Masterclass Assignment: Real-time Global Illumination

Joe Withers*

Abstract

This document details the approach I took to complete the Masterclass assignment, evaluates the results, and lists possible improvements I could make in the future.

1 Introduction

For this assignment I decided to implement the technique *Interactive Indirect Illumination using voxel cone tracing* [Crassin et al. 2011], commonly referred to as VXGI, to achieve realtime global illumination. This technique achieves real-time global illumination rendering the directly illuminated scene geometry into a three-dimensional texture, which can be sampled in a deferred shading pass using cone tracing to calculate accurate indirect diffuse and specular lighting terms.

2 Method

I used C++ and OpenGL for this assignment, utilising the NCCA Graphics Library NGL [Macey 2014] to interact with OpenGL, and Qt5 to create the user interface. I based my implementation around a demo of the Sponza atrium using NGL [Macey 2017], that although is quite old, saved me from spending a lot of time writing a grouped Wavefront OBJ and MTL file parser. I extended upon this demo by converting it to use deferred shading, which is necessary for VXGI and other post process effects, using tutorial material from LearnOpenGL [Vries 2016a].

The first step of implementing VXGI was to create a voxelized representation of the albedo and surface normal information in the scene. The method used in the source material [Crassin et al. 2011] is explained in great detail in an OpenGL Insights chapter titled *Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer* [Crassin and Green 2012], but the basic concept is as followed:

1. Generate three projection matrices for orthogonal projections, covering the scene geometry equally, and aligned to each of the coordinate axes.
2. Draw the scene with depth testing disabled, ensuring fragments are generated for every triangle.
3. For each triangle, determine which of the coordinate axes is most aligned with the surface normal, thus generating the maximum number of fragments when transformed using the corresponding projection matrix for that axis.
4. Using either the NVIDIA OpenGL extension *GL CONSERVATIVE RASTERIZATION NV*, or a geometry shader, enlarge each triangle so that fragments are generated regardless of whether it covers a pixels center. This helps to reduce 'cracks' in the voxelisation on surfaces that are angled away from the coordinate axes.
5. Using the fragment coordinates, depth, and the axis chosen in step 3, infer the texel coordinate that the fragment corresponds to in the target 3D textures. Store the fragment albedo and normal information in 3D textures using a moving average.

*e-mail: joewithers96@gmail.com

Whilst this technique is somewhat easy to explain, I had not previously used the required Image Load/Store commands so I referred to an implementation I found on GitHub [Lin 2013] to guide me on the OpenGL side of things.

The next step was to inject the scene's emissive values into a 3D texture. This is achieved by raytracing towards the light, for each of the non-transparent texels in the voxelised textures. This is where my implementation differs from the source material [Crassin et al. 2011] as I opted to use a dense 3D texture as opposed to a sparse voxel octree. Whilst this was significantly easier to set up, it also severly impacts the speed at which raytracing can be performed through this 3D texture (See Figure 2 for performance metrics). The source material [Crassin et al. 2011] also details a method for mipmapping this texture anisotropically along each of the coordinate axes, however I opted to use OpenGL's `genMipMaps` method to save time.

The final step was to implement cone tracing in the deferred shading pass. This pass calculates the indirect light contribution, specular lobe contribution, and also soft shadows, using the emissive 3D texture and the textures generated by the G-Buffer.

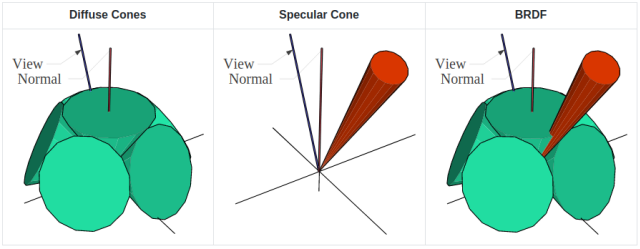

Figure 1: *A series of diagrams showing how the combination of a specular cone and multiple diffuse cones can be used to approximate a material BRDF. [Villegas 2016] The diffuse cones form a normal-oriented hemisphere to approximate the diffuse reflections, whilst the specular cone follows the reflection vector and it's aperture varies dependent on the material roughness.*

For the indirect and specular light contributions I referred to VCTRenderer [Villegas 2016] for example code on implementing an efficient cone tracing function, and how it can be used to approximate a BRDF as shown in Figure 1. I then adapted the specular component to work with the PBR materials to ensure the specular cone aperture is physically plausible for any given viewing angle.

To calculate the the direct lighting contribution I referred to tutorial materian from LearnOpenGL [Vries 2016b] to ensure that the PBR textures passed through the G-Buffer are combined in a physically plausible way. I referred again to VCTRenderer [Villegas 2016] to implement soft shadows into the direct lighting component, which involves calculating a fragment's occlusion by tracing a cone back towards the light source and accumulating the opacity values at each step.

Finally I added additional controls to the interface and passed them as uniforms to the deferred shading pass, allowing the user to control the following parameters:

- The overall contributions of each lighting component.

