4-23-2024

Volume Ray Casting for Special Effects

Jesse Cole Rheal
University of Alabama in Huntsville

Volume Ray Casting for Special Effects

by

Jesse Cole Rheal

An Honors Capstone

submitted in partial fulfillment of the requirements

for the Honors Diploma

to

The Honors College

of

The University of Alabama in Huntsville

4/23/2024

Honors Capstone Project Director: Dr. Timothy Newman

Jesse Rheal 4/23/2024

Student (signature) Date

Timothy S. Newman 4/24/2024

Project Director (signature) Date

Letha Etzkorn Digitally signed by Letha Etzkorn Date: 2024.04.24 15:10:28 -05'00'

Department Chair (signature) Date

Sean Lane Digitally signed by Sean M Lane Date: 2024.04.24 16:41:58 -05'00'

Honors College Dean (signature) Date

HONORS COLLEGE
THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Honors College

Frank Franz Hall

+1 (256) 824-6450 (voice)

+1 (256) 824-7339 (fax)

honors@uah.edu

Honors Thesis Copyright Permission

This form must be signed by the student and submitted with the final manuscript.

In presenting this thesis in partial fulfillment of the requirements for Honors Diploma or Certificate from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department, Director of the Program, or the Dean of the Honors College. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

Jesse Rheal .
Student Name (printed)

Jesse Rheal
Student Signature

4/23/2024
Date

Table of Contents

Dedication

To my father who always encouraged me to do more.

Abstract

The techniques that fall under volume rendering, such as volume ray casting, have been widely used to visualize three-dimensional medical scan (e.g., CT scan) data. Furthermore, in entertainment media, they've sometimes been used as part of processes that simulate environmental effects such as clouds, fog, and sunbeams. Key findings from our project, which explores computational efficiency issues for volume ray casting, are described here. The project uses a framework based on OpenGL, a popular API for programming graphics processors. The report here focuses on differences in performance between the CPU and GPU when performing ray calculations. Results indicate that on average the GPU implementation was 7.18 times faster than the multithreaded CPU implementation and 59.3 times faster than the best single-threaded CPU implementation.

Introduction

Volume ray casting uses rays to render translucent, three-dimensional points in space. It is different from rasterization which transforms vertices into–typically opaque–surfaces by lighting pixels within the bounds of the vertices, whereas volume ray casting determines every pixel's value by sending rays through the scene. It is also different from ray tracing, a method of realistically rendering surfaces of objects by simulating the movement of light though spawning secondary rays from rays that collide with objects. Volume ray casting is concerned with rendering volumes as transparent structures, not just their surfaces; it doesn't simulate the movement of light; in other words, it doesn't spawn secondary rays.

The class CS445, Intro to Computer Graphics, was used as the basis for this project. The class covered the history of graphics techniques that developed from the first drawing program, Ivan Sutherland's 1963 Sketchpad, to the modern day [1]. In the class, graphics programs were written in C++ and OpenGL to learn techniques such as transformations, projection, lighting, blending, et cetera. The aforementioned techniques were critical in this project's development, but volume rendering topics were not covered.

Therefore, research outside of the classroom on volume rendering techniques was imperative. Volume ray casting works by sampling, interpolating, and lighting points along rays that move through the scene and somehow compositing those points into final pixel values. The techniques to achieve volume ray casting include gradient calculation, image-order and object-order, composition, and transfer functions. Firstly, the gradient is a vector that indicates the magnitude and direction of change in a dataset [2]; it is used in place of a normal vector in traditional raster rendering. Image-order means that the rays propagate out of the screen towards the scene, and object-order means rays start from the scene and move towards the screen [3].

Compositing is the method of collapsing the ray into a final value [2]. Finally, transfer functions map dataset values to color values [3], however they were not used in this project. That covers all the volume ray casting techniques that were used in this project.

Chapter 1: Volume Rendering Approach

The natural complexity of volume rendering has led to many techniques being developed over the years. There are more decisions to be made on how to represent a volume on a two-dimensional surface as opposed to representing solid primitives. This chapter outlines three key decisions made for this project which include data representation, gradient algorithm, and image-order technique.

Data Representation

In volume rendering, the underlying data being rendered can represent many things. A popular use case for volume rendering is for visualizing medical data [3]. Different medical imaging machines–such as MRI, CT, and PET–output three, or even four, dimensional data. The data represents properties of the imaged object. Those properties can be anything such as density, heat, color, or pressure.

The data's underlying property in this project is ambiguous as it is not real data. For all intents and purposes, the property can be thought of as brightness. Indeed, the scalar property represents the red, green, blue, and alpha values of the voxels equally. Therefore, lower values are darker and more transparent than lighter values.

