Real-time Procedural Environments

Video demo:https://youtu.be/IYnwF6dtijg?si=dyTkm2M_YLJVc_QW

Abstract:

This project aims to develop a real-time application for procedurally generated terrain, vegetation, and lighting. The program will combine noise based terrain generation, L-system plant modeling, and voxel cone tracing for global illumination, with all parameters customizable through an interactive interface. My individual topic was implementing voxel cone tracing (aka VXGI) and other lighting related tasks to achieve real-time, physically convincing lighting.

Group technical summary:

Owan	I was responsible for building the entirety of the deferred renderer, lighting and related supporting systems, and to provide my teammates the tools to integrate their own work with little friction. These systems include the renderer and renderable system, where renderable is a simple interface designed for my teammates to implement/extend so that their geometry can be voxelized and rendered. In addition to this, I had also provided example shaders and classes to my teammates to help them with integrating their work.
Zaki	Zaki's responsibility was to create a procedural terrain system, and to provide plausible material and geometry information to the renderer. Due to the world space scale of Zaki's work, we had to work quite closely due to the limitations of the voxelizer in order to ensure his work was at a scale that fit the voxel bounds. Zaki was also responsible for providing Sophia with points in world space so that Sophia can place trees on top of the terrain.
Sophia	Sophia's responsibility was to create procedural vegetation via L-systems, create plausible materials and geometry for said vegetation, and to place the vegetation on the terrain using Zaki's provided world space points.

Introduction to Voxel Cone Tracing:

Objective:

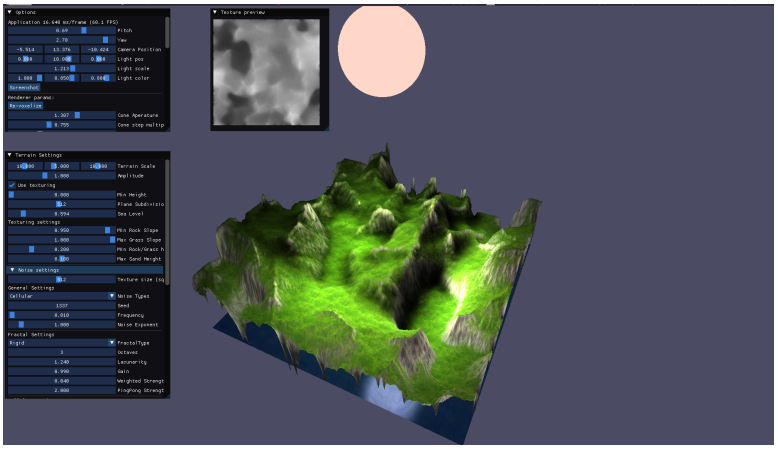
My objective was to light a scene in real time in a convincing manner. The problem I aimed to solve is to light a scene convincingly, due to the dynamic procedural nature of my teammates work lighting could not rely on slow baking techniques. The solution I landed upon was voxel cone tracing— a novel technique that had brief use around 2013— but eventually lost momentum due to its limitations.

Technical problems and tasks:

The first technical challenge I had was how to approach the voxelization process when my teammates are primarily using shaders to generate/translate their geometry, this means that voxelization cannot occur before my teammates' shaders get run. Another important part that also complicates this process is that voxelization 'renders' the scene from multiple views, meaning that view matrices need to be modified by the voxelizer without being overwritten, without also 'needing to know about' my teammates uniforms and related drawing logic.

Later on shadows rendered were incredibly dark, and lit surfaces were blown out. When approaching feedback from friends and peers regarding the extreme contrast, it was suggested the first lighting passes did not look convincing.

As rendering was my responsibility, enabling my teammates to properly evaluate and iterate on their own work would further improve the progression of the environment with time-availability. This involved tweaking a variety of rendering parameters supporting my teammates' work. For example, during an early iteration, Zaki had complaints about the terrain not being uniformly lit and it being difficult to evaluate his own work because of it. This is just an example of one of the many integration tasks I was responsible for ‘fixing’.