- Whether to enable or disable the calculation of each lighting component, as shown in Figures 4, 5, 6.
- The cone aperture for tracing the soft shadows. Larger values can be used to estimate larger light sources.
- The light falloff exponent k relative to the $distance$ to the light. Light falloff is calculated as $1/(distance^k)$, so the default value of 2.0 is physically plausible.
- A multiplier for the specular cone aperture. Smaller values can make the scene highly reflective as shown in Figure 7.

3 Results

Figure 3 shows a series of screenshots of my final results with all of the lighting passes enabled. Whilst I am pleased with the outcome, there are also a number of visual artefacts that I discuss in Section 4. Figure 2 shows some performance metrics for the individual lighting passes at different voxel resolutions. Note that memory usage is high, even at low resolutions, as I opted to use dense 3D textures as opposed to sparse 3D textures. The time taken to perform the light injection pass also scales quite poorly at high resolutions because of this. Despite this, moving around the scene maintains a smooth 60 frames per second (limited by the OpenGL context), providing the user does not move the light. The voxelization and shading passes seem to scale very well with increased resolution, taking on average 2 and 4.5 miliseconds respectively at all of the tested resolutions.

Voxel Resolution	256^3	512^3	768^3
Frames per second	60	60	60
Memory usage	1063MiB	2615MiB	6628MiB
Voxelization	2ms	2ms	2ms
Light Injection	65ms	378.5ms	12294.5ms
Shading	4.5ms	4.5ms	4.5ms

Figure 2: *Averaged performance results when running with a NVIDIA GTX 1080 8GB at 2560x1080.*

4 Future Improvements

Unfortunately I wasn't able to implement a sparse data structure as described in the source material [Crassin et al. 2011] before the assignment deadline. Having a sparse data structure to store the emissive voxel texture would save a significant amount of GPU memory consumption, allowing the resolution for voxelization to be much higher; currently with dense 3D textures it is capable of exceeding both the 2 gigabyte memory limit of my GPU, and the 8 gigabyte limit of those at university.

The speed at which lighting rays are marched during the light injection pass would also be improved, as it would be able to step further through the voxel texture whilst ray marching if the higher levels in the octree are known to be empty.

I did start working on a sparse voxel octree as described in OpenGL Insights [Crassin and Green 2012] by using an OpenGL Atomic Counter for calculating the number of fragments needed for the voxel fragment list, and I believe I understand the technique well enough should I wish to implement it in the future.

The source material [Crassin et al. 2011] also describes a method for anisotropic mip-mapping of the 3D texture, storing a version of the texture for each directional axis, which improves visual accuracy when cone tracing. Whilst I could have implemented this with the dense 3D textures I was using, I opted to use OpenGL's built in mip-mapping to save time.

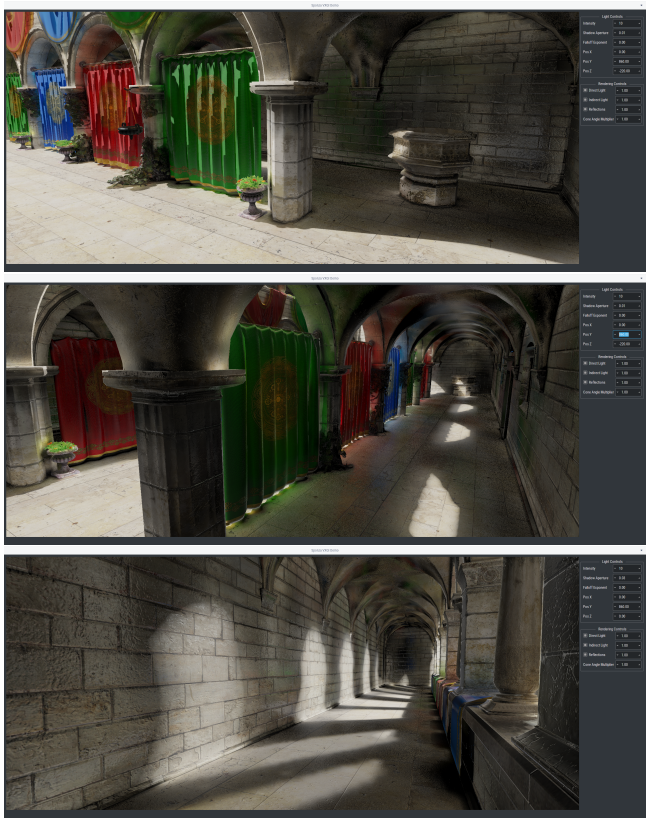

Figure 3: *Three screenshots showing the full scene with a voxel resolution of* 512^3.