Furthermore, the data in this project is said to be isotropic; each data point is equally spaced along three orthogonal axes [3]. This simplification provides many benefits. For one, the positional part of the data is not explicitly stored in memory. Instead, the position of the voxel is derived from its position in the buffer. Each voxel is mapped to an integer position in three dimensional cartesian coordinates. Another benefit is that it's quite simple to find the nearest voxels given an arbitrary floating point position in three dimensional space. A single rounding

operation and a couple of integer additions are all that are needed. Interpolation of voxel values also becomes much simpler which is discussed in Chapter Four.

Gradient Algorithm

Lighting a surface in graphics must take into consideration which way the surface is facing. This is normally done with the normal, a unit vector that is orthogonal to the planar surface. However, in my implementation of volume rendering, there are no surfaces. Instead, there are single points in space. The gradient represents how a point is oriented given information about its neighbors. More specifically, it is the change in the property value along each orthogonal axis represented as a unit vector.

The algorithm for the gradient used in this work is called the central difference gradient estimator [2]. It is a cheap and simple way of calculating the gradient.

A gradient vector $\widehat{g}$ as defined by Lichtenbelt follows:

$$g_x = f(x - 1, y, z) - f(x + 1, y, z)$$

$$g_y = f(x, y - 1, z) - f(x, y + 1, z)$$

$$g_z = f(x, y, z - 1) - f(x, y, z + 1)$$

"Where $f(x, y, z)$ is the value of the voxel at position (x, y, z) in the data set" [2]. The vector is then normalized. The gradient vector is used in lieu of a normal vector in lighting calculations.

Image-Order Technique

For every pixel on the drawing canvas, there is a ray associated with it. The ray projects from the screen into the scene while sampling values at regular intervals. Those values are then summed to become the color of the ray's associated pixel. This method is a subset of the image-order techniques called x-ray rendering [3].

There is no global illumination. In other words, the program makes no attempt to simulate how light moves through the volume. Instead, each sample point only takes into account the value at the point, its gradient, and the light. Each sample point is said to be locally illuminated. This technique results in less appealing special effects, but it is much more performant.

Chapter 2: Ray Casting Pipeline

This chapter outlines all the steps taken to go from three-dimensional data to colors on the screen. Those steps include ray sampling, voxel interpolation, sample point lighting, and ray compositing.

Ray Sampling

A ray is a point that has a direction vector. Each pixel on the canvas has an associated ray. The ray starts at the pixel's position in world coordinates. Then, the ray moves in the direction the viewer is looking while sampling points at a regular rate. Each ray moves parallel to each other which results in an orthographic viewing projection.

Voxel Interpolation

When the ray samples a point, the point is a floating point vector that doesn't have an associated property value or gradient. Both need to be interpolated from surrounding voxels. There are many different ways to interpolate said values. Some techniques include nearest neighbor, bilinear interpolation, trilinear interpolation, et cetera.

The technique used in this work is known as Barycentric interpolation. In Bán's words, "Barycentric interpolation on an n-dimensional simplex is a linear combination of the vertices where the weights sum to one" [4]. A simplex is a polytope with the smallest number of vertices possible in its dimension. Simply put, a simplex in two dimensional space is a triangle, and a simplex in three dimensional space is a tetrahedron. Therefore, doing Barycentric interpolation in three dimensional space assumes interpolation between four vertices which make up a tetrahedron. However, each voxel is made of eight vertices. In order to do Barycentric interpolation, each voxel must be broken down further into six mutually exclusive tetrahedra.

The tetrahedron the sample point is inside of can be determined through inequality conditions where x, y, and z are localized floating-point–between 0 and 1–coordinates of the sample point:

$$x > y,\ y \geq z,\ z \geq x$$

Once the tetrahedron is determined, it becomes a matter of breaking the problem down into a system of linear equations. Once those linear equations are solved, what's left are four non-negative real values that add up to one. Each value is a Barycentric weight which determines how much of each vertex of the tetrahedron to use. The weights multiply the vertex attributes to determine the interpolated property value and gradient of the sample point.

Sample Point Lighting

Once the property value and gradient of the sample point are determined, they are used in a Blinn-Phong lighting calculation. The Blinn-Phong lighting model is arguably the most popular model used in computer graphics. It is a combination of lighting properties including ambient, diffuse, and specular [1]. Ambient light is light that affects every surface evenly no matter the direction the object is facing. Diffuse lighting is light that reflects equally in all directions which takes into account the orientation of the object. Finally, specular lighting is light that is more intense when the viewer is looking directly at it.