[Early iteration of VCT implementation]

Summary of background and related works:

Background context:

VCT efficiently simulates soft global illumination, glossy reflections and ambient occlusion in real time by voxelizing a scene, then sampling the voxels in order to estimate a fragments color using a hierarchical data structure for example mip-maps, where sampling from lower resolution mip-map represents a wider segment in a cone. Despite its advantages, the method remains memory intensive and can be difficult to integrate due to its voxelization and mipmapping requirements. Effectively VCT provides a middle ground between traditional rasterization based rendering and full ray tracing, it has seen brief use in some game engines such as unreal engine 4 under the name ‘Sparse voxel octree global illumination’ (SVOGI) but has since been somewhat abandoned due to the extreme VRAM usage, light leaking, scene size constraints—meaning that the technique is only viable for small mostly static environments.

Related works / references:

- C. Crassin, F. Neyret, M. Sainz, S. Green, E. Eisemann (2011)
  Interactive Indirect Illumination Using Voxel Cone Tracing. NVIDIA Technical Report
  https://research.nvidia.com/sites/default/fil es/publications/GIVoxels-pg2011-authors.pdf
  C. Crassin
  NVIDIA GTC Slides (2012)
  https://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/SB134-Voxel-Cone-Tracing-Octree-Real-Time-Illumination.pdf
- Fredrik Prantare (2013)
  A real-time global illumination implementation using voxel cone tracing. Implemented in C++ and GLSL.
  https://github.com/Friduric/voxel-cone-tracing

Summary of my contribution:

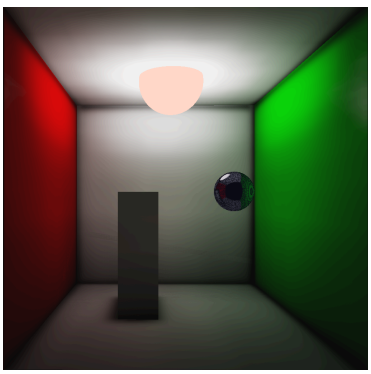
My contribution to the project was to the entirety of the renderer, be it supporting systems or examples for my teammates on how to integrate. Outside of technical contributions, I played a major role in organizational tasks such as meeting deadlines, setting up team meetings and helping my teammates integrate their work. The files I was responsible for writing the majority of go as follows:

cubeRenderable.cpp	voxelizer.hpp	point_light_frag.glsl
example_renderable.cpp	cube_vert.glsl	voxelizer.cpp
point_light_renderable.cpp	cube_frag.glsl	voxel_debug_frag.glsl
renderable.hpp	example_vct_compatible_frag.glsl	lighting_pass_frag.glsl
renderer.hpp	example_vct_compatible_vert.glsl	gBufferPrepass.hpp
gBufferLightingPass.hpp	fullscreen_quad_vert.glsl	Application.cpp

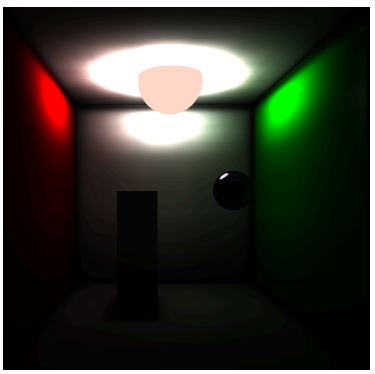
Solutions and implementation details:

As mentioned before, a core technical problem was how to handle voxelization and lighting cleanly, whilst abstracting away the complexity of the renderer from my teammates. My solution for this was the creation of a renderer and renderable system in conjunction with a deferred renderer. The way it works is my teammates extend a renderable interface, where they would set their uniforms and make draw calls as would normally be done. However the interface allows the renderer to manipulate the projection and view matrices allowing for the voxelization process to occur. Voxelization and the writing to the gBuffer is done by a singular GLSL function I provide to my teammates, which takes material properties, normals and position information and either writes it to the gbuffer or voxel texture depending on what pass is occurring. This system also allows my renderer to not need to know about the complexities of my teammates work, since draw calls and the setting of their own uniforms was handled by them.

