbf{c} + \xi_1 \mathbf{a} + \xi_2 \mathbf{b},$$

With c being one of the vertices of the parallelogram, a and b being the edges and xl (the greek letter) representing the 2 different random variables. The color is calculated using bilinear interpolation.



Cornell box with default parallelogram light with 300 samples

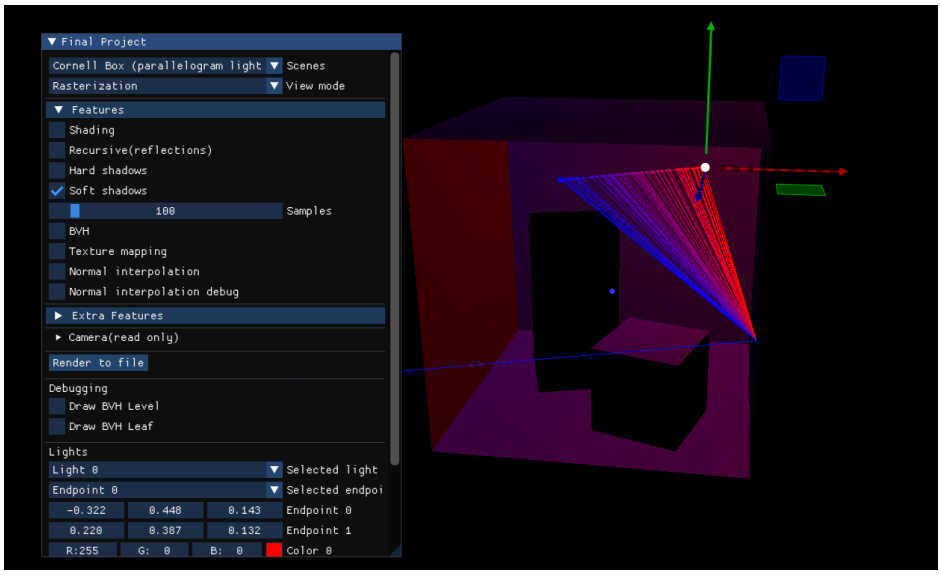




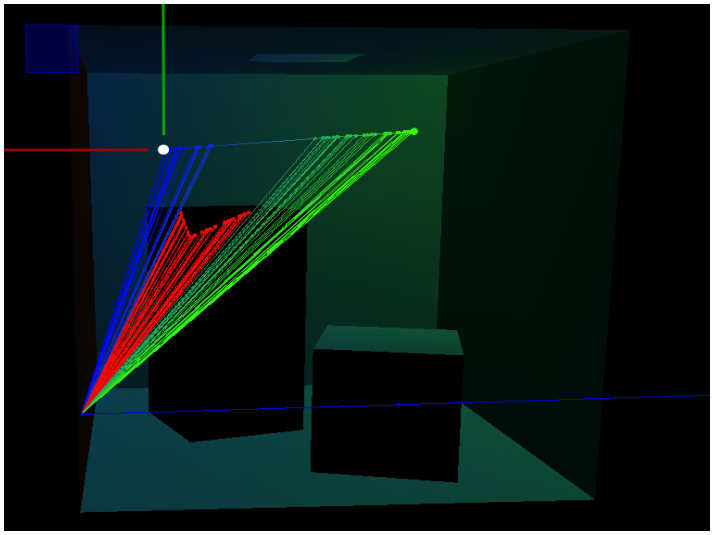
Cornell box with all white segment light source with 300 samples.

Visual Debug:

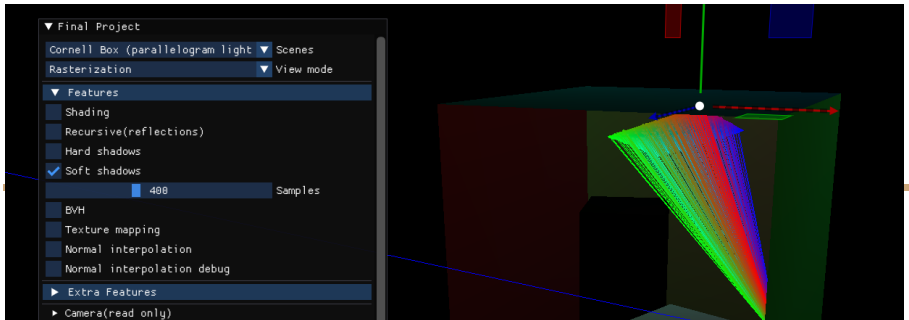
For this visual debug multiple auxiliary methods were created. `softShadowsDebug()` is the general method that utilizes `getAreaLights()` to get all the area lights (segment and parallelogram)in the scene and `drawShadowRay()` to draw all the shadow rays.



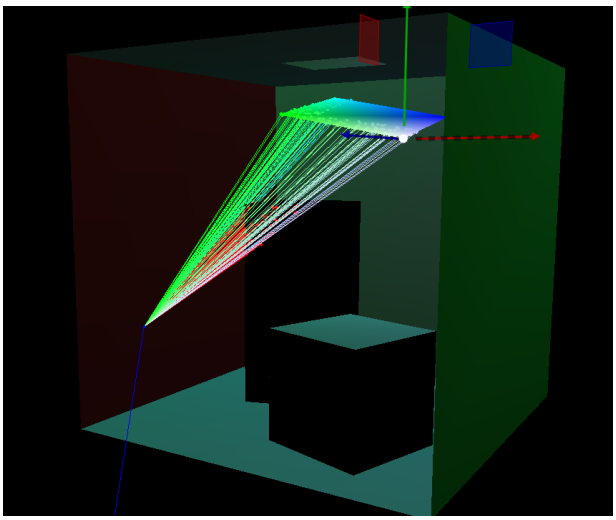
Visual debug for a fully visible segment light with 2 different colors in the endpoints with 100 samples.

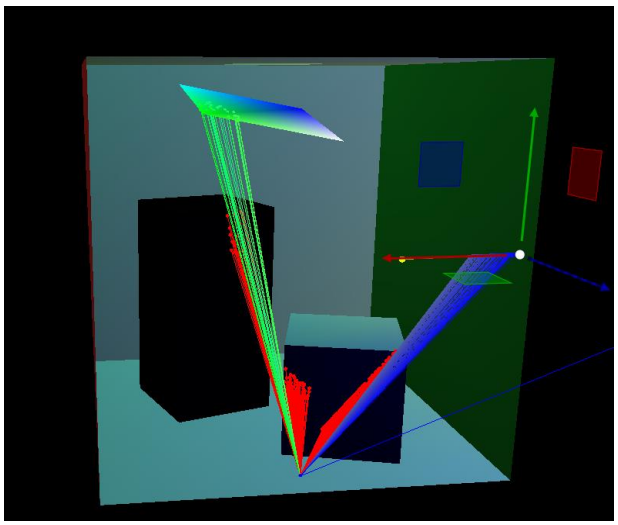


Visual debug for a point partially occluded from a segment light. The red colored rays represent light samples that are not visible from the point.



Visual debug for a fully visible parallelogram with 4 different colors in each of the endpoints (400 samples).





Visual debug for a partially occluded parallelogram light (left) and visual debug for a scene with both types of area lights.

Note: if shading is enabled for a scene with area lights and soft shadows isn't then `testVisibilityLightSample()` will not be called making all pixels be completely shaded, since we don't check for shadows.

Barycentric coordinates for normal interpolation

Description:

`computeBarycentricCoord()` - This function computes the barycentric coordinates of a point given the point and the three vertices of the triangle. This is implemented with same method described in the book with the following formulas:

$$\alpha = \frac{\mathbf{n} \cdot \mathbf{n}_a}{\|\mathbf{n}\|^2},$$

$$\beta = \frac{\mathbf{n} \cdot \mathbf{n}_b}{\|\mathbf{n}\|^2},$$

$$\gamma = \frac{\mathbf{n} \cdot \mathbf{n}_c}{\|\mathbf{n}\|^2},$$

$$\mathbf{n}_a = (\mathbf{c} - \mathbf{b}) \times (\mathbf{p} - \mathbf{b}),$$
$$\mathbf{n}_b = (\mathbf{a} - \mathbf{c}) \times (\mathbf{p} - \mathbf{c}),$$
$$\mathbf{n}_c = (\mathbf{b} - \mathbf{a}) \times (\mathbf{p} - \mathbf{a}).$$

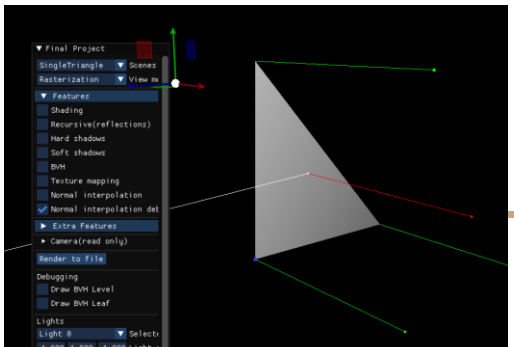
Where a, b, c are the three vertices, p is the point and n is the triangle normal.

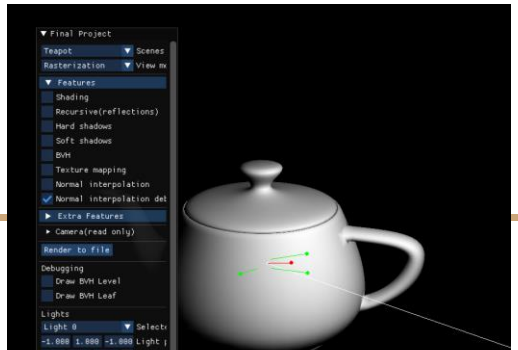
`interpolateNormal()` - In this function the interpolated normal is calculated from the three given vertex normals and barycentric coordinates. To calculate the interpolated normal, we use this formula:

$$\mathbf{n} = \alpha\mathbf{n}_0 + \beta\mathbf{n}_1 + \gamma\mathbf{n}_2.$$

Afterwards we normalise the output vector to make it a true normal.