Ray Composition

After each sample point's color has been determined via Barycentric interpolation and Blinn-Phong lighting, each value is simply summed along the ray to become the final pixel color value on the canvas. As stated earlier, this results in an x-ray-like model. It most closely

resembles how x-ray data is rendered [3]. For more realistic special effects, a better model would take into consideration how light moves about the volume.

Chapter 3: Technology Used

Languages

The project utilizes C++ and GLSL for its programming languages. C++ is a systems language known for its performance and low level capabilities. GLSL stands for OpenGL Shading Language and is used for writing code that runs on GPUs.

Libraries

The libraries include GLEW, GLUT, GLM, and “siv::PerlinNoise.” GLEW, the OpenGL Extension Wrangler Library, is used for OpenGL bindings. GLUT, the OpenGL Utility Toolkit, is used for general input and output with the user’s devices. GLM, OpenGL Mathematics, is a useful mathematics library for vector and matrix manipulation that closely follows GLSL syntax. Finally, “siv::PerlinNoise” is a single header perlin noise generator.

Hardware

All benchmarking was done with a Radeon RX 580 8GB GPU and a Ryzen 5 2600 CPU on Ubuntu 22.04.3 LTS.

Chapter 4: Results

The scene rendered is 60^3 data points of three-dimensional Perlin noise. The cube is scaled up by a factor of 4 by having each ray spaced 0.25 units apart in both the x and y directions. Different sample rates were taken to see differences in quality and performance.

