Real-Time Global Illumination Using Voxel Cone Tracing

Team 02

RAMANJEET SINGH and ATHARV GOEL

1 INTRODUCTION

The problem of real-time global illumination has long been a critical challenge in computer graphics with a wide-reaching impact, owing to its potential to elevate the quality of interactive 3D content across various domains. It bridges the gap between the visual fidelity of offline rendering and the real-time interactivity required in applications like gaming, simulations, and design.

Traditionally, methods like path tracing and photon mapping have been widely popular for their high-quality photorealistic rendering. However, these are unsuitable for real-time applications. To this end, modern solutions have emerged focusing on image-space rendering for triangle-based scenes and voxel-space methods for capturing off-screen information.

This project seeks to solve the real-time global illumination problem using a voxel cone tracing approach to approximate indirect lighting. This approach has been proved to be highly effective by Cyril Crassin et al.[2], which this project will closely reference.

2 LITERATURE REVIEW

Real-time global illumination (GI) in 3D scenes often involves rendering meshes comprised of triangles, where lighting for each pixel is recalculated for each frame. These algorithms inherently operate in triangle space, utilizing ray-triangle intersection tests to determine pixel illumination. However, the computational complexity of such triangle-based approaches becomes prohibitive as the scene complexity grows with its increasing number of triangles and vertices.

There are well-established off-line solutions for accurate global-illumination computation, such as path tracing [4] and photon mapping [8]. These have been extended with optimizations that often exploit geometric simplifications [5][1], or hierarchical scene structures [9], but do not achieve real-time performance.

To address this challenge, alternative adaptive and efficient solutions have emerged. Some of the most efficient real-time GI algorithms for rendering triangle-based scenes operate in image-space and disregard off-screen information [7]. Meanwhile, a range of promising real-time algorithms for voxel-based global illumination work in voxel-space and consider off-screen information [6][2].

One important milestone in this journey was the breakthrough made by Thiedemann and colleagues in 2011. Their approach combined path tracing with an atlas-based boundary voxelization technique, an early step towards voxel-based global illumination. Although it could only handle a subset of indirect lighting effects, mainly near-field single-bounce illumination, it established a promising example for future progress. [6].

The influential work by Crassin et al., "Interactive Indirect Illumination Using Voxel Cone Tracing"[2] utilized the power of GPUs to create filtered mipmaps for real-time voxel-based lighting representations. They

Authors' address: Ramanjeet Singh, ramanjeet21085@iiitd.ac.in; Atharv Goel, atharv21027@iiitd.ac.in.

introduced a sparse voxel octree to improve rendering performance by reducing sample counts and minimizing step distances during cone tracing. They also developed a new method for injecting light called splatting, further enhancing the algorithm's capabilities.

The Crassin et al. algorithm notably supported various lighting effects at interactive frame rates, including glossy reflections, ambient occlusion, and indirect diffuse lighting. This approach's versatility and ability to perform well regardless of the scene represented a significant advancement in real-time global illumination. The efficient octree-voxelization scheme also allowed for the real-time rendering of intricate and complex scenes.

3 MILESTONES

The implementation for this project has been divided into rough milestones as follows.

S. No.	Milestone	Member	Status
	Mid evaluation		
1	Voxelize the triangle mesh-based scene	Raman	Done
2	Inject direct lighting into the voxels	Raman	Done
3	Apply linear filtering to generate 3D mipmaps	Atharv	Done
	Final evaluation		
4	Generate diffuse cones to accumulate light rays	Atharv	Done
5	Implement specular cones for glossy materials	Raman	Done
6	Implement voxel cone tracing to compute ambient occlusion	Raman	Done
7	Adapt the ambient occlusion computation for indirect illumination	Atharv	Done

4 APPROACH

4.1 Overview

In real-time scenarios, it is crucial to have fast 3D scan conversion (voxelization) of traditional triangle-based surfaces. By utilizing OpenGL, a surface voxelization algorithm can create a 3D texture through GPU hardware rasterization and the image load/store interface. The process involves rasterization, selecting the dominant axis based on triangle normals, and performing an orthographic projection. Fragments generated during the rasterization process are used to calculate intersected voxels, resulting in voxel fragments with 3D coordinates and direct diffuse Phong lighting components. These fragments are then directly written to the destination 3D texture using OpenGL's *imageStore()* operation.