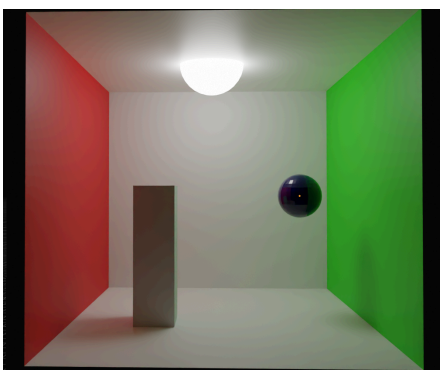
The extreme contrast issue was solved with a few different solutions. The first one that was tried was scaling the primary light source to be more of a disc, resulting in more uniform lighting. The second solution was the addition of filmic tone mapping, which mellowed out the extremes of the image. The third solution was modifying contrast directly in the final fragment shader, since even after tone mapping, the contrast was still too strong when compared to ‘ground truth’ Blender renders. For reference, here is a before and after (minus light scaling) with a Blender render for reference:



[VCT]

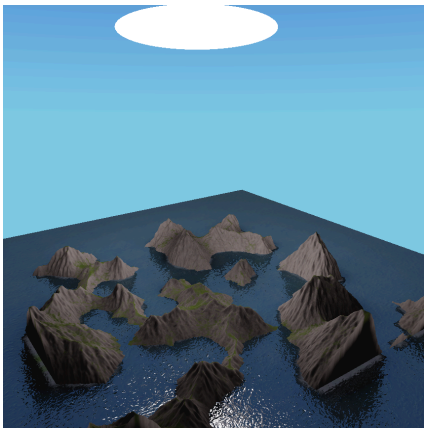


[VCT no tonemap & contrast]



[Blender render]

Results and what was achieved:



When compared to the ‘ground truth’ the results are not ideal. However, considering the relative simplicity of this technique compared to serious realtime 3d renderers for example those used in modern game engines, it achieves quite a lot visually. The features the renderer supports go as follows:
1 bounce global illumination, Indirect diffuse lighting using a monte carlo approach and cosine weighted hemisphere sampling, geometry reflections using cone aperture randomness to hide the inherent blockyness of said reflections, specular highlights, fresnel effect using schlicks approximation, metallic / dielectric materials, ambient occlusion derived from diffuse cone transmittance, procedural gradient sky, filmic tone mapping, contrast adjustments and a variety of debug views.
You may view a video demo HERE.

[Render as of 14/10/25]

Discussion:

Limitations
The voxelizer uses a simple technique of rendering from multiple views in order to capture the geometry, however this is flawed in that sometimes there are gaps and geometry that does not get captured perfectly, so tweaking the voxelizer parameters before voxelizing is often needed in order to achieve
Because of this, lighting settings often need to be tweaked on geometry changes as well, otherwise the scene may look blown out, flat or too dark or banding artifacts may be visible.
I opted for a simple 3d texture + mip-maps instead of sophisticated data structures such as sparse voxel octrees. The renderer is incredibly memory intensive, disallowing me to increase voxel resolution any further than it currently is, being 512^3. Another downside of the voxelization pass is the very slow mip-map generation, without mip-map generation, voxelization is fast enough to occur every frame, however the mip-map generation results in second long slowdowns forcing this process to only occur on demand, added as an Imgui button. Due to these limitations, the program is only runnable with GPU’s that have sufficient memory, ~2gb.
Strengths:
Despite its limitations, when the parameters are ‘dialed in’ the results are quite surprising considering the relative simplicity of the technique. Achieving convincing color bleed and reflections without the complexities of ‘faking’ these attributes using things such as reflection probes. Another strength of my implementation is world space awareness, many realtime global illumination techniques are reliant on screen space techniques, resulting in offscreen light sources not being accounted for, this isn't the case for VCT that has knowledge of the scene offscreen.
Integration:
I integrated my work as early as possible, creating the core systems and interfaces for my teammates to then build upon when integrating their work. Integration was for the most part relatively smooth, though I had to constantly make adjustments to the renderer parameters in order for the new changes to be lit as expected. The only intrusive aspects of integrating my teammates' work with mine, was the mentioned GLSL function and uniforms that they needed to add to each of their shaders, and the restrictive nature of my teammates conforming their work to my renderable interface. I did have to make some adjustments to my systems to better suit my teammates, for example adding support for renderables to be able to use multiple shaders at once. As for physical accuracy, a ground truth render was done late into development, had this been done earlier it may have been useful for highlighting the renderers visual weaknesses. Cornell box ground truth comparisons will definitely be something I use for any future graphics work, as to steer my work away from inaccuracy.