Visual debug: For the visual debug all the vertex normals and the interpolated normal are drawn. We set the origin of the interpolated normal ray at the intersection point in the triangle.



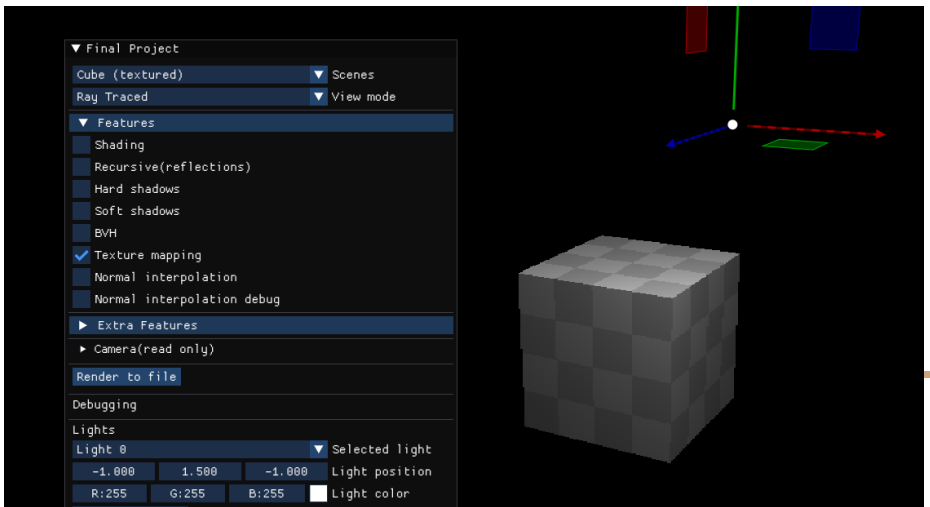


Textures

Description:

`interpolateTexCoords()` – In `bvh.intersect()` after detecting an intersection, we compute the barycentric coordinates of the pixel and also extract the texture coordinates from all 3 vertices of the intersect triangle. Finally, we pass everything into this method to perform a weighted sum, with the weights being the barycentric coordinates, to calculate the texture coordinate.

`acquireTexel()` – Now that we have the texture coordinates, when computing shading we check if the mesh has textures, and if it does acquire texel will be called. Since the texture coordinates are between 0 and 1 we have to multiply it with the image width/height. Also, since the image is inverted vertically when loaded we have to take y = 1 - y. The pixels in the texture image are stored in an array, so to get the texel(x,y) the formula given in the lecture is used - `y * image.width + x`. This texel will now replace kd when calculating shading.

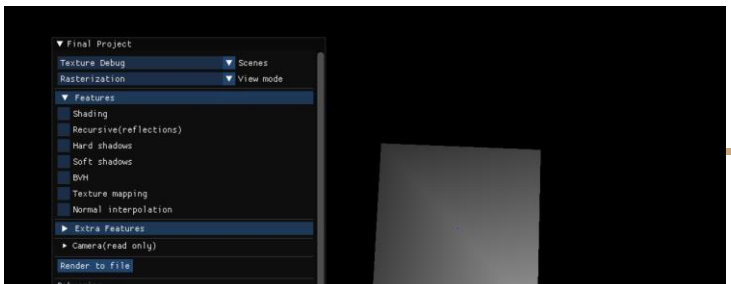


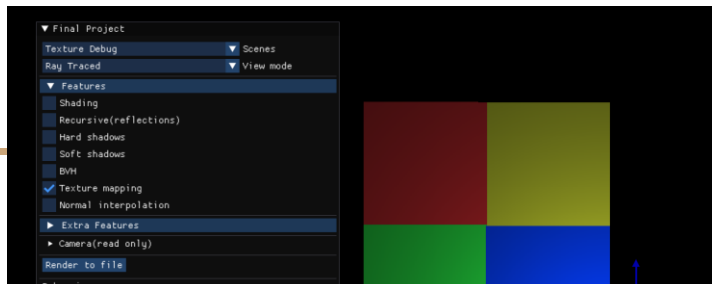
Raytraced cubed with textures applied to it.

Raytraced cubed with textures applied to it.

Visual Debug:

For this visual debug an extra scene called “Texture Debug“ is added. This scene contains a quad with a simple 2x2 texture.





Quad without any texture applied.

Quad with a 2x2 texture applied.

Extra features

Transparency

Description:

The main idea of transparency is combining the color of an object with another object that is behind it, if the first object is transparent.

We know an object in a scene is transparent by looking at its material.transparency value. In `getFinalColor()` we check if the transparency value of the current object isn't one. If that is the case, then the object is transparent and should have its color combined with the color of what is behind the object. We have implemented this functionality recursively so that there can be multiple transparent objects behind each other.

We do the combining of colors with the following formula, where c is the final color, alfa the transparency value of the current object and cf, cb, the color of the current object and the object behind it respectively.

$$\mathbf{c} = \alpha\mathbf{c}_f + (1-\alpha)\mathbf{c}_b.$$

To make the recursive part work, we create a new ray each time we intersect with a transparent object, with the origin of the new ray located where the previous one ended and pass that into the recursively called `getFinalColor()`.

Visual debug:

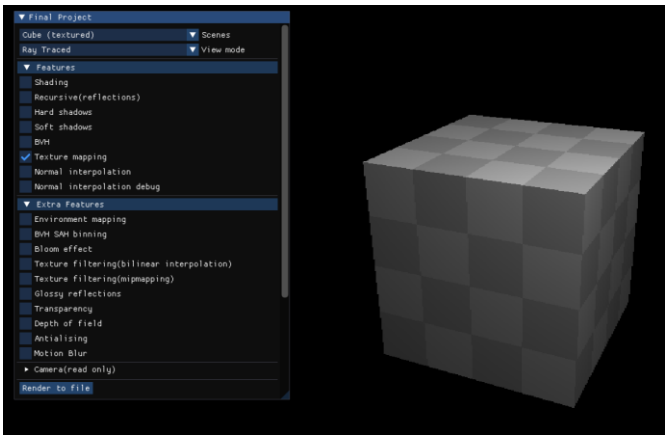
Texture filtering: Bilinear interpolation

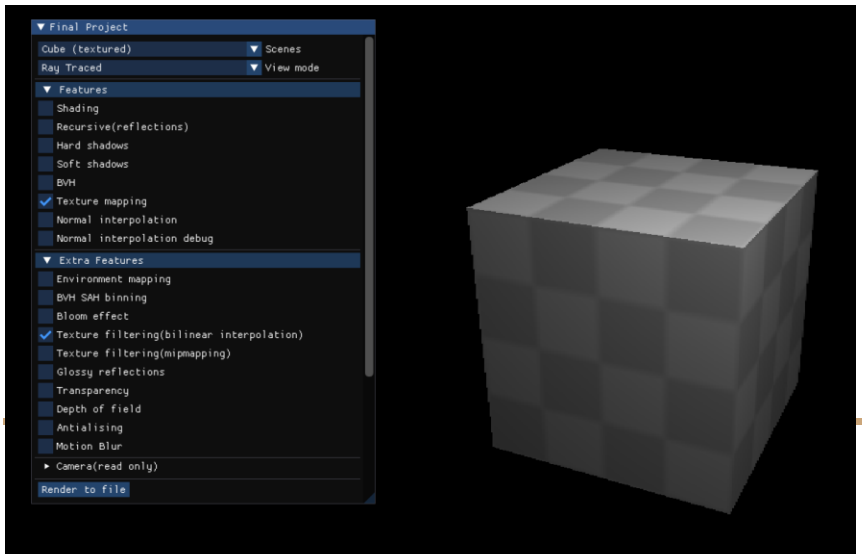
Description:

Bilinear interpolation is a common technique used in texture mapping to counter oversampling cases. All code relative to this feature is in the `acquireTexel()` method in texture.cpp, below the corresponding flag.

The implementation of this method is directly based from the course book "Fundamentals of Computer Graphics by Steve Marschner and Peter Shirley" and the snippet of code referenced is on page 264.

