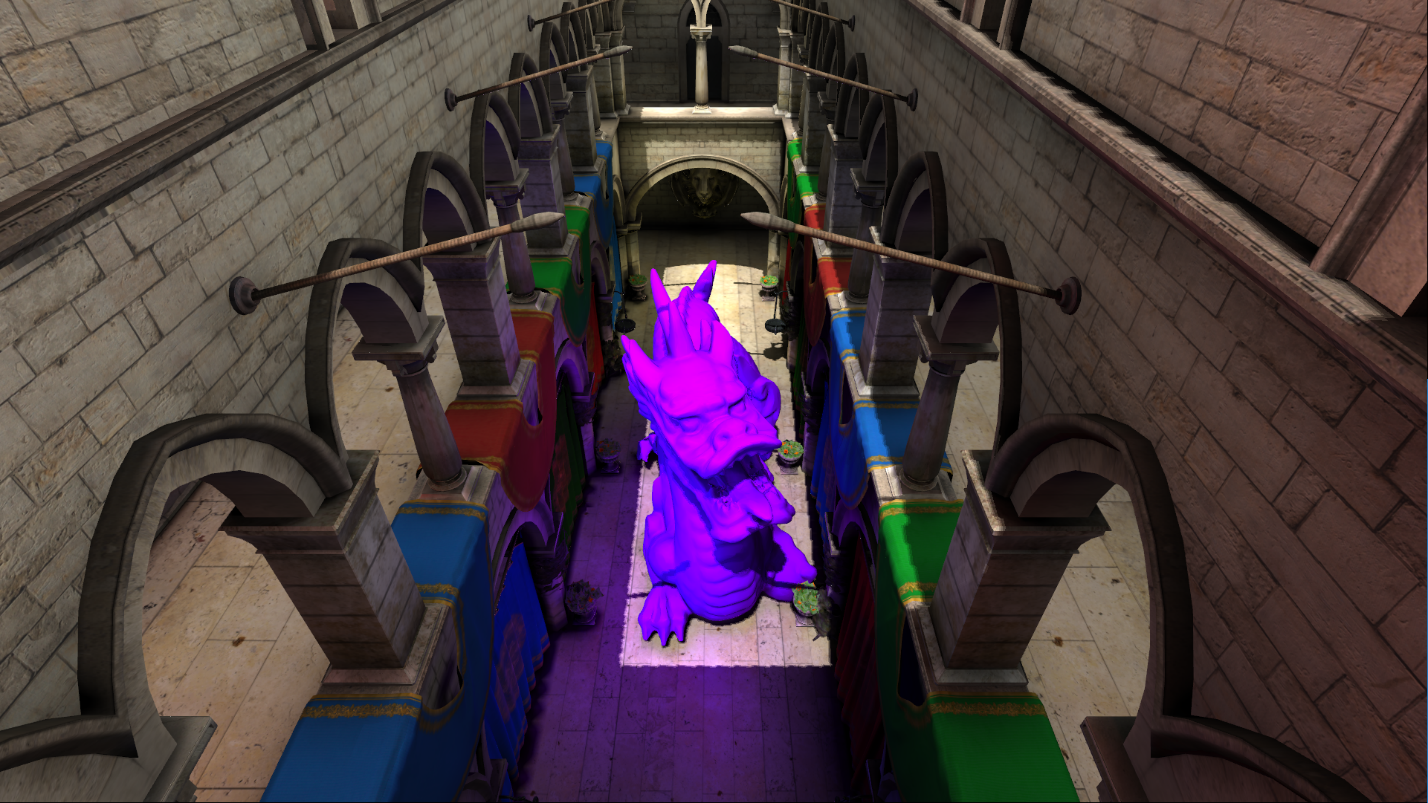
# Voxel Cone Tracing Report

By: Max Oomen



# Index page

**Introduction 3**

**Voxelization 4**

**Sparse Octree Construction 7**

**Light Insertion 9**

**Mipmapping 9**

**Neighbour Duplication 10**

**Cone Tracing 10**

**Future work 13**

**Sources 14**

## Introduction

My main goal was to implement real time global illumination. Global illumination simply means all lighting in a scene, i.e. direct lighting and indirect lighting. Most games only simulate direct lighting and add and an ambient factor to fake indirect lighting. However hardware is getting fast enough to simulate indirect lighting, which vastly improves realism. The main advantages of this algorithm is that it works in most game scenes, it allows for different quality levels with different performance costs, and it can be completely executed on the GPU. The main disadvantages of this algorithm is that the memory consumption is very high, and it does not work very well for scenes with very small high detailed geometry.