4.2 Voxelization

We used the efficient implementation of a surface voxelization algorithm using OpenGL described in [3]. The algorithm generates a regular 3D texture through the utilization of the GPU hardware rasterizer and the image load/store interface introduced in OpenGL 4.2. The implementation aims to achieve real-time processing, particularly in scenarios where rapid voxelization of triangle-based surface data is essential.

4.2.1 Rasterization. We used the GPU's hardware rasterization pipeline to generate the voxels. The key idea is to test whether a triangle B's plane intersects a given voxel (V) and whether the 2D projection of B along the dominant axis of its normal (one of the three main axes of the scene that provides the largest surface for the projected triangle) intersects the 2D projection of V. We implemented conservative rasterization to ensure that

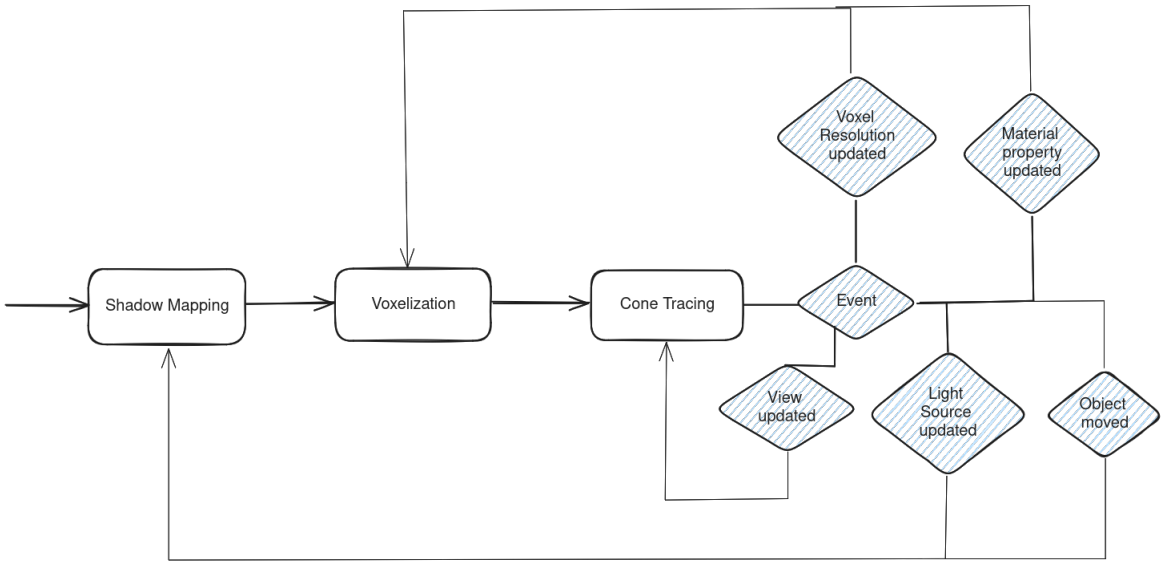

Fig. 1. Render loop

small triangles are not dropped out of voxel information. This requires an Nvidia GTX9x0 or better GPU. Our voxelization scheme defaults to non-conservative raster for unsupported hardware.

4.2.2 Dominant Axis Selection. Each triangle of the mesh undergoes orthographic projection along the dominant axis of its normal. This dominant axis is selected on a per-triangle basis within a geometry shader, where information about the triangle's vertices is available.

To determine the dominant axis, we calculate the maximum value for $l_{\{x,y,z\}} = |n.v_{\{x,y,z\}}|$, where n represents the triangle normal and $v_{\{x,y,z\}}$ denotes the three main axes of the scene. Once the axis is chosen, the projection along this axis is performed using orthographic projection, also calculated inside the geometry shader.