Like in texture mapping, we have to start by multiplying the texture coordinates between 0 and 1 by the width/height of the texture (this originates the variables `u_p` and `v_p`) to get the j and i values necessary to get the correct texel index. However, now this texel index is initialized as a float because the decimal part (represented by `a_u/a_v`) will be used as the weights (as alpha and beta) in the interpolation. `B_u` and `b_v` represent `1-a_u` and `1-a_v.` Now, since we want to interpolate with the 4 nearest texels from the texture coordinates we take the floor of `u_p` and `v_p` to give us the (i,j) values for the texel. Also we need the neighboring texels, thus `iu1` represents i+1 and `iv1` represents j+1. Finally we apply bilinear interpolation using `a_u` and `a_v` as weights, and since the pixels are stored in an array we have to transform (i,j) to a valid index, thus the texels used will be: `image.pixels[j * image.width + i]/image.pixels[(j+1) * image.width + i]/image.pixels[j * image.width + (i+1)]/image.pixels[(j +1) * image.width + (i+1)]`



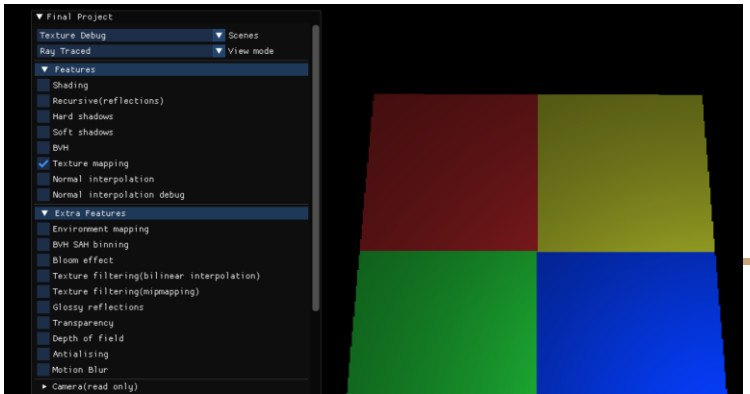


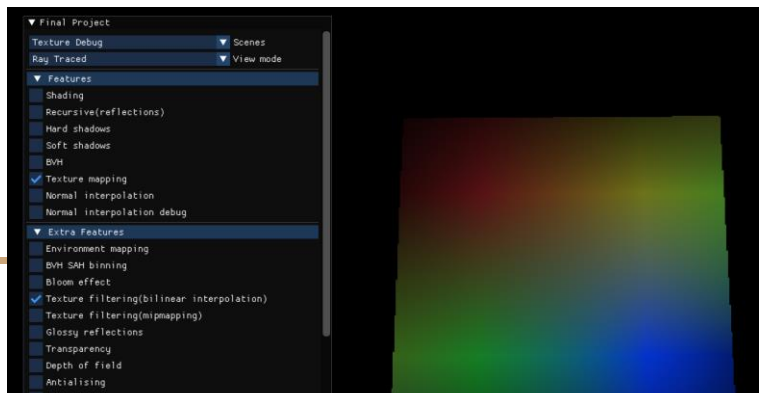
We can see the difference between the cube textured normally (the image above) and the cube textured using bilinear interpolation (image below).

Normally textured teapot (left) and bilinearly interpolated textured teapot (right).

Visual debug:

For the visual debug the same custom scene from the texture mapping debug, found in "Texture Debug", is used. This custom scene was purposefully colored with 4 different colors in each pixel for the bilinear interpolation effect to be very noticeable.





It can be seen here very well the effect that bilinear interpolation does in this particular texture.

Bloom filter

Description:

For the bloom filter 2 additional sliders were added to control the size of the box filter as well the value of threshold for the pixel brightness.

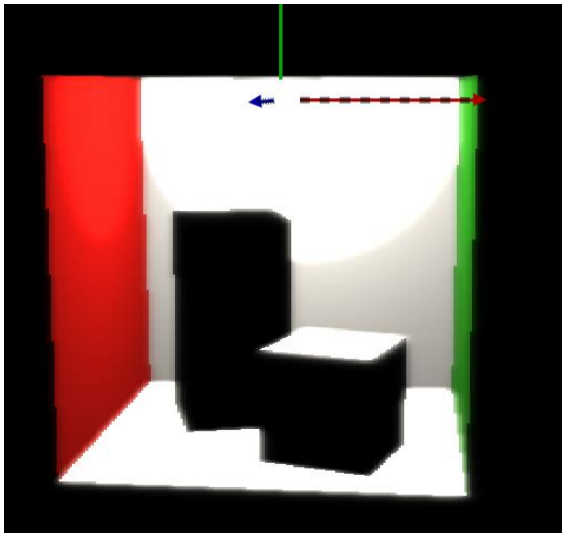
The main implementation of the bloom filter is on main.cpp (starting on line 423), since this is a post processing filter that happens after the initial ray tracing is done. This is because we need to apply a threshold to the image, by only keeping bright pixels, above this threshold, and setting to black all other pixels. There are also 2 auxiliary methods in screen.cpp explained below:

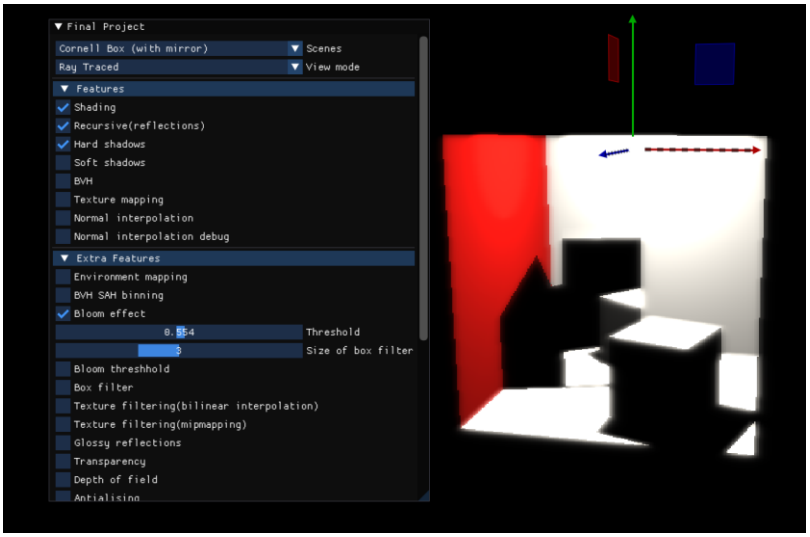
`Screen::setPixel(int index, const glm::vec3& color)` - Has the exact same functionality as the default setPixel method, however this default method takes x and y (the coordinates of the pixel) as the parameters, which was not convenient for my specific implementation. Because of this, my method just takes the index of the pixel in the `m_textureData` array, so that I can more easily set its color.

`Screen::boxFilter(glm::vec3 pixel, int index, int size)` - This method applies the blur effect needed for the bloom effect. For my implementation I chose the box filter,

because it's effective and I am also very comfortable with it, since it was directly discussed in the lectures. The size of the box filter can be changed with the slider, the minimum size being 1, which will apply a 3x3 box filter, meaning that we take a 3x3 box around every pixel and average its color with the colors of the pixels in this box. Pixels outside the array boundary are set to 0.

After the threshold, the value for every pixel calculated by the box filter is added to the original colors.

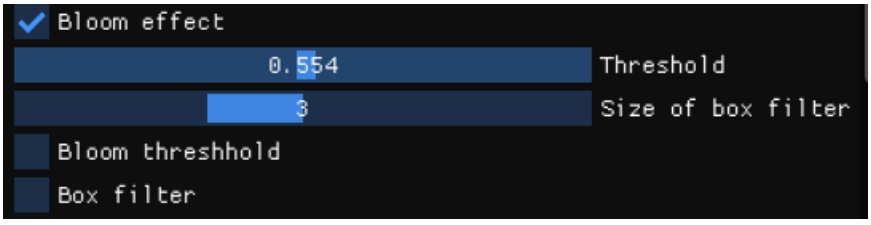




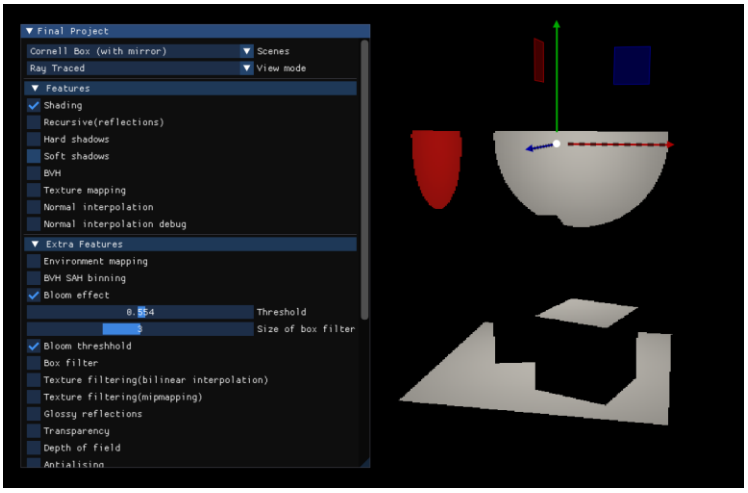
We can see here the bloom filter effect in the Cornell box with a point light source.

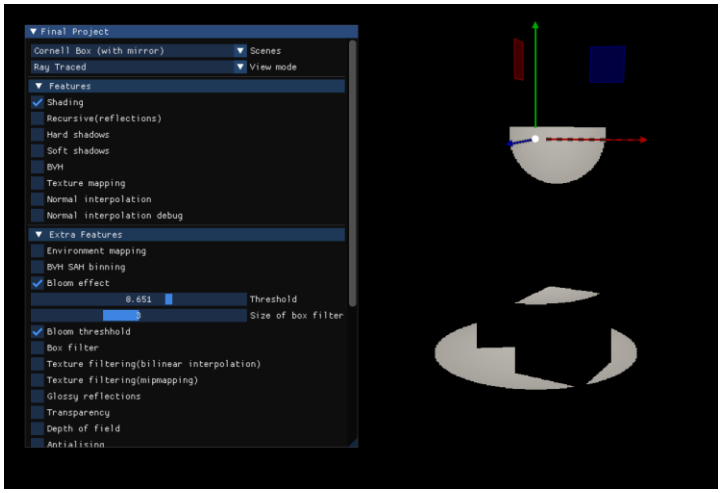
Visual debug:

For the visual debug 2 extra checkboxes were added when clicking the bloom filter checkbox.