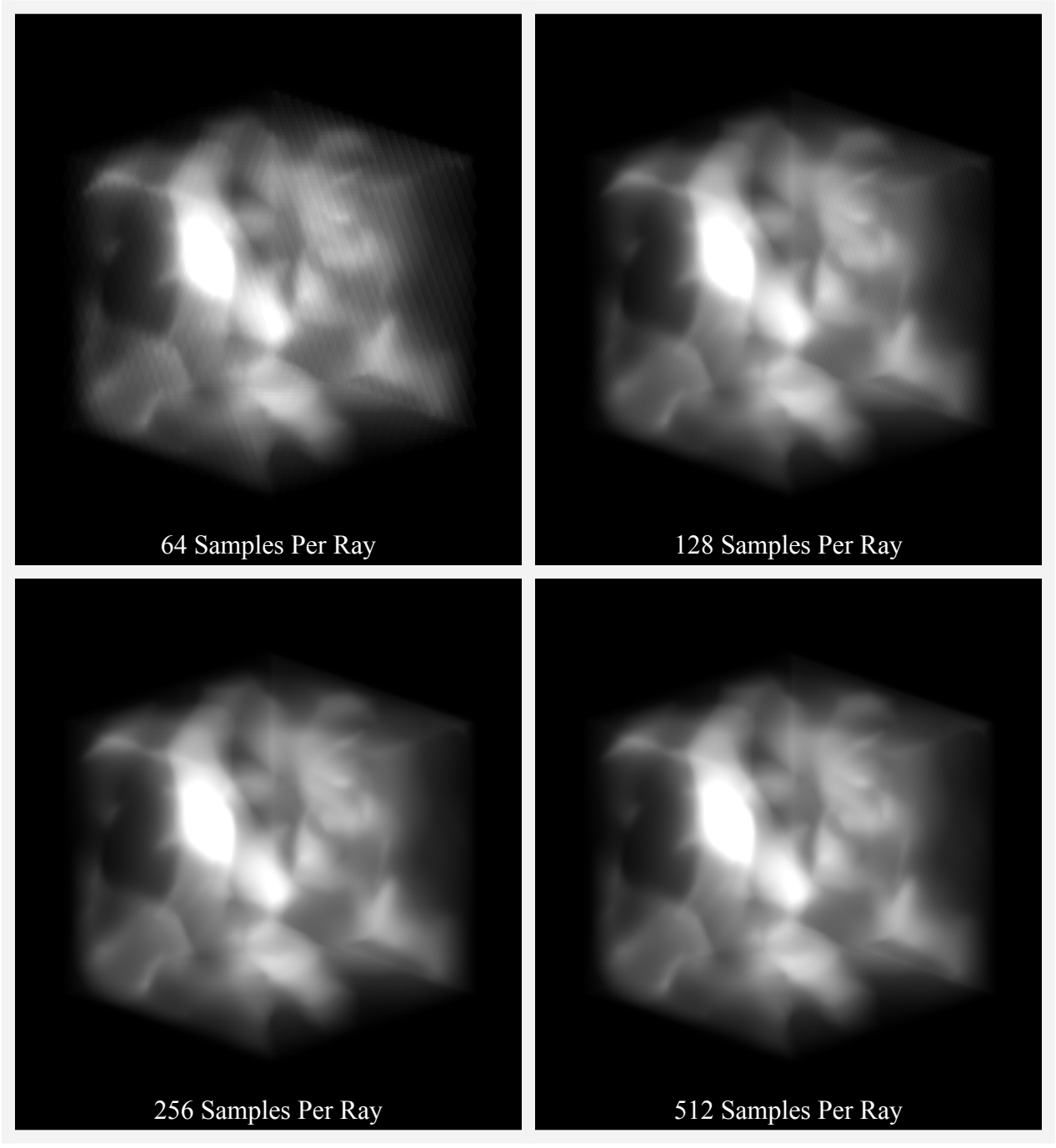


Figure 1: The results

Chapter 5: Performance

Per-fragment ray calculations that were being computed on the CPU were rewritten to be computed on the GPU. Tests were performed on both implementations where both had to render 60 frames, or 1 second worth of video, where each frame is 400x400 pixels. As stated earlier, each pixel or fragment has an associated ray calculation. Therefore, each frame consists of 160,000 ray computations.

How many sample points a ray calculates depends on the sample rate. Sample rate in this context means that for a sample rate of 1, a sample is taken every 1 unit length of distance stepped along the ray. Multiple sample rates were tested including 0.5, 1, 2, and 4. For these rates, each ray calculates 64, 128, 256, and 512 points respectively because each ray has a constant length of 128. If each ray calculates 128 sample points, then 20,480,000 sample points must be computed for a single frame.

What makes up a sample point computation depends. If the sample point is completely outside the volume, it simply returns no color value. Otherwise, it must do Barycentric interpolation and then do a Blinn-Phong lighting calculation to determine its value.

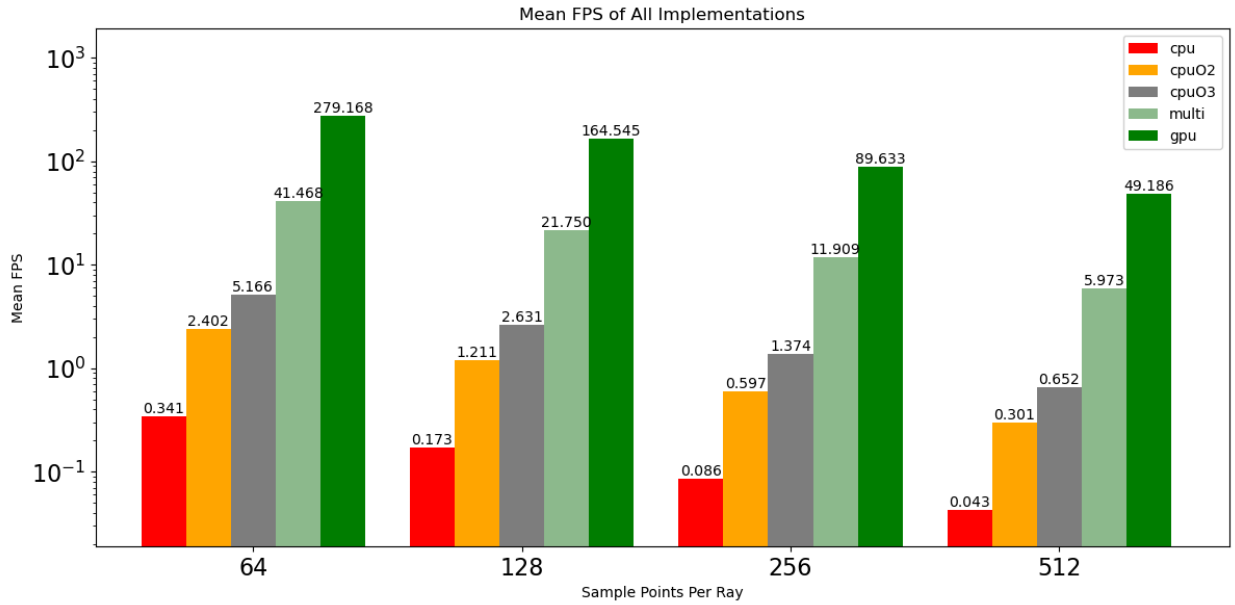
From worst to best performance: CPU without optimization, CPU with 02 optimization, CPU with 03 optimization, CPU with multithreading, and GPU. Graph 1 shows the mean FPS of each implementation on a logarithmic scale. As expected, the GPU outperforms all other implementations by a wide margin. However, the multithreaded version works quite well as a substitute if one doesn't care about real time performance.