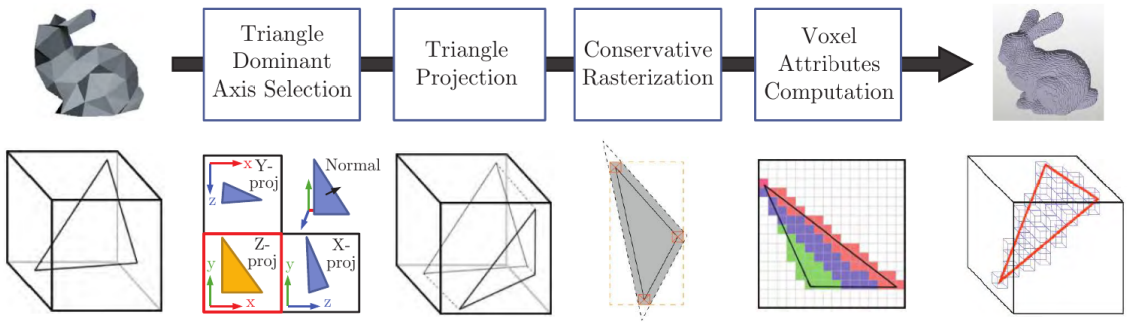

Fig. 2. Illustration of our simple voxelization pipeline. Diagram sourced from [3].

4.2.3 Store Direct Lighting in Voxel Grid. We now take each projected triangle and pass it through the standard setup and rasterization pipeline to perform 2D scan conversion (Figure 1). The 2D viewport resolution is adjusted to match the lateral resolution of the voxel grid (for example, $512x512$ pixels for a 512^3 voxel grid), to obtain fragments that correspond to the 3D resolution of the voxel image.

The algorithm relies on accessing the image to write data into the voxel grid. Therefore we also disable framebuffer operations, including depth writes, depth testing (glDisable(GL_DEPTH_TEST)), and color writes (glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE)) as they are not required. For each 2D fragment, the intersected voxels are calculated within the fragment shader using position and depth information that is interpolated from the vertex values at the pixel center.

This information is used to create voxel fragments, which are representations of voxels that are intersected by a specific triangle. Each voxel fragment contains a 3D integer coordinate within the destination voxel grid and one attribute value: the direct diffuse Phong component of that fragment.

To complete the process, the voxel fragments are directly written from the fragment shader to their corresponding voxel in the destination 3D texture. This is done using the *imageStore()* operation introduced in OpenGL 4.2.

4.3 Voxel Cone Tracing

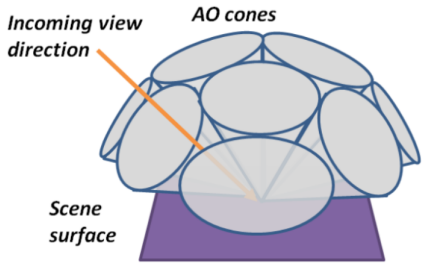

Fig. 3. Partitioning the integral into a set of cones. Diagram sourced from [2].

4.3.1 The Algorithm. The hemispherical integration in the rendering equation can be represented as a summation of N cones:

$$L_o(\mathbf{p}, \omega_o) = L_e(\mathbf{p}, \omega_o) + \int_\Omega f(\mathbf{p}, \omega_i, \omega_o) \cdot L_i(\mathbf{p}, \omega_i) \cdot (\mathbf{n} \cdot \omega_i)\, d\omega_i \tag{1}$$

$$L_o(\mathbf{x}, \omega) \approx L_e(\mathbf{x}, \omega) + \frac{1}{N} \sum_{i=1}^{N} V_c(\mathbf{x}, \omega_i) \tag{2}$$

where, for a regular partition, each $V_c(\mathbf{x}, \omega_i)$ resembles a cone.

In the process of volume ray marching, the cone's direction is determined by iterative sampling steps. Each step involves sampling the surrounding volume to find surfaces and radiance. The sampled volume is adjusted dynamically based on the cone's diameter at each specific point. This calculation is done using trigonometry and the aperture angle.

Unlike traditional ray tracing, volume ray marching continues through objects, sampling along its path until a specified occlusion threshold is reached. Occlusion values are important as they track how much of the cone's volume is blocked by individual surfaces along its path. The final result of the traced cone, is determined by

the accumulated color value (c) and occlusion value (α) sampled at each step. These variables are continuously updated through front-to-back accumulation throughout the entire process.

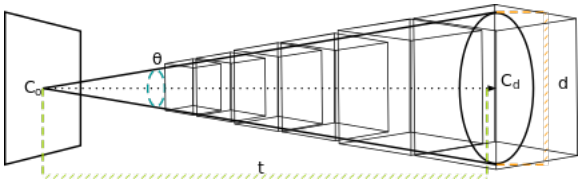

Fig. 4. Visual representation of a cone used for cone tracing[2].