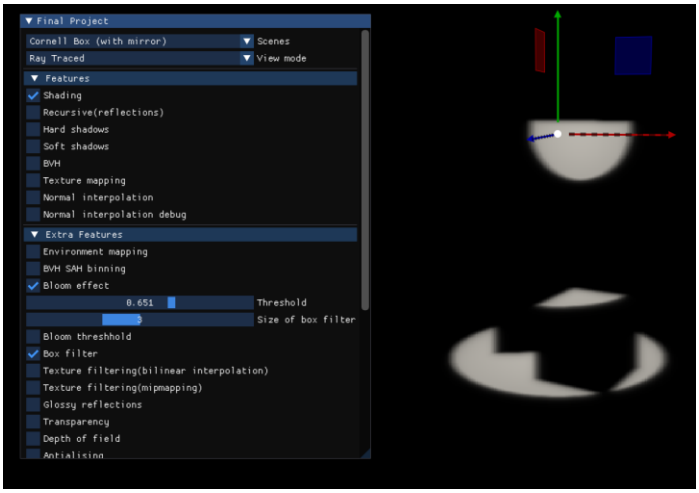


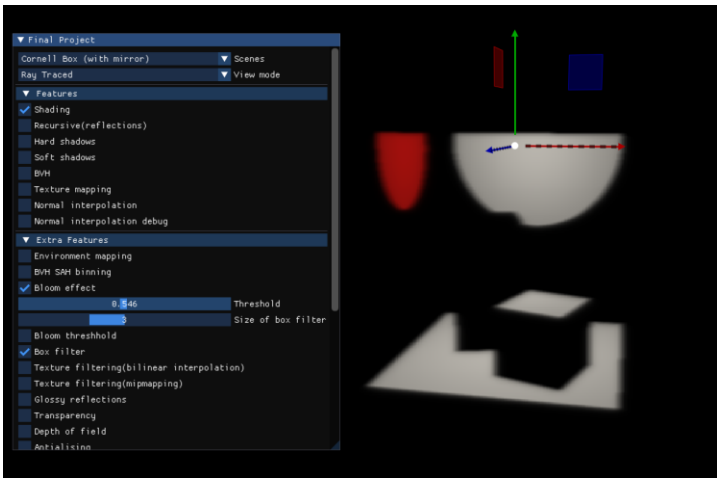
The bloom threshold filter displays the image after the threshold has been applied. This flag in combination with the slider makes it very easy to see the pixels that remain after the threshold and which ones are set to black.





Then the box filter checkbox lets you see the image after the blur is applied.

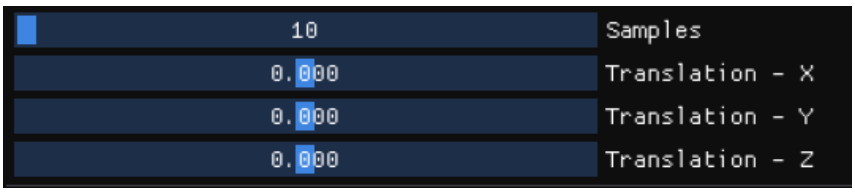




Motion blur

Description:

For motion blur 4 new sliders were added to the UI. With the slider we can control: the number of samples taken and the translation that the mesh does in any of the 3 directions.

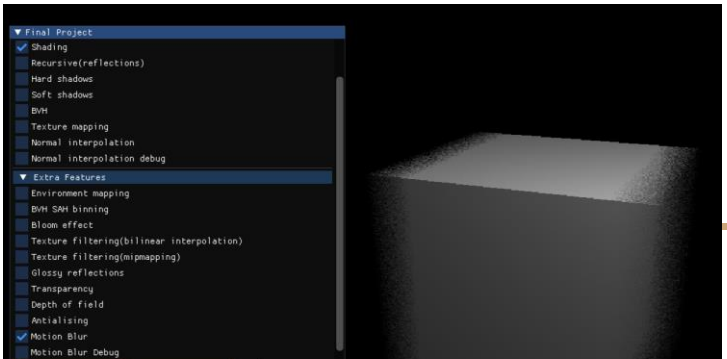


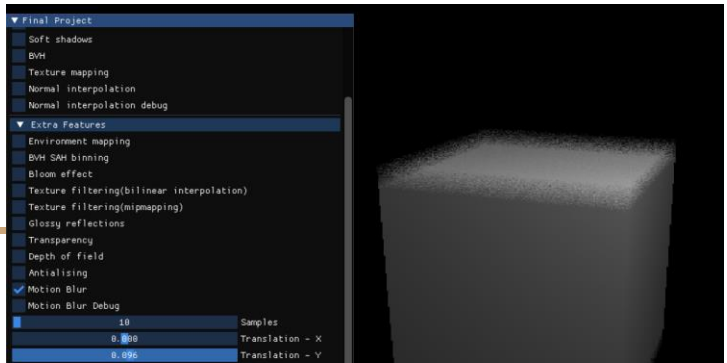
Like the box filter the main logic of motion blur is on main.cpp starting in line 405. Two auxiliary methods were created: `renderRayTracingMotion()` in render.cpp and `Trackball::generateRayMotion()` in trackball.cpp.

Firstly in main the number of samples determines how many renders we take having with the mesh in different positions. The final color of the scene will be the average of all these renders.

`renderRayTracingMotion()` - Essentially the same as the default renderRayTracing() method, but takes extra parameters. The parameters regarding translation can be directly controlled in the slider and represent the maximum position that the mesh can move. Meaning that if the slider "Translation X" has 0.2, for a particular render the farthest it can be from the original position is 0.2 in the X axis.

`generateRayMotion()` - is called in renderRayTracingMotion instead of generateRay(). This method takes the translation parameters and one random variable uniformly distributed between 0 and 1. This RV represents a random time. To represent motion, instead of moving the mesh (which is a lot harder in this particular case) we essentially move the camera instead, which will give equivalent results. Each viewing ray will have a random time set to it. Depending on the time that it has it will see the mesh in different positions. This is possible by offsetting the origin of the ray (`ray.origin = { position().x + translationX * time, position().y + translationY * time, position().z + translationZ * time }`). We can see that depending on the time it will be translated a certain amount, with the maximum being the translation threshold for time = 1.

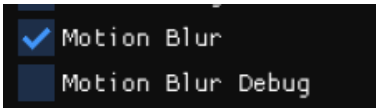




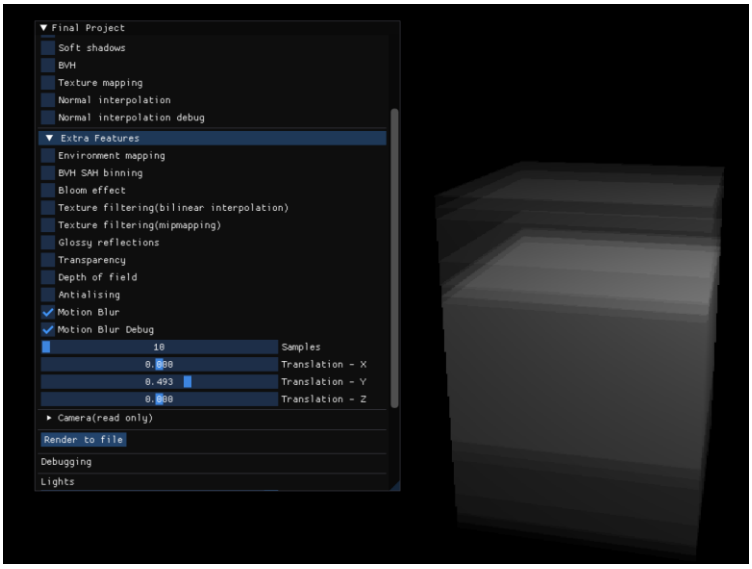
We can see here the cube with translations both on the X and Y axis (with 10 samples).

Visual debug:

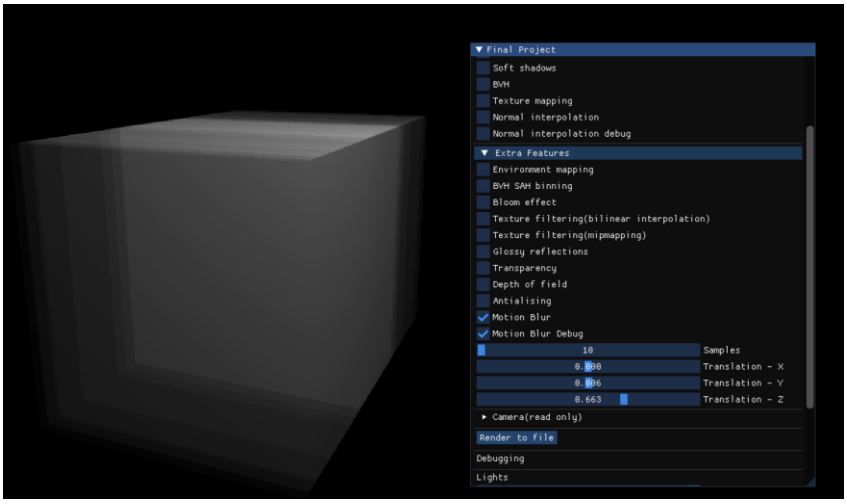
For this visual debug one extra checkbox is added to represent it, it will appear after clicking motion blur.



When this checkbox is ticked, now when moving the sliders we will see all the different positions that the mesh will have according to the random variable that represents time. This is very useful, since fully understanding the movement of the mesh makes it easy to know which behavior to expect from the blur itself.



We can essentially see all the positions of the 10 sampled cubes when translating in the Y direction. So each ray will hit one of these cubes depending on its time parameter.



Same behavior now for the Z axis.