Graph 2 shows the speed up factor of each implementation versus the best single threaded implementation, CPU with 03. Each implementation increases performance by roughly a magnitude of one. At the very least, this shows that compiler optimization should be used when

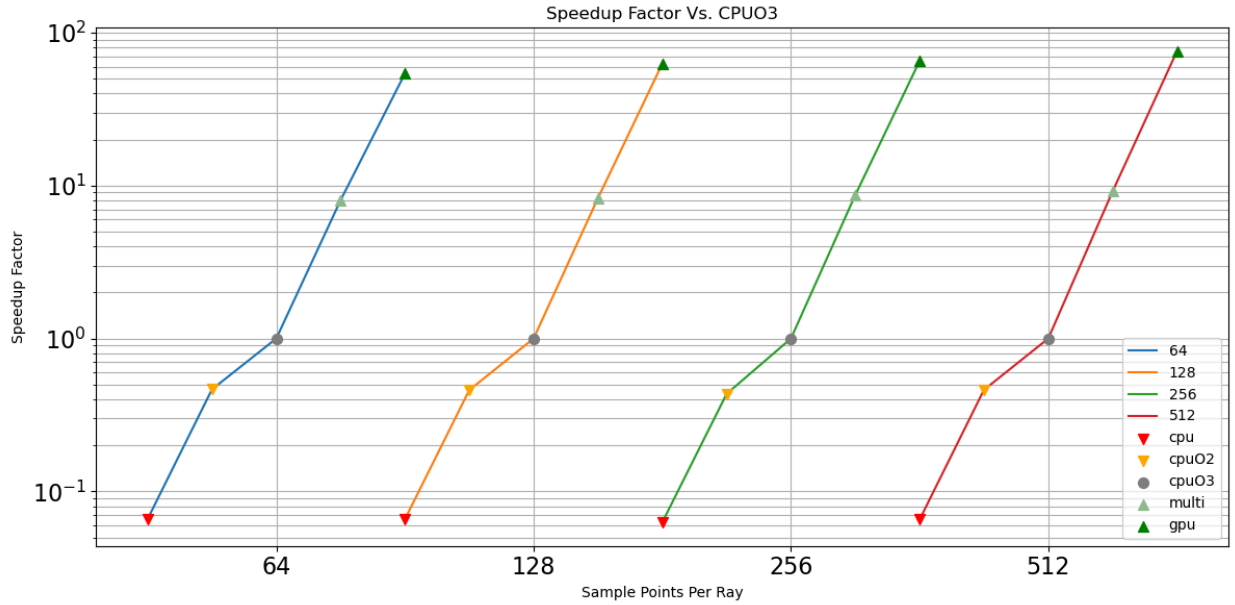
possible if performing heavy computations. Of course, any computations that can be parallelized will benefit greatly from multithreading or GPU processing.

Benchmarks

- cpu: gcc without optimization
- cpu02: gcc with -02 optimization
- cpu03: gcc with -03 optimization
- multi: gcc with multithreading, 64 threads was found to be optimal
- gpu: GLSL implementation



Graph 1: Mean FPS



Graph 2: Speedup Factor

Conclusion

Volume ray casting is a niche area of an already niche field of Computer Science. Still, there are many implementations of volume ray casting. The combination of methods used in this paper to produce rendered volumes is merely one approach out of a myriad of approaches. However, this project's approach is one of the most straightforward ones. The reason for its simplicity is twofold. Firstly, for a glimpse into the world of ray casting, and secondly to deepen my understanding of topics learned in class.

Furthermore, this paper shows how well GPUs perform on applications where the computations are easily parallelizable. It just goes to show that domain specific architectures should be utilized to their fullest potential especially in applications with heavy computations. It only makes sense to write code for the architecture where that code will see the greatest performance increase.

Reference List

[1] S. Guha, *Computer Graphics through Opengl®: From Theory to Experiments: Comprehensive Coverage of Shaders and the Programmable Pipeline*. Boca Raton: Taylor & Francis Group, 2019.

[2] B. Lichtenbelt, R. Crane, and S. Naqvi, *Introduction to volume rendering*. Upper Saddle River, NJ: Prentice-Hall, 1998.

[3] C.D. Hansen and C.R. Johnson, "Ch. 7 Overview of Volume Rendering," in *The Visualization Handbook,* Amsterdam: Elsevier Butterworth Heinemann, 2005. pp-127–174.

[4] R. Bán and G. Valasek, "Tetrahedral Interpolation on Regular Grids." The Eurographics Association, 2021. Available: https://doi.org/10.2312/egp.20211031 [Accessed April 9, 2024].