This algorithm works like ray tracing, but does some approximations to increase performance. Instead of sampling from the main scene, it samples from a voxelized version that approximates the scene. In ray tracing an indirect diffuse bounce requires casting many rays for a good result. This algorithm approximates bundles of these rays with cones, where a diffuse bounce can be calculated with a few cones. Cone tracing works by stepping over different mip levels of the voxelized scene.

The implementation is written in C++ and OpenGL. This is because I am most familiar with these, to speed up the prototyping phase.

## Voxelization

**Introduction**

There are many ways to voxelize a scene. I first started with a brute force CPU algorithm, I simply filled all voxels touching a triangle. This gave horrible results. I then read about using the rasterizer on the GPU for voxelization. I immediately went for this option since it required the minimal amount of code to be written.

The algorithm works by rasterizing the scene, where every generated fragment is a voxel. The viewport size first has to be set to the resolution of the voxel model. For example for a 256x256x256 voxel model, the viewport will be set to 256x256. Since the rendering pipeline only allows for a 2D output, a 3D output can be achieved by disabling writes to the render target and instead using imageStore to write to a 3D texture.

**Projecting triangles**

One issue that arises is that polygons on the XZ and YZ planes are not outputting voxel fragments. Curved surfaces also have holes in them. (See figure 1).

This issue is solved by doing an orthonormal projection on the most dominant axis of the triangle. The most dominant axis is found by taking the absolute values of the normal of the triangle, the axis with the greatest value is the dominant axis. This creates an almost correct result, this only issue being that there are holes in the voxel representation. (See figure 2).

Figure 1 Figure 2

**Conservative rasterization**

To fill the holes a conservative rasterization can be used. A normal rasterization generates a fragment for every pixel centre that touches a triangle, while conservative rasterization generates a fragment for every pixel that touches a triangle. Very recent graphics cards support conservative rasterization, however my laptop did not, so a conservative rasterization had to be emulated. This is done by expanding the triangle and creating an AABB around it to discard invalid fragments. (See figure 3). This does fill the hole problem, however it creates some voxel artefacts (figure 4), and the UV coordinates and normal are invalid (figure 5), leading to a worse quality voxel representation of the scene.

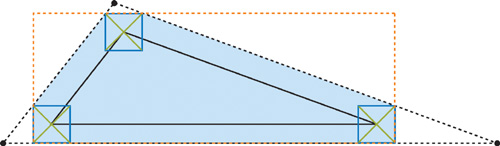


Figure 3 (source: <https://a248.e.akamai.net/f/248/10/10/http.developer.nvidia.com/GPUGems2/gpugems2_chapter42.html>)