Texture filtering: MipMaps

Description:

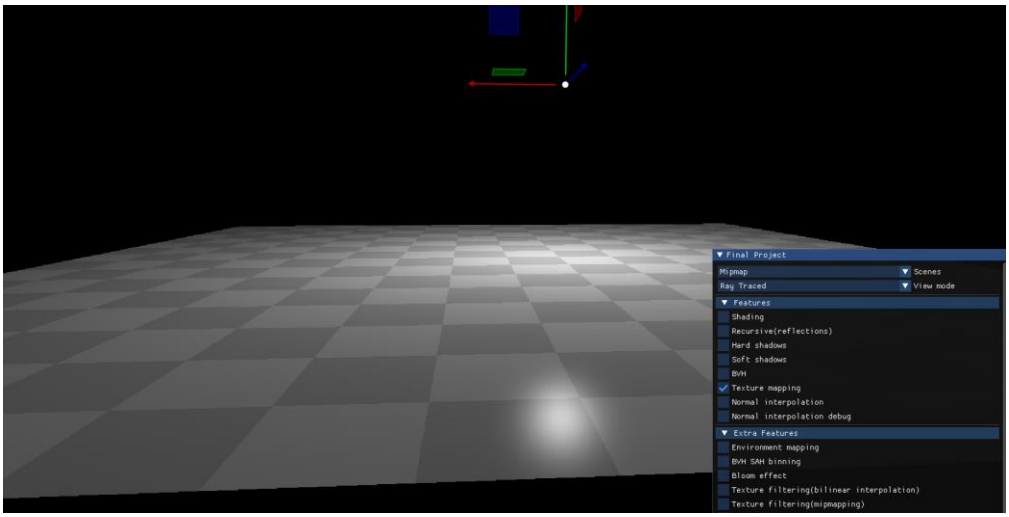
For mipmapping, in `loadScenePrebuilt()` in scene.cpp, once we load a mesh an auxiliary method in scene.cpp called `createMipMap()` is called to create the mipmap pyramid. This avoids having to create the mipmap every time we hit a pixel, massively increasing performance. 3 additional methods are present in texture.cpp, including an MipMap struct that has stored the level, width, height and an array of pixels.

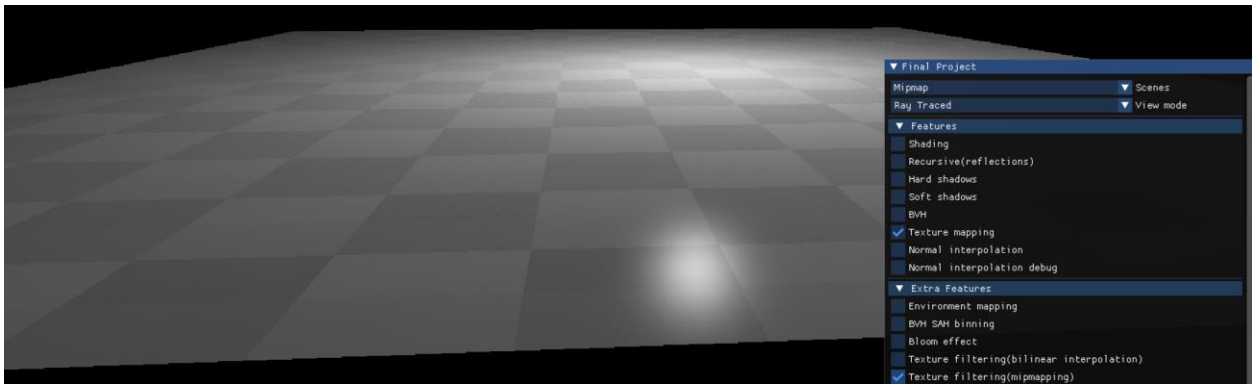
`initialiseMipMap(const Image& image)` - method in texture.cpp that just takes the initial texture image and puts it into the mipmap chain as level 0.

`generateMipMapPyramid()` - `createMipMap()` calls this method in a loop until a float variable representing the texture width is smaller than 1. For each call it will create the next mipmap level, which has half of the height and width of the previous one, as well as downsize the previous mipmap image to fit into these new dimensions. For this every new pixel equals the average of 4 pixels in the previous texture. We can achieve this by taking 2x2 boxes of pixels, in order to average them.

`acquireTexelMipMap()` - same functionality as the `acquireTexel()` method, but here we pass the mipmap level that we want to access in the mipmap chain.

When accessing the textures (in shading.cpp), to compute the level of detail necessary we take the log (base 2) of the distance from the viewpoint to the surface point hit. With this simple heuristic we can know from what mipmap level we should extract the texture. To have better effects trilinear interpolation is also used. For this we take the decimal part of the distance previously calculated to use as the interpolation weights. Using these weights we perform a weighted sum between the mipmap level taken and the level next to this one to get the final kd value.

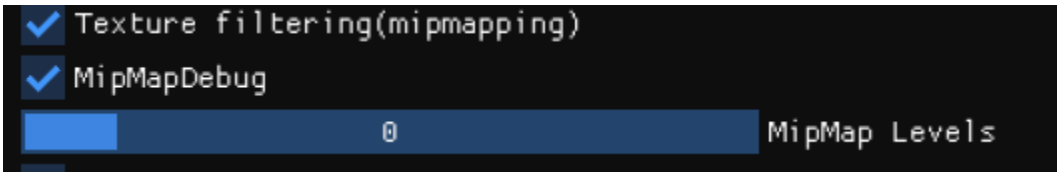




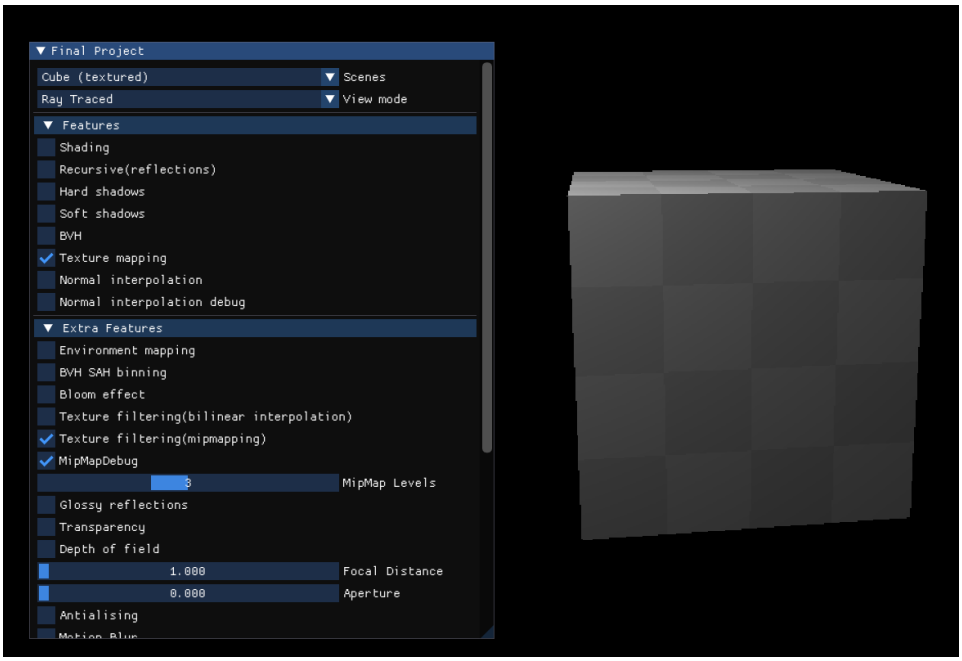
We can see here the effects that the mipmap has for points at different distances. Note: this extra scene can be found on the "Mipmap" tab.

Visual debug:

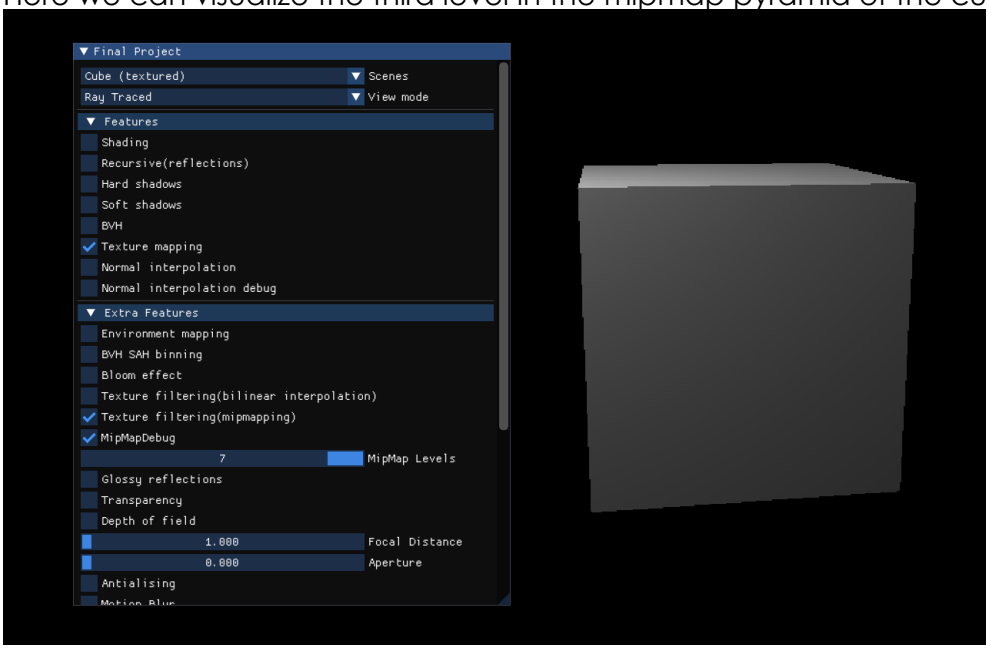
For this visual debug an extra checkbox was added. When pressing this checkbox a slider will appear that will allow us to control which mipmap level should be applied to the texture in the mesh.



With this we can essentially visualize all levels in a mipmap chain, giving us a good idea of how the final texture should look according to the distance of the pixels.