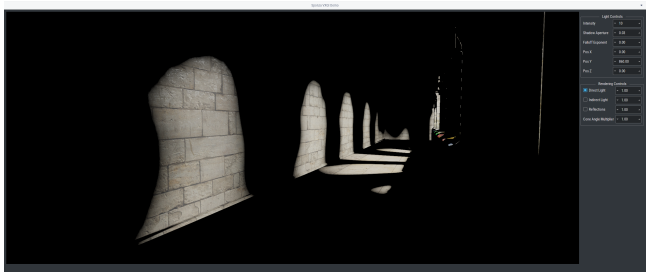

Figure 4: *A screenshot showing just the direct lighting component.*

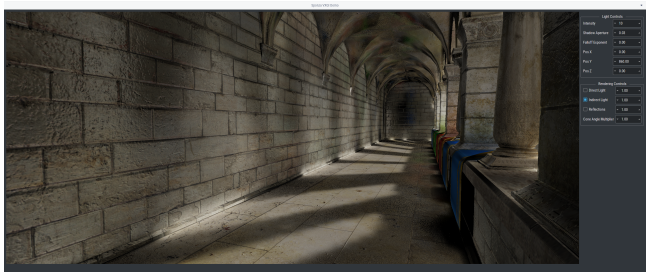

Figure 5: *A screenshot showing just the indirect lighting component.*

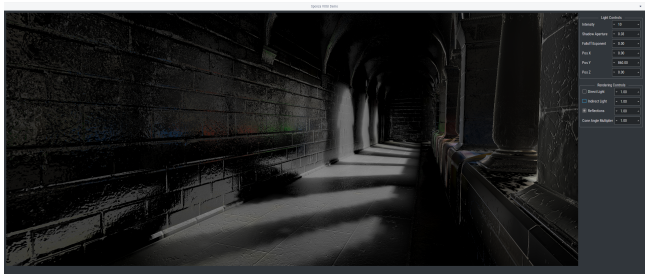

Figure 6: *A screenshot showing just the specular component.*

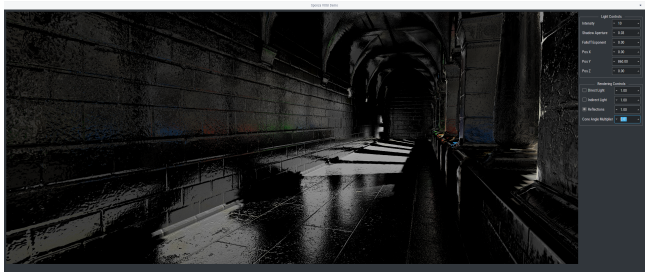

Figure 7: *A screenshot showing just the specular component, but with the minimum cone aperture.*

There are also a number of visual artefacts, the most noticable of which being the cracks in the voxelization as seen in Figure 8. I understand this to be due to incorrect conservative rasterization, however I have the NVIDIA *GL CONSERVATIVE RASTERIZATION NV* extension enabled and have confirmed it to be working, so I believe it could be an issue with using the extension whilst having multisampling enabled (as it is currently). I found that using a geometry shader to achieve the same result led to quite unusable artefacting.

Another artefact is quite visible noise in the indirect lighting component, as seen in Figure 9. I am not sure what causes this, though I suspect it could be because the hemisphere of indirect cones are aligned with the normal-mapped surface normal as opposed to the surface normal recieved from the geometry, which results in excessive variance between indirect lighting calculations for neighboring fragments. Banding artefacts are visible in the specular component, as seen in Figure 10, which occurs at certain viewing angles and at certain values for the specular aperture multiplier. I suspect this is a combination of excessive variance in surface normals, as mentioned previously, but could also be a result of limited numerical precision in calculating the specular component.

References

CRASSIN, C., AND GREEN, S. 2012. Octree-based sparse voxelization using the gpu hardware rasterizer. *OpenGL Insights*, 303. Accessed 20 Nov 2017.

CRASSIN, C., NEYRET, F., SAINZ, M., GREEN, S., AND EISEMANN, E. 2011. Interactive indirect illumination using voxel cone tracing. *Symposium on Interactive 3D Graphics and Games on - I3D 11*.

LIN, C.-T., 2013. Sparse voxel octree. `https://github.com/otaku690/SparseVoxelOctree`. Accessed 17 Nov 2017.

Figure 8: *A screenshot showing voxelization artefacts.*

Figure 9: *A screenshot showing noise in the indirect lighting.*

Figure 10: *A screenshot showing banding artefacts in the specular component.*

MACEY, J., 2014. Ngl the ncca graphics library. https://github.com/NCCA/NGL.

MACEY, J., 2017. Glsl physically based rendering - sponza demo. https://github.com/NCCA/PBR/tree/master/PBRSponza. Accessed 24 Oct 2017.

VILLEGAS, J., 2016. Vctrenderer. https://github.com/jose-villegas/VCTRenderer. Accessed 30 Nov 2017.

VRIES, J. D., 2016. Deferred shading. https://learnopengl.com/#!Advanced-Lighting/Deferred-Shading. Accessed 25 Oct 2017.

VRIES, J. D., 2016. Pbr lighting. https://learnopengl.com/#!PBR/Lighting. Accessed 11 Oct 2017.