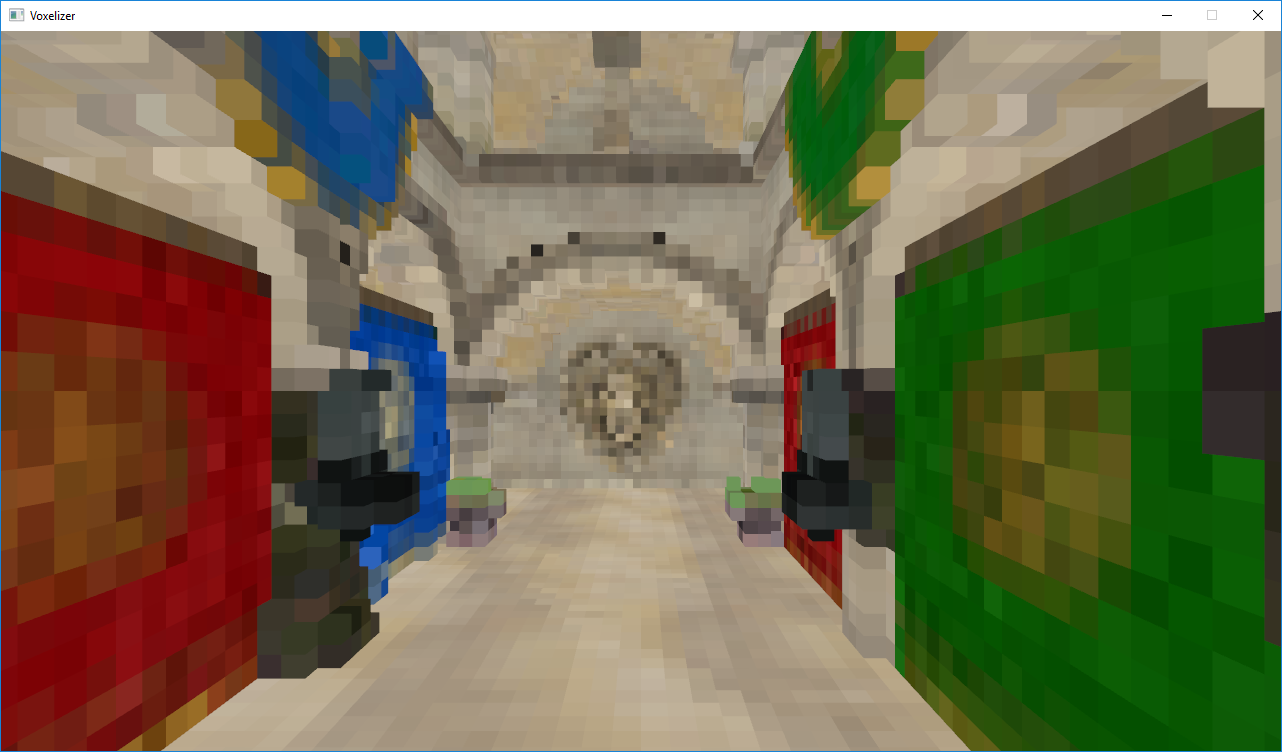
 

Figure 4 Figure 5

**Multiple triangles**

The original paper about voxel cone tracing recommends to rasterize every triangle on al 3 planes, unlike other sources which use conservative rasterization. I implemented this technique using a geometry shader that outputs three projected triangles. This gave far better results. (See figure 6)

One last quality wise issue is that multiple triangles will intersect with a voxel. This results in racing conditions between threads filling in voxels. This is solved by averaging all voxel fragments together. Averaging the values is trickier than usual, it relies on an atomic compare and swap function and storing the number of values averaged in the alpha byte. This introduces the limitation that only values that fit in less than 32 bits can be stored, since atomic operations on the GPU can only be executed on 32 bit integer values, and a few bits need to be used to store the number of values averaged.



Figure 6

**Code**

The voxelization code is in the following files:

Voxelization vertex shader: [assets/shaders/voxelization/voxelization.v.glsl](https://swarm1.nhtv.nl/swarm/files/Y2015B-Y3-IGADEngine/voxels/src/VoxelizerTest/VoxelizerTest/assets/shaders/voxelization/voxelization.v.glsl)

Voxelization geometry shader: [assets/shaders/voxelization/voxelization.g.glsl](https://swarm1.nhtv.nl/swarm/files/Y2015B-Y3-IGADEngine/voxels/src/VoxelizerTest/VoxelizerTest/assets/shaders/voxelization/voxelization.g.glsl)

Voxelization fragment shader: [assets/shaders/voxelization/voxelization.f.glsl](https://swarm1.nhtv.nl/swarm/files/Y2015B-Y3-IGADEngine/voxels/src/VoxelizerTest/VoxelizerTest/assets/shaders/voxelization/voxelization.f.glsl)

Voxelization C++ code: [src/voxelizer.cpp](https://swarm1.nhtv.nl/swarm/files/Y2015B-Y3-IGADEngine/voxels/src/VoxelizerTest/VoxelizerTest/src/voxelizer.cpp) (the function void Voxelizer::Voxelize(…) (line 468) launches the voxelization)

## Sparse Octree Construction

**Introduction**

One of the main issues of storing voxels in a 3D texture is that the amount of memory required scales cubically with the resolution of the voxel model, the exact formula is , where is the resolution and is the number of bytes per voxel. The comes from the fact that mipmaps also need to be stored. Since colours, normals and incoming light has to be stored, that is 12 bytes per voxel. Using a resolution of 512x512x512 that will result in a total amount of 1.8GB. Modern graphics cards usually have about 2GB of memory, this will not fit when also textures and models have to be stored on it.

A better way to store the voxels is by constructing a sparse octree. The cost scales linearly, , where is the number of voxels stored, and is the number of bytes per voxel. There is a 4 byte overhead per voxel to store a child pointer. Nodes are grouped per 8, this way only a single pointer is required to traverse to all 8 children. Note that the storage scales linearly with the number of voxels stored, not with the resolution.

**Advantages and disadvantages**

Another advantage that the octree brings is that nodes can be marked static. This allows for a system were only voxels from dynamic voxels are cleared and filled back.

The main disadvantage of this structure is that in a worst case scenario the storage size will be even greater than a 3D texture, however this will practically never happen in a game scene. Another disadvantage is that the ability to hardware quadrilinear interpolation is lost. There is however a way to achieve hardware trilinear interpolation which is described later. Additionally there is the disadvantage that a lookup is no longer , it has become .