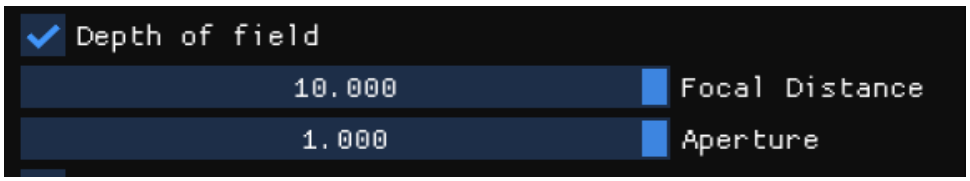
Here we can visualize the third level in the mipmap pyramid of the cube texture.

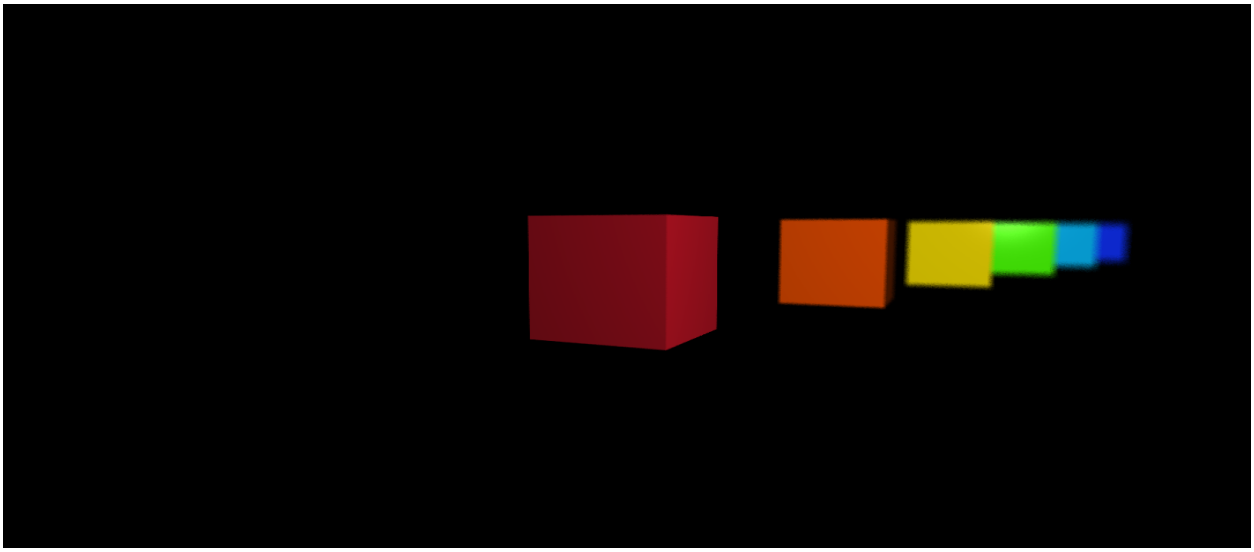


Here we can check the final mipmap level (which is just 1 pixel) for the cube texture.

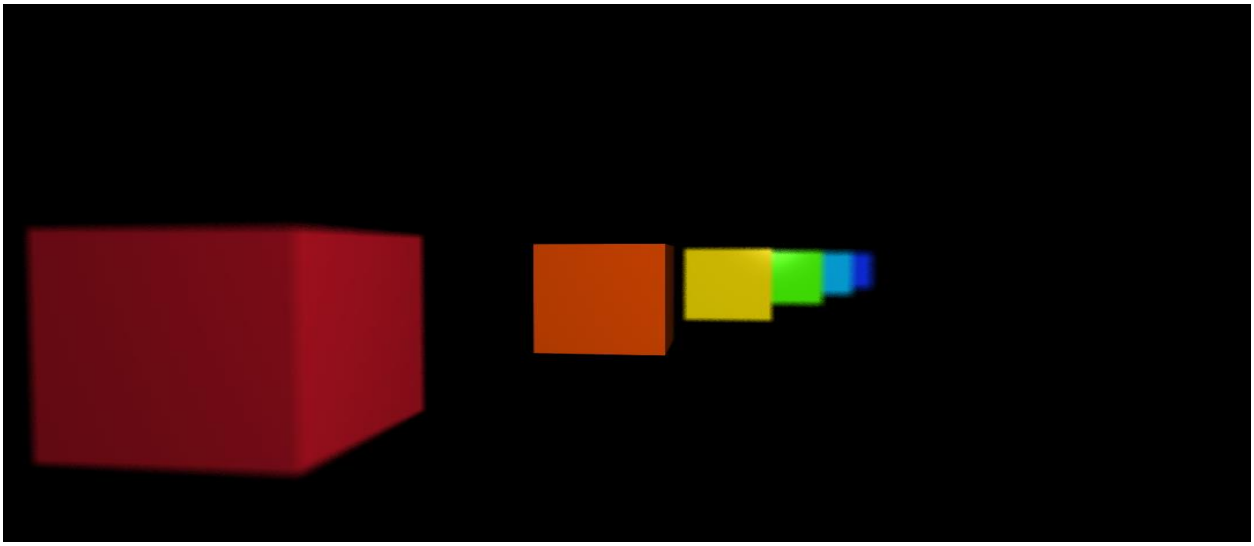
Depth of Field

E.g. (all images have Shading + Depth of Field)



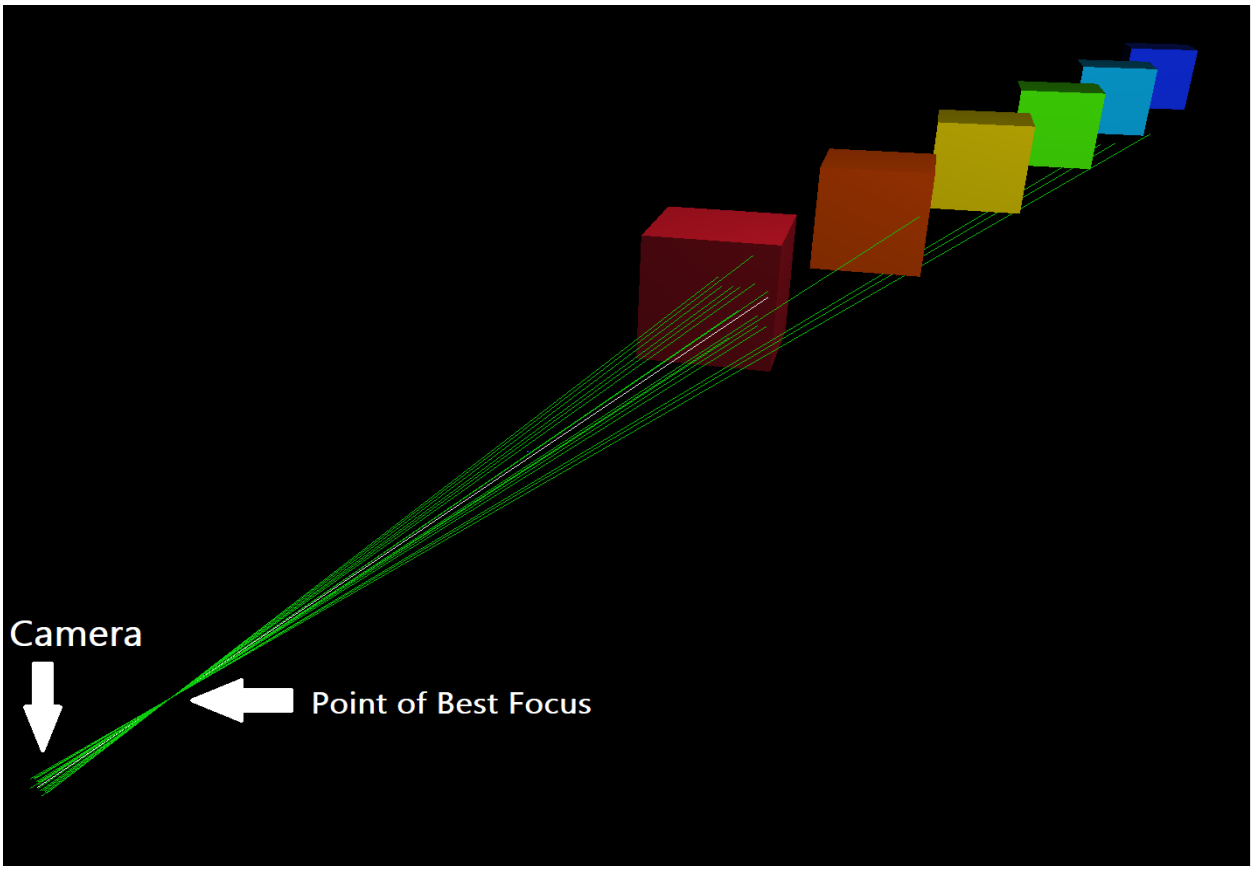


(^focus on the red cube)

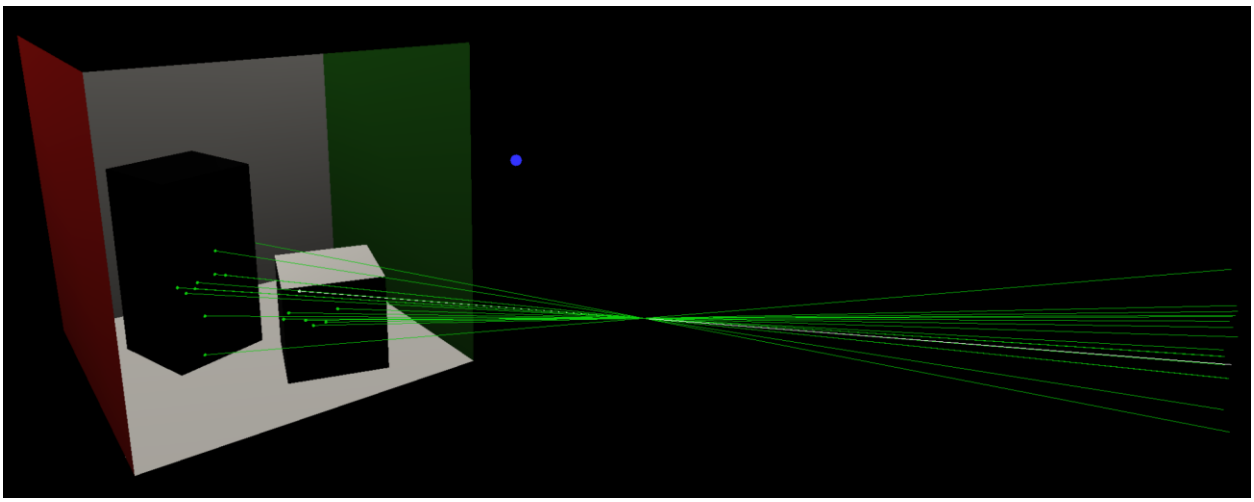


(^focus on the orange cube)

Visual Debug:



(^aperture factor of 1.0)



(^aperture factor of 5.0)

Environment Map

Description:

E.g.