In our approach, each cone is characterized by an origin (C_o), a direction (C_d), and an aperture angle (θ). As we progress through the cone steps, we define the diameter (d) based on the traced distance (t) using the equation $d = 2t \times \tan(\frac{\theta}{2})$. To determine the mipmap level for sampling, we use $V_{\text{level}} = \log_2(\frac{d}{V_{\text{size}}})$, where V_{size} is the size of a voxel at the maximum level of detail.

Following Crassin's approach, for each cone trace, we actively track the occlusion value (α) and the color value (C), representing the indirect light toward the cone origin (C_o). At each step, we retrieve the occlusion value (α_2) and outgoing radiance (C_2) from the voxel structure. The C and α values undergo continuous updates using volumetric front-to-back accumulation: $C = \alpha C + (1 - \alpha)\alpha_2 C_2$ and $\alpha = \alpha + (1 - \alpha)\alpha_2$. To maintain integration quality between samples, we adjust the distance (d') between steps by a factor (β), where $\beta = 1$ preserves the current diameter (d), and values less than 1 enhance quality at the expense of increased sampling and reduced performance.

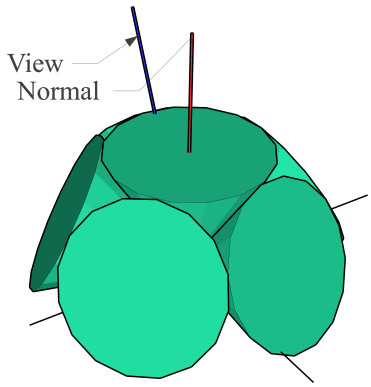

Fig. 5. Diffuse Cones

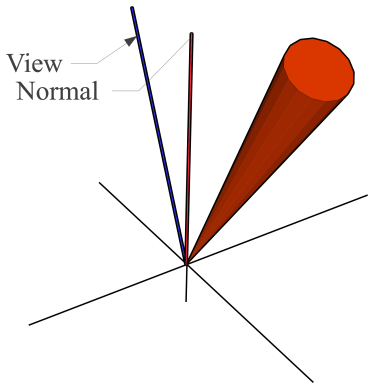

Fig. 6. Specular Cones

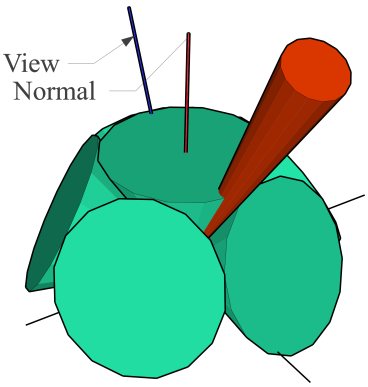

Fig. 7. BRDF

4.3.2 Indirect Diffuse Lighting. We applied the above-discussed algorithm to compute indirect diffuse lighting at a fragment. Typically for Phong-like BRDFs, a few large cones (~5) estimate the diffuse energy coming from the scene, while a tight cone in the reflected direction with respect to the viewpoint captures the specular component [2]. Figure 7 shows the BRDF cones.

We trace 5 diffuse light accumulation cones from every fragment. 1 forward cone is traced in the direction of the normal, and 4 side cones each rotated 45 from the normal toward the surface. All cones are given an aperture of 75°. Figure 5 shows the diffuse cones.