**Construction**

The construction of the Octree is done on the GPU. Construction of a data structure in a massive multi-threaded environment is very different from construction in single threaded environment. The construction algorithm is implemented in two shaders. The first one is in the fragment shader of the voxelization, this is to prevent an intermediate buffer to store the voxels. Every thread will traverse the octree to the position of the voxel. If a node needs to be split new nodes will need to be allocated. This is done using a buffer of already allocated nodes and an atomic counter. Before the allocation of more nodes can happen, the thread first needs to acquire a lock, this is to prevent multiple threads from splitting the node at the same time. This can’t be made lockless since it can’t be done in an atomic operation. If a thread fails to acquire a lock it will store itself in a buffer for a deferred execution.

**Deferred threads**

When all the models have gone through the rendering pipeline for rasterization, there can be values stored in the buffer of stored threads. A compute shader is launched to insert voxels for all these stored threads. A compute shader run might also store threads for deferred execution. To solve this the compute shader is ran until the thread buffer is empty.

**Hardware trilinear interpolation**

One of the big downsides of the sparse octree is that hardware interpolation does no longer work. This is because the nodes are stored in a Shader Storage Buffer Object (SSBO) instead of a texture. To work around this issue, the leaf nodes are removed and instead replaced with a pointer to a texture brick. (Note: since GLSL does not support pointers, it is actually an offset of an array.) The texture bricks are all stored in a single 3D texture. The size of a texture brick is 3x3x3. Where the top right front 2x2x2 texels store the actual voxels of that node, the rest is used to duplicate the neighbouring values. Figure 7 is a representation of what this would look like for a 1D case with a binary tree.



Figure 7

Note: One thing that is not displayed in figure 7 is that inner nodes also get a texture brick, this is to store the mips.

**Code:**

Octree construction fragment shader: [assets/shaders/voxelization/voxelization.f.glsl](https://swarm1.nhtv.nl/swarm/files/Y2015B-Y3-IGADEngine/voxels/src/VoxelizerTest/VoxelizerTest/assets/shaders/voxelization/voxelization.f.glsl)

Deferred octree construction compute shader: [assets/shaders/voxelization/deferredVoxelization.c.glsl](https://swarm1.nhtv.nl/swarm/files/Y2015B-Y3-IGADEngine/voxels/src/VoxelizerTest/VoxelizerTest/assets/shaders/voxelization/deferredVoxelization.c.glsl)

## Light Insertion

**Description:**

The next step is to insert lighting into the octree. First the scene is rendered from the light’s perspective, storing depth values in a texture. Then a compute shader is launched with a thread for every pixel in the depth texture. Every pixel represents a photon and is inserted into the octree. My implementation currently stores pre filtered diffuse lighting, this will later have to be changed to a more flexible system to allow for specular lighting. The photons can’t be inserted into the octree during the rendering of the scene, since it is unknown what object blocks what. The depth texture can also be reused as shadow maps which is required for the final rendering pass.

**Code**

Shadow mapping vertex shader: [assets/shaders/shadow\_mapping.v.glsl](https://swarm1.nhtv.nl/swarm/files/Y2015B-Y3-IGADEngine/voxels/src/VoxelizerTest/VoxelizerTest/assets/shaders/shadow_mapping.v.glsl)

Shadow mapping fragment shader: [assets/shaders/shadow\_mapping.f.glsl](https://swarm1.nhtv.nl/swarm/files/Y2015B-Y3-IGADEngine/voxels/src/VoxelizerTest/VoxelizerTest/assets/shaders/shadow_mapping.f.glsl)

Light insertion compute shader: [assets/shaders/voxelization/insert\_light.c.glsl](https://swarm1.nhtv.nl/swarm/files/Y2015B-Y3-IGADEngine/voxels/src/VoxelizerTest/VoxelizerTest/assets/shaders/voxelization/insert_light.c.glsl)

Shadow Mapping C++ code: [src/app.cpp](https://swarm1.nhtv.nl/swarm/files/Y2015B-Y3-IGADEngine/voxels/src/VoxelizerTest/VoxelizerTest/src/app.cpp) (the function void App::RenderShadowmap() (line 671) launches the shadow mapping code)

Light insertion C++ code: [src/voxelizer.cpp](https://swarm1.nhtv.nl/swarm/files/Y2015B-Y3-IGADEngine/voxels/src/VoxelizerTest/VoxelizerTest/src/voxelizer.cpp) (the function void Voxelizer::InsertLight(…) (line 803) launches the light insertion)

## Mipmapping

**Description:**

The cone tracing works by sampling from lower resolutions of the voxel model as the cone gets wider. The lower resolution is achieved by simply generating mipmaps. For an octree of levels, this can be done in steps. For every step a compute shader is launched with one thread per possible voxel location. A thread will then average the values from its children and store that. This is a very inefficient way of generating mipmaps, but this was the simplest way to get a correct result.

The original paper mentions a, what seems to be, different algorithm for generating the mipmaps. I found the explanation in the paper to be too confusing. Simply averaging eight values seems to work fine.