(^ environment map + normal interpolation)

(^ environment + map)

(^environment map + recursive reflections + normal interpolation)

References:

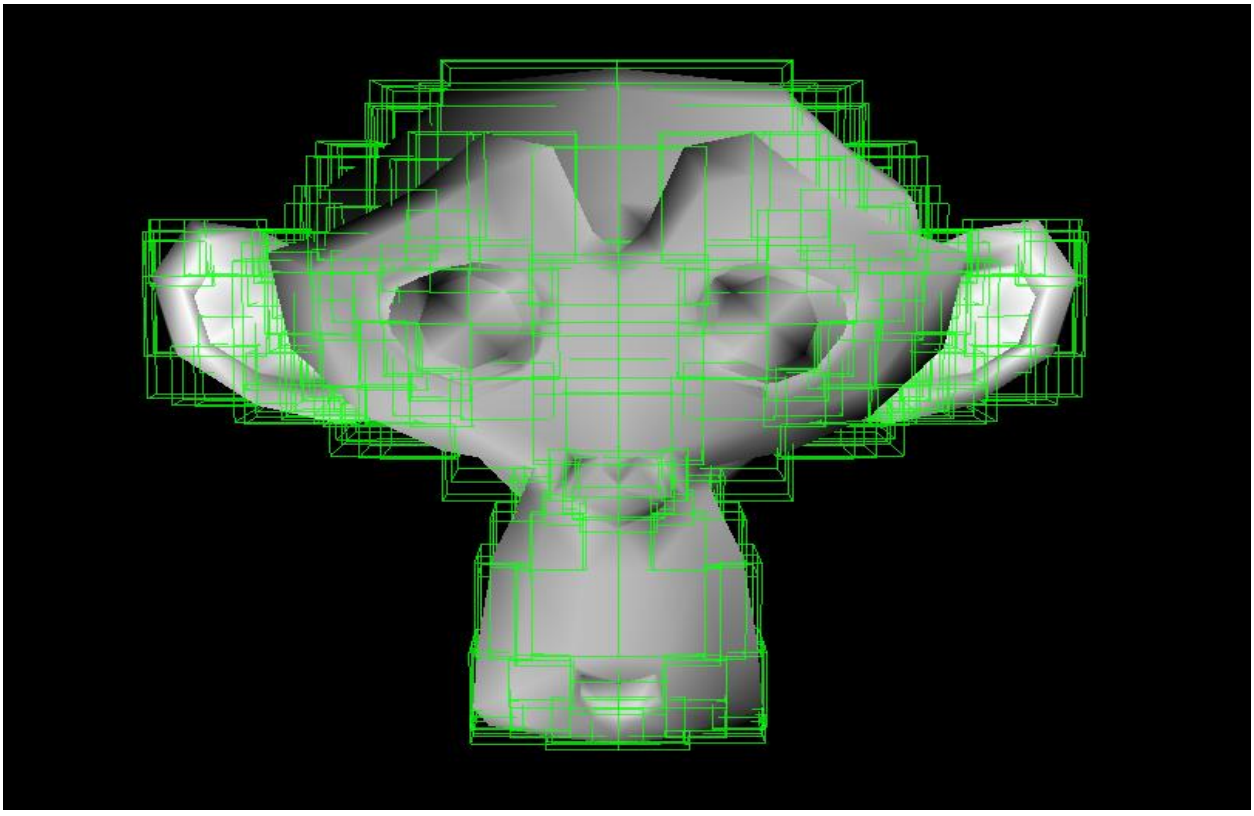
https://en.wikipedia.org/wiki/Cube_mapping

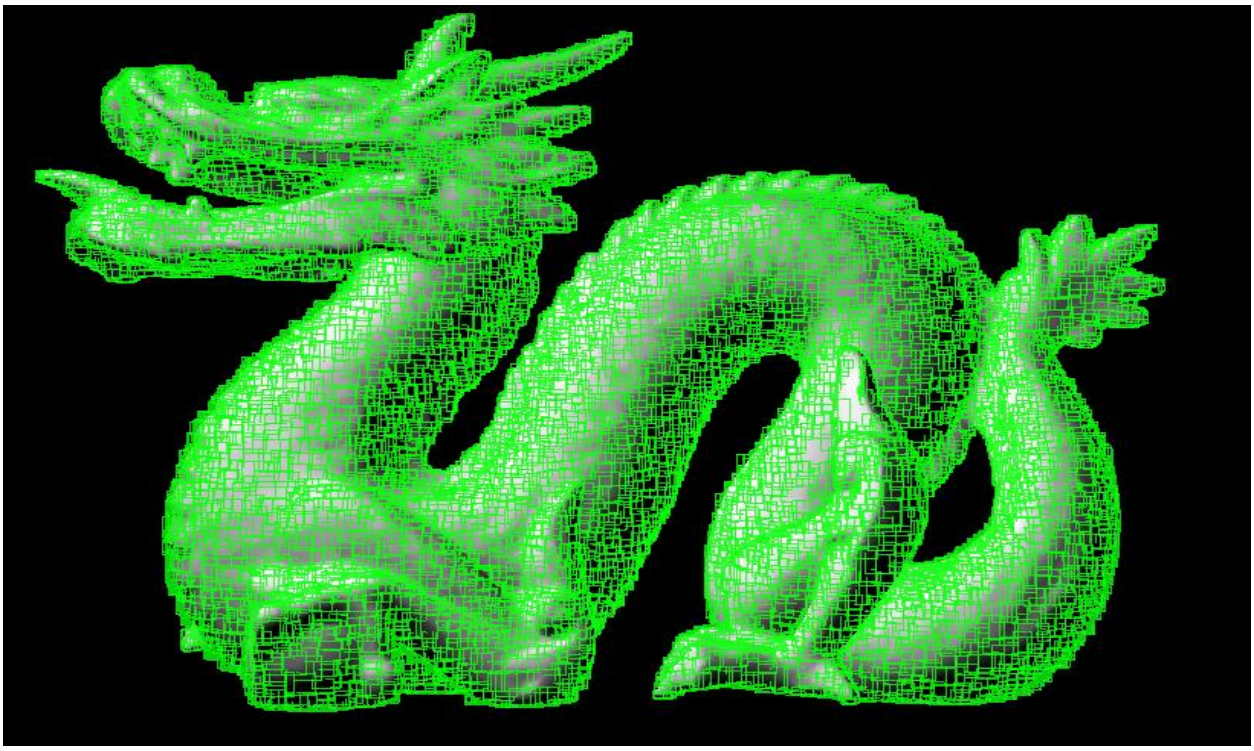
Surface Area Heuristic + Binning

Performance Table:

	Cornell Box	Monkey	Dragon
Time to Render	31.21 milliseconds	25.28 milliseconds	25.23milliseconds
BVH Levels	5	10	17

SAH+Binning performs better for bigger models. For example, the dragon runs 1.6 times faster while SAH+Binning is on.