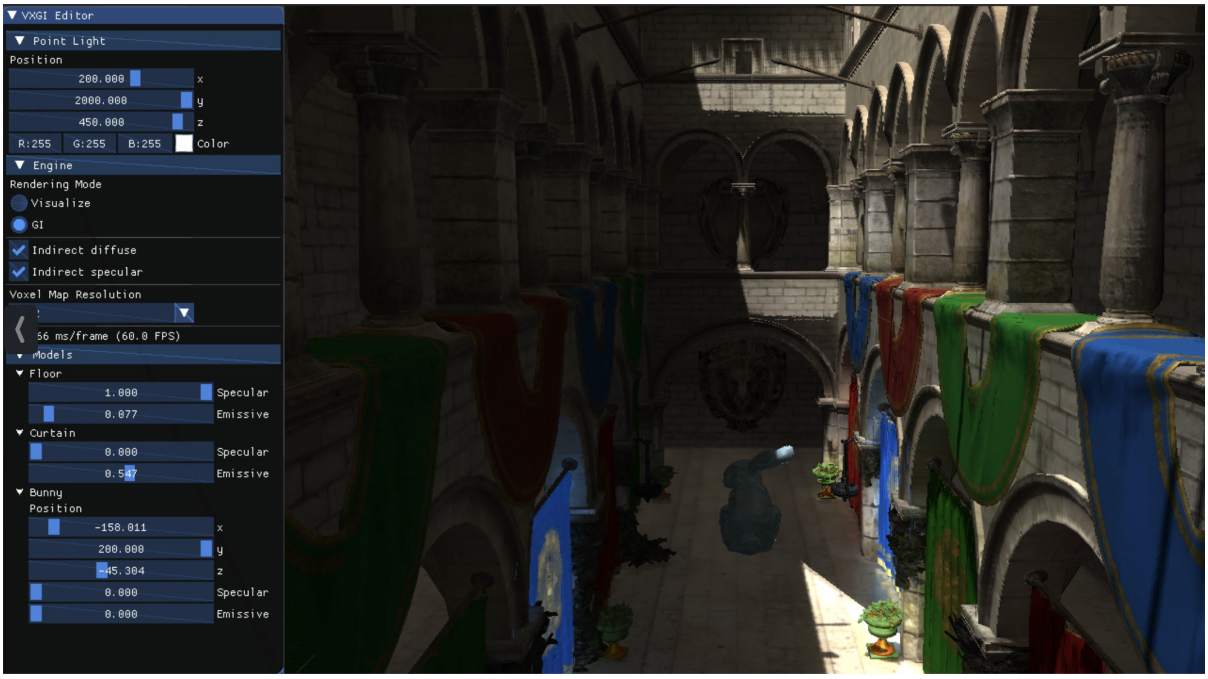

Fig. 8. A screenshot of the Crytek Sponza scene rendered at 1080p. A single point light source is used, and global illumination is approximated in real-time. The scene also features a dynamic Stanford Bunny on the floor.

4.3.3 Indirect Specular Lighting. The indirect specular light is accomplished using a perfectly reflected cone in the direction of the reflected view vector. We chose an aperture value of 15° for the specular cones. Figure 6 shows the specular cones.

4.3.4 Emissive Materials. Emissive materials are easy to simulate. We don't compute shadows for them during the voxelization stage. This automatically ensures that the material will illuminate its surrounding areas because of the direct lighting stored in the voxel grid.

5 RESULTS

The approach described in Section 4 yielded promising results. We rendered the Crytek Sponza scene with our engine and achieved satisfactory performance. Figure 8 shows a screenshot from an interactive, 3D scene running in real-time.

Our UI interface conveniently allows modifying various object properties, light source properties, voxelization resolution, dynamic mesh position, dynamic mesh properties and much more. One can try around several different settings to change the scene aesthetics.

We also implemented the ability to visualize the voxels. Unit cubes are scaled and translated to a unique position corresponding to an index in the voxel map. To render the complete scene using voxels, $VOXEL\_DIM^3$ unit cubes are rendered by using **glDrawArraysInstanced()**. The correct index of each cube is computed in the vertex shader by using the **gl_InstanceID** shader global attribute.