**Code:**

The compute shader generating the mipmaps: [assets/shaders/voxelization/generate\_mipmap.c.glsl](https://swarm1.nhtv.nl/swarm/files/Y2015B-Y3-IGADEngine/voxels/src/VoxelizerTest/VoxelizerTest/assets/shaders/voxelization/generate_mipmap.c.glsl)

## Neighbour Duplication

Neighbour duplication is required for the hardware texture trilinear interpolation to work. My implementation does this in the same shader as the mipmap generation. This is because it was the simplest way to implement. Every node will copy the neighbouring data of its children over. There are a total of 7 neighbouring nodes to visit, filling a total of 19 texels.

The shader also calculates neighbour pointers which are going to be used later for a more optimized algorithm of copying neighbouring data.

**Code:**

The compute shader duplicating the neighbouring data: <https://swarm1.nhtv.nl/swarm/files/Y2015B-Y3-IGADEngine/voxels/src/VoxelizerTest/VoxelizerTest/assets/shaders/voxelization/generate_mipmap.c.glsl>

## Cone Tracing

Cone tracing is tracing a ray that has a width, this is to approximate tracing a bundle of rays. A cone has an origin, direction and a height to radius ratio. The origin is offset by a constant in the direction of the surface normal, this is to prevent the cone hitting a voxel generated by the surface it’s originating from. Cone tracing is simulated by starting at the origin and then stepping along the cone in every time increasing steps. A level of detail (LOD) value is determined based on the radius, where a greater radius means a lower LOD. A visual representation is displayed in figure 8.

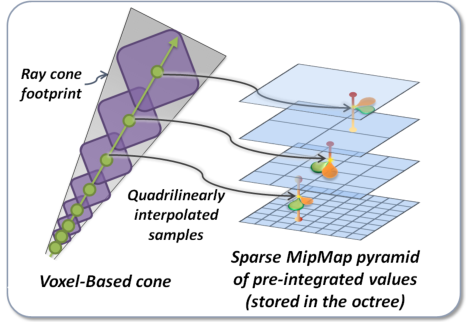


Figure 8 (source: <http://maverick.inria.fr/Publications/2011/CNSGE11b/GIVoxels-pg2011-authors.pdf>)

RGB values are sampled from the lighting data stored in the octree, and an A value is sampled from the colour data in the octree. The RGB value represents incoming light on the voxels, the A value represents occlusion. These values are added to an accumulator. The RGB values are linearly attenuated as the cone becomes wider. The RGBA values are then added to the accumulator. When the alpha value of the accumulator is greater or equal to one, the cone is fully occluded and has to be stopped. The total value of the accumulator represents indirect lighting entering the cone and lighting the surface.

The paper talks about linearly attenuating the samples from the voxel data structure as the distance from the surface increases. In here they introduce a new factor which is not described anywhere in the paper. I suspect it is a value that has to be tweaked until the result looks good.

I added a scale factor that was not talked about in the paper, which helped me create a far better image, which was to scale the read values based on the mip levels. The scale I used is pow(8, LOD), which turns the average into a total.

My implementation of cone tracing has some bugs that have to be fixed. However it does show relatively convincing indirect illumination being simulated (figure 10).

**Code:**

The fragment shader tracing cones: [assets/shaders/mesh\_shader.f.glsl](https://swarm1.nhtv.nl/swarm/files/Y2015B-Y3-IGADEngine/voxels/src/VoxelizerTest/VoxelizerTest/assets/shaders/mesh_shader.f.glsl)

The shared shader containing shared functionality: [assets/shaders/voxelization/shared.glsl](https://swarm1.nhtv.nl/swarm/files/Y2015B-Y3-IGADEngine/voxels/src/VoxelizerTest/VoxelizerTest/assets/shaders/voxelization/shared.glsl)

The C++ code calling the rendering code: [src/app.cpp](https://swarm1.nhtv.nl/swarm/files/Y2015B-Y3-IGADEngine/voxels/src/VoxelizerTest/VoxelizerTest/src/app.cpp) (The function int App::Render() (line 848) is the start of all rendering invocations)

