



NHTV BREDA UNIVERSITY OF APPLIED SCIENCES

PERSONAL PROJECT

Minecraft: Ray Traced

Marco Jonkers

Supervisor: David H rchner

Abstract

This project attempts to implement a GPU ray tracer in the video game Minecraft, using CUDA. The ray tracing itself is done in CUDA. At first, I attempt to ray trace the vertex buffers given by Minecraft. Then, I implement my own data structures for increased performance.

April 4, 2017

Contents

1	Introduction	2
2	Minecraft Forge	2
3	Minecraft	2
3.1	Minecraft Rendering System	2
4	Minecraft Forge	3
4.1	Coremods	3
5	Ray Tracing OpenGL Vertex Buffers	3
5.1	Viewport and Ray Origin	3
6	Challenges	3
7	Static Geometry	3
8	Benchmarks	3
9	Vertex Buffer Preprocessing	3
10	Future work	3
11	References	3
12	List of Figures	3



1 Introduction

Minecraft is a video game developed and published by Mojang. In Minecraft, the game world is mostly comprised of unit blocks. Similar to other ray tracing research projects by Intel Corporation. Also shortly explain the benefits of ray tracing versus regular rasterization. Minecraft's world is made up of voxels, which is nice for ray tracing.

2 Minecraft Forge

Here I explain how the project is set up and the technologies involved. Minecraft is coded in Java. The Minecraft Forge project allows me to do two things: Listen to specific events using pre-installed hooks Edit Minecraft bytecode before it is loaded Using Java Native Interface (JNI), I can call into C++ code from Java. The C++ code contains the CUDA kernel. Data is passed between OpenGL and CUDA using CUDA's graphics interop layer.

I am using the Minecraft Forge API. Forge is built on top of the Mod Coder Pack (MCP). MCP is a tool for decompiling Minecraft. A copy of the game is obtained by using the official installer from Mojang. Forge uses Gradle. During development, Java loads my classpath directly. In release mode, my class files would have to be compressed into an archive first. The path to my DLLs is passed to the virtual machine.

3 Minecraft

3.1 Minecraft Rendering System

There are four geometry groups:

Solid This is solid geometry.

Cutout Used for glass.

Mipped Cutout Identical to Cutout, but mipmapped.

Translucent Used for block which have partial transparency (alpha blending).

Every geometry group has its own vertex buffer.

4 Minecraft Forge

4.1 Coremods

5 Ray Tracing OpenGL Vertex Buffers

I explain something here that is found in Figure 10.

5.1 Viewport and Ray Origin

The viewport is passed from Java to C++ using ByteBuffers.

6 Challenges

The challenges of this project include:

Ray tracing performance Because of gameplay.

Acceleration structure rebuild speed Because of gameplay.

7 Static Geometry

Test for citing [1].

I am also citing [2].

I am also citing [3].

8 Benchmarks

9 Vertex Buffer Preprocessing

10 Future work

Future work is addressed here.

11 References

- [1] John Amanatides, Andrew Woo, et al. A fast voxel traversal algorithm for ray tracing. In *Eurographics*, volume 87, pages 3–10, 1987.
- [2] Paulo Ivson, Leonardo Duarte, and Waldemar Celes. Gpu-accelerated uniform grid construction for ray tracing dynamic scenes. *Master's thesis, Departamento de Informatica, Pontificia Universidade Catolica, Rio de Janeiro*, 2009.
- [3] Erik Reinhard, Brian Smits, and Charles Hansen. Dynamic acceleration structures for interactive ray tracing. In *Rendering Techniques 2000*, pages 299–306. Springer, 2000.

12 List of Figures

1	Sequence diagram	4
2	Top-down view of vertex buffer array	5
3	Diagram of a RenderChunk	6
4	Viewport and ray origin calculations	6
5	Binary interaction	7