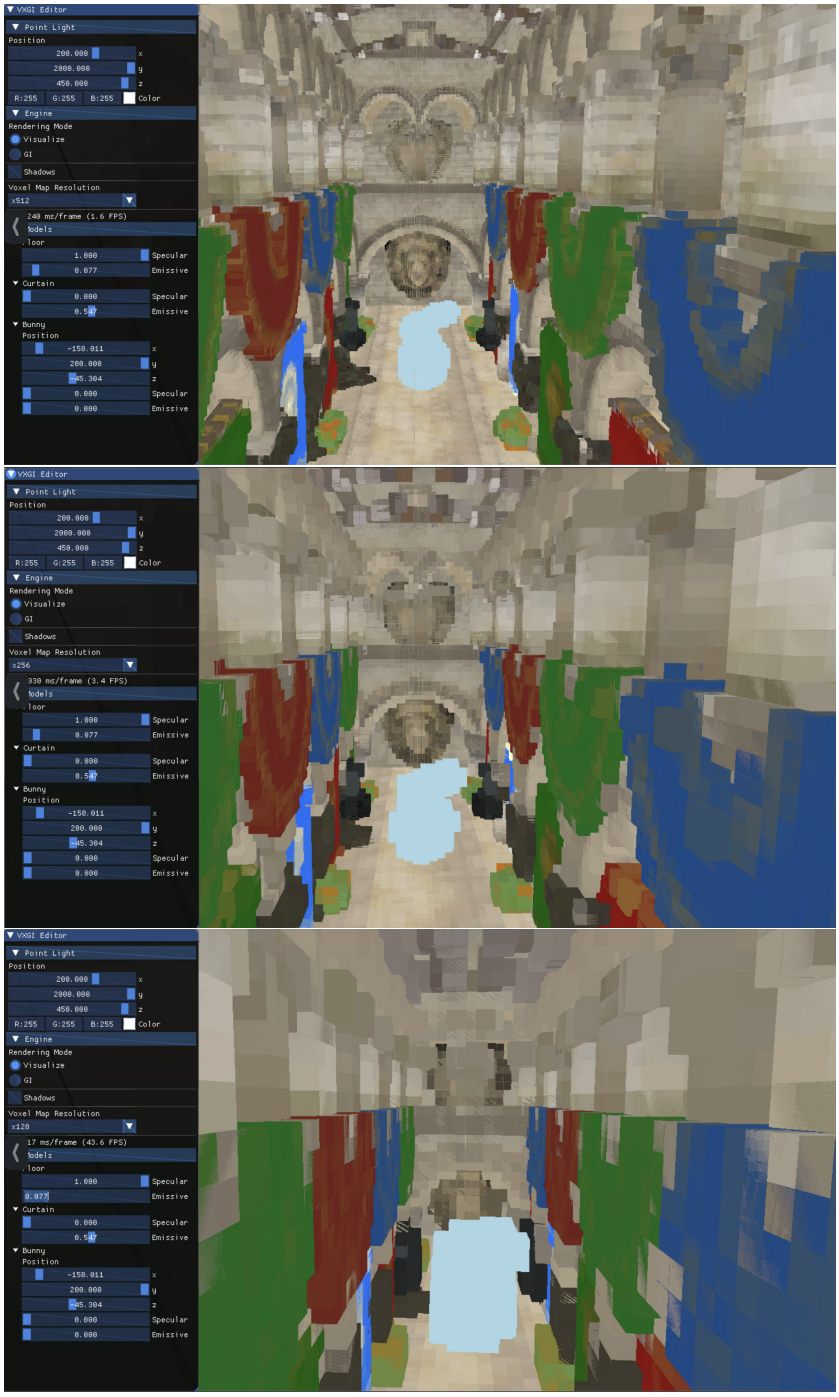

Fig. 9. Voxelization of the scene from the same position and camera angle. The three figures visualize the scene with voxel resolution set to 512, 256 and 128 respectively.

REFERENCES

[1] Batali D. Christensen P. H. 2004. An irradiance atlas for global illumination in complex production scenes. *EGSR* (2004).
[2] Miguel Saintz Simon Green Elmar Eisemann Cyril Crassin, Fabrice Neyret. 2011. Interactive indirect illumination using voxel cone tracing. *Pacific Graphics* (2011).
[3] Simon Green Cyril Crassin. 2012. *OpenGL Insights*. Taylor Francis Group.
[4] Kajiya J. T. 1986. The rendering equation. *SIGGRAPH* (1986).
[5] Lamorlette A. Tabellion E. 2004. An approximate global illumination system for computer generated film. *ACM Trans. Graph.* (2004).
[6] Grosch T. Muller S. Thiedemann S., Henrich N. 2011. Voxel-based global illumination. *ACM* (2011).
[7] Hans-Peter Seide Tobias Ristchel, Thorsten Grosch. 2009. Approximating dynamic global illumination in image space. *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games* (2009).
[8] Jensen H. W. 1996. Global illumination using photon maps. *EGSR* (1996).
[9] Pan M. Bao H. Wand R., Zhou K. 2009. An efficient gpu-based approach for interactive global illumination. *ACM Trans. Graph* (2009).