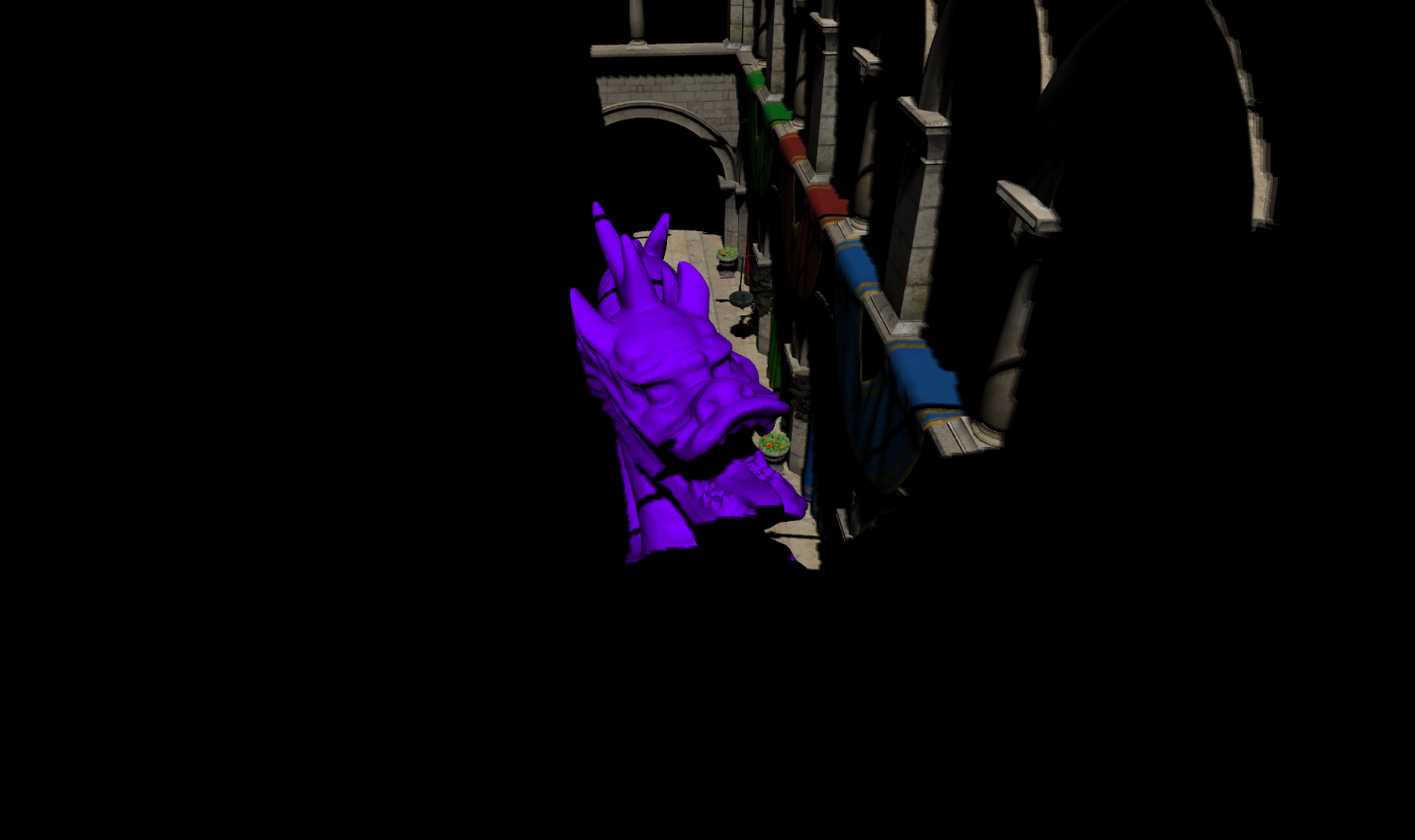


Figure 9: Indirect Illumination disabled

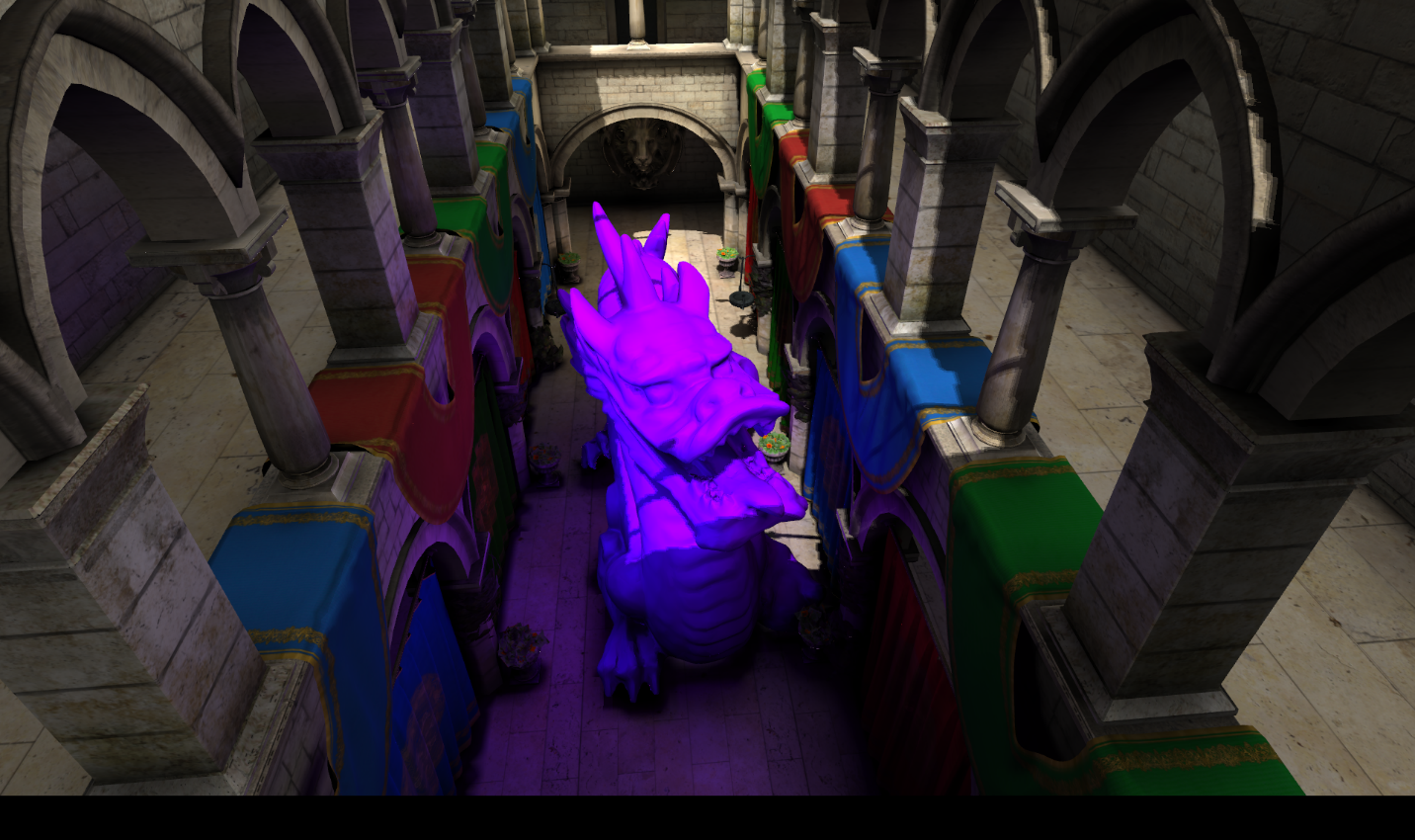


Figure 10: Indirect Illumination enabled

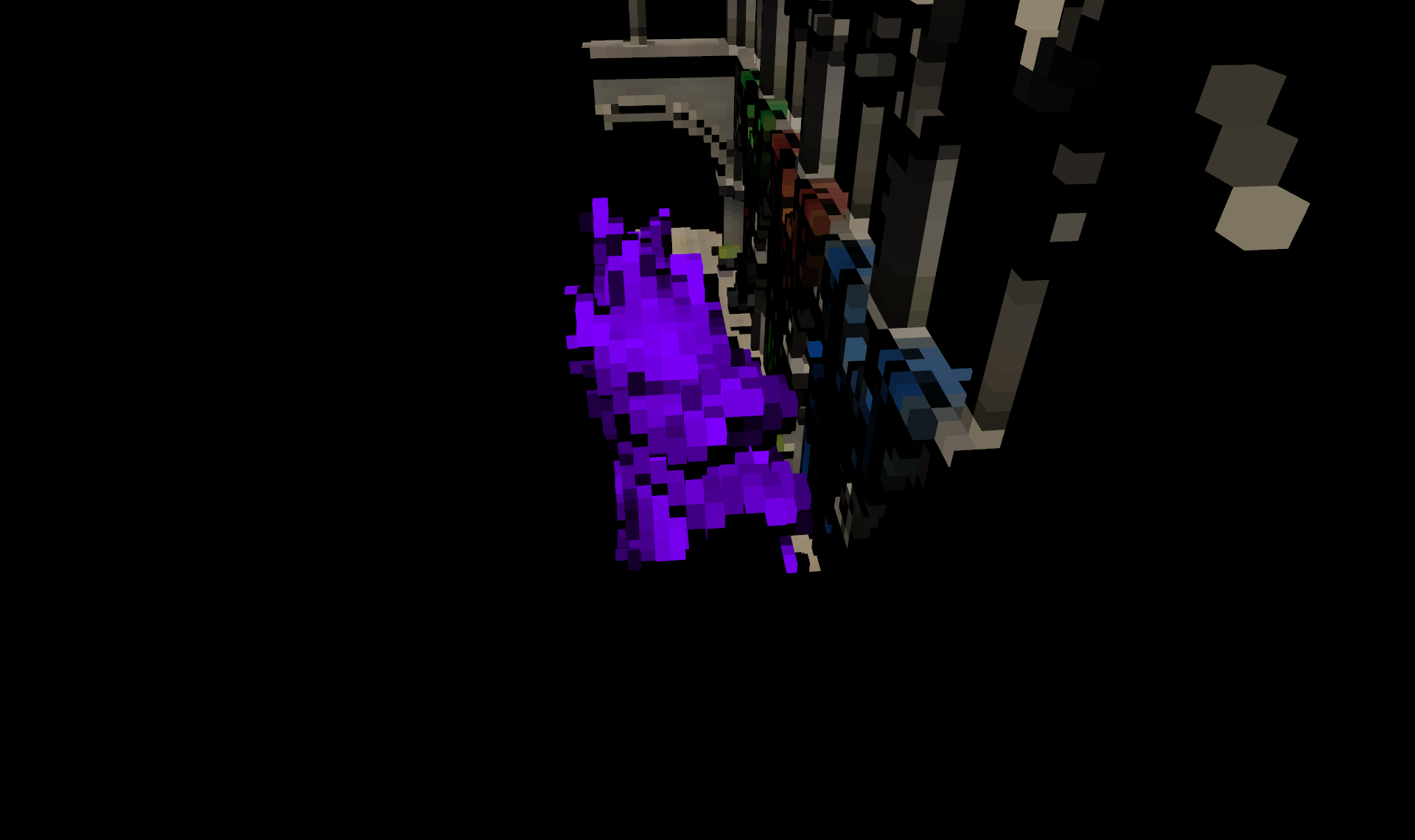


Figure 11: Debug representation of lit voxels.

## Future work

**Improving quality**

Although I managed to create some convincing images, there is still a lot of room for improvement. My implementation currently only support indirect diffuse lighting, since it is stored pre filtered. In the voxel cone tracing paper they talk about storing incoming radiance as a gaussian lobe, I want to look into this to add indirect specular reflections.

I am currently looking for a benchmark, that produces an as realistic as possible image to compare my results to, I am thinking about one of these three options: write a path tracer, use the Nvidia’s OptiX path tracer, or generate lightmaps using beast in unity. Having a reference image will help tweak constants and find bugs to create an as realistic image as possible.

**Learning from Q-Games**

At the end of the project I found set of slides by Q-games who managed to do multiple bounces of indirect illumination using voxel cone tracing. I am interested in implementing this as well for future faster graphics cards that could run that at a real time frame rate.