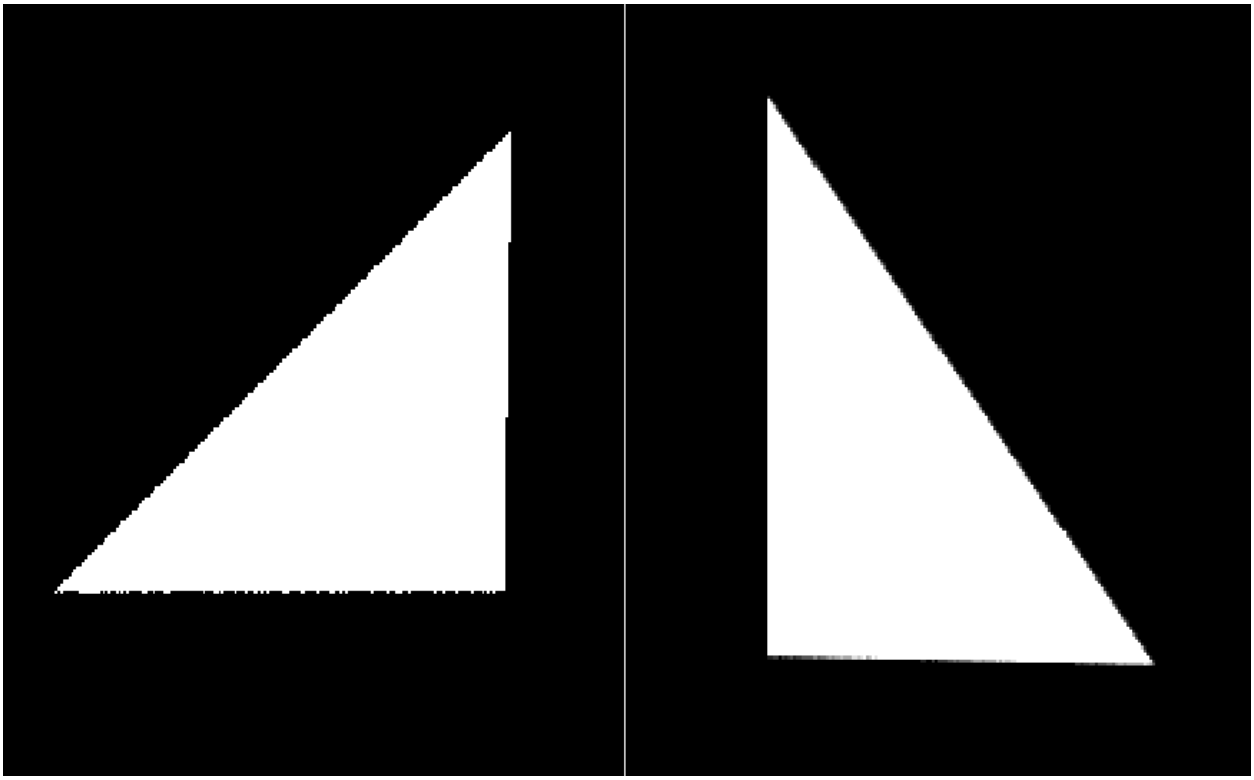


Cast multiple rays per pixel with irregular sampling

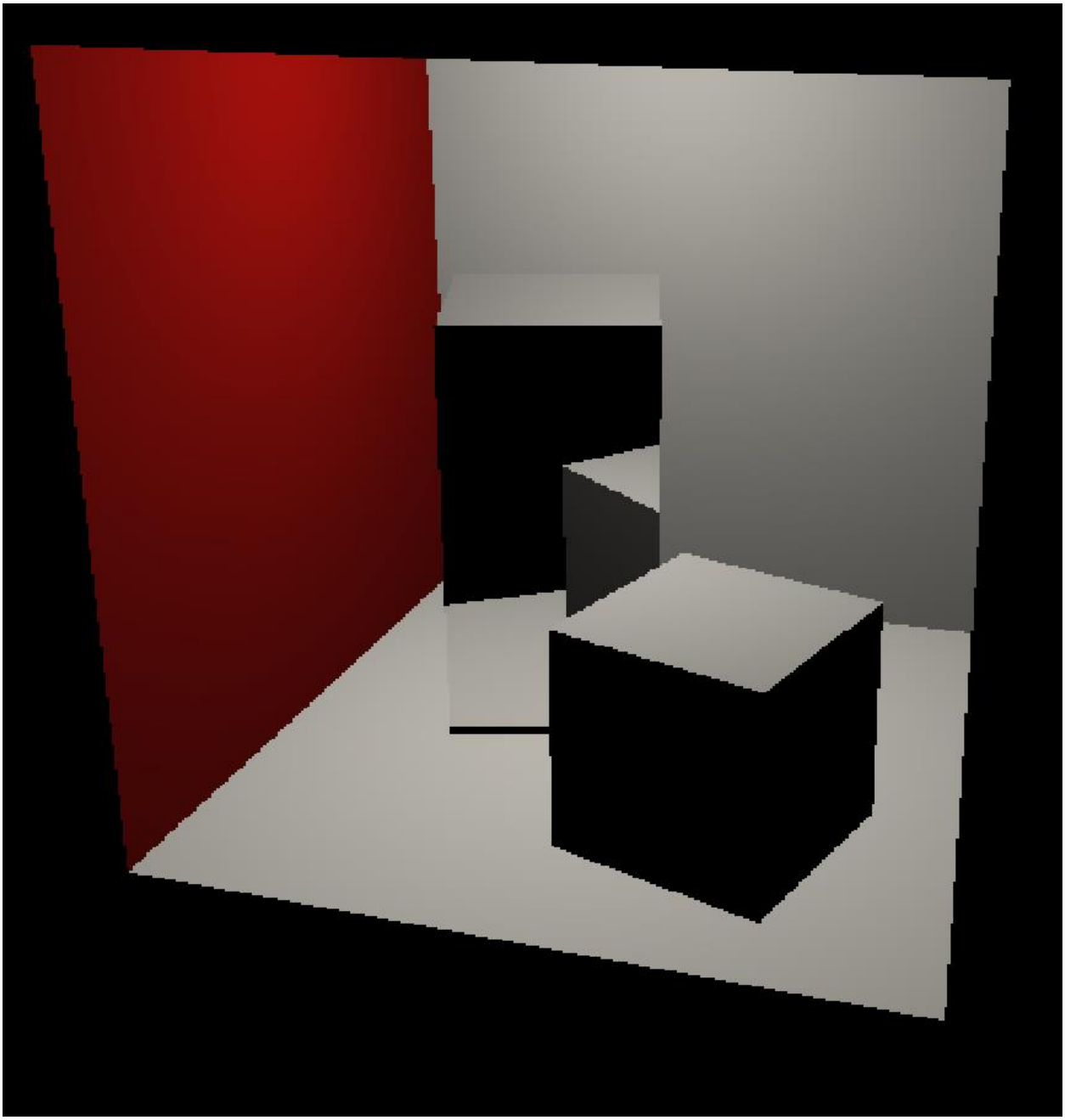
Also known as Antialising.

For every pixel of the screen we average the colours of it and its surroundings. We do the latter by generating a number of randomly generated pixels. For example, for every pixel, we generate a grid of size n x n and we take a random point.

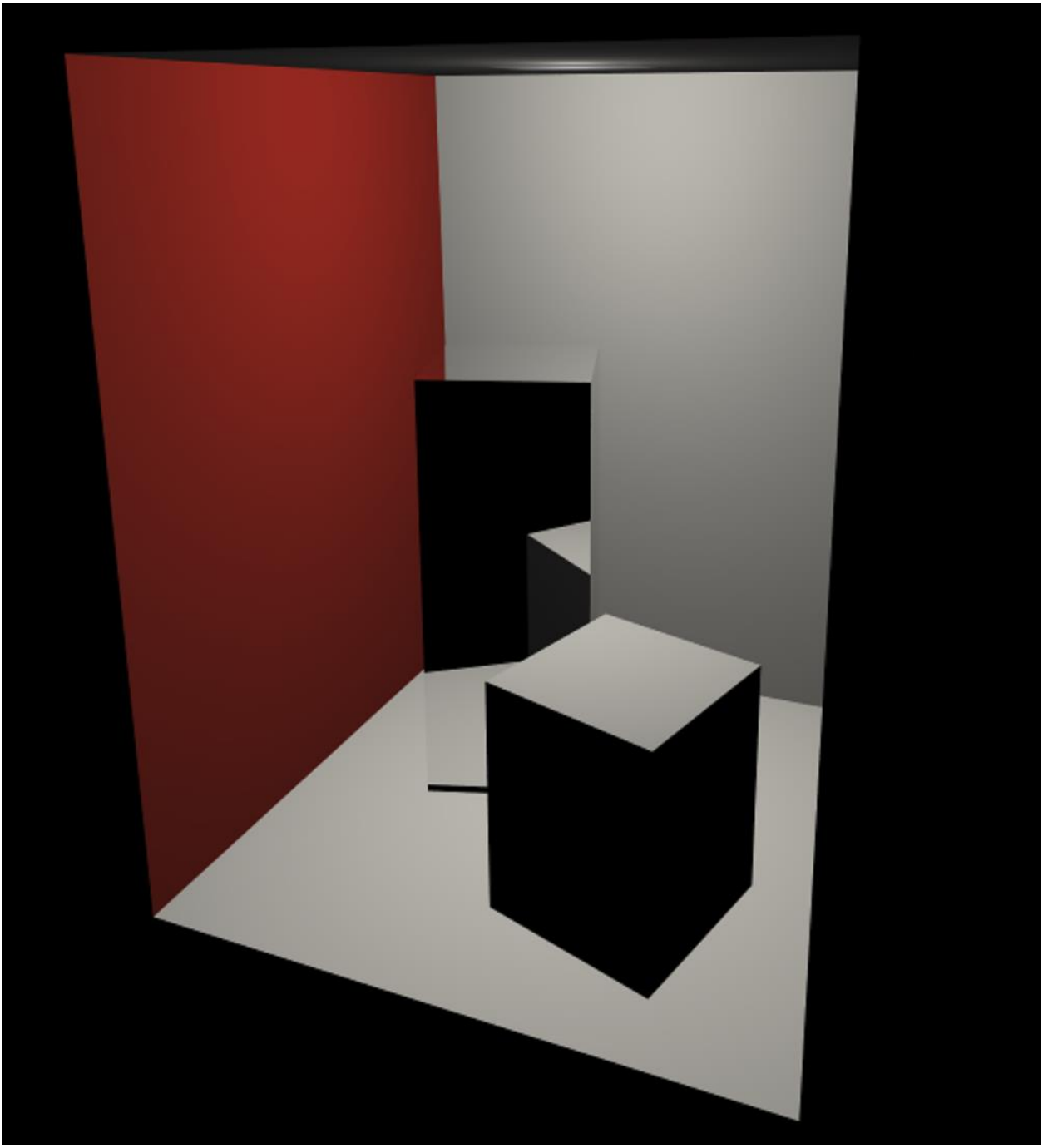
E.g.



(without (left) and with (right) irregular sampling)



(Cornell box without anti aliasing but with shading and recursive reflections)



(Cornell box **with anti aliasing** but with shading and recursive reflections)