Q-Games also talked about cascading voxels, where nearby geometry is voxelized at a fairly high resolution, and geometry that is further away voxelized at lower resolutions. (The same idea as cascaded shadow maps, less detail further away.) This works perfectly for voxel cone tracing since far away geometry is going to sample big voxels anyway since the cones get wider as the distance increases.

**Possible optimizations:**

There are also a lot of optimizations I can do. First of all deferred shading would improve performance by a lot. In my current implementation a mesh gets drawn, all shading gets calculated, and then something could be drawn over it. This problem can be avoided by first drawing all geometry to G buffer and then calculating shading for every pixel. It also allows for optimizations where the indirect illumination is calculated at half the resolution with more detail on edges.

Another optimization is to voxelize all static geometry once, and the only re voxelize the dynamic parts. This could potentially also increase the performance of the mipmapping since only nodes with dynamic children will have to be mipmapped. One downside to this is that scenes are no longer be fully dynamic.

The original voxel cone tracing paper also talks about an optimization to improve the neighbour duplication time. I will have to look into this, since my implementation spends a lot of time duplicating neighbours.

## Sources

The original Voxel Cone Tracing paper by Cyril Crassin et al: <http://maverick.inria.fr/Publications/2011/CNSGE11b/GIVoxels-pg2011-authors.pdf>

A blog post about a voxel cone tracing implementation by Simon: <http://simonstechblog.blogspot.nl/2013/01/implementing-voxel-cone-tracing.html>

An article from GPU Gems 2 about conservative rendering Jon Hasselgrn, Tomas Akenine-Möller and Lennart Ohlsson: <https://a248.e.akamai.net/f/248/10/10/http.developer.nvidia.com/GPUGems2/gpugems2_chapter42.html>

A chapter from OpenGL insights talking about real time voxelization in OpenGL by Cyril Crassin and Simon Green: <https://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGLInsights-SparseVoxelization.pdf>

The slides with speaker’s notes about voxel cone tracing in The Tomorrow Children by James McLaren from Q-Games: <http://fumufumu.q-games.com/archives/Cascaded_Voxel_Cone_Tracing_final_speaker_notes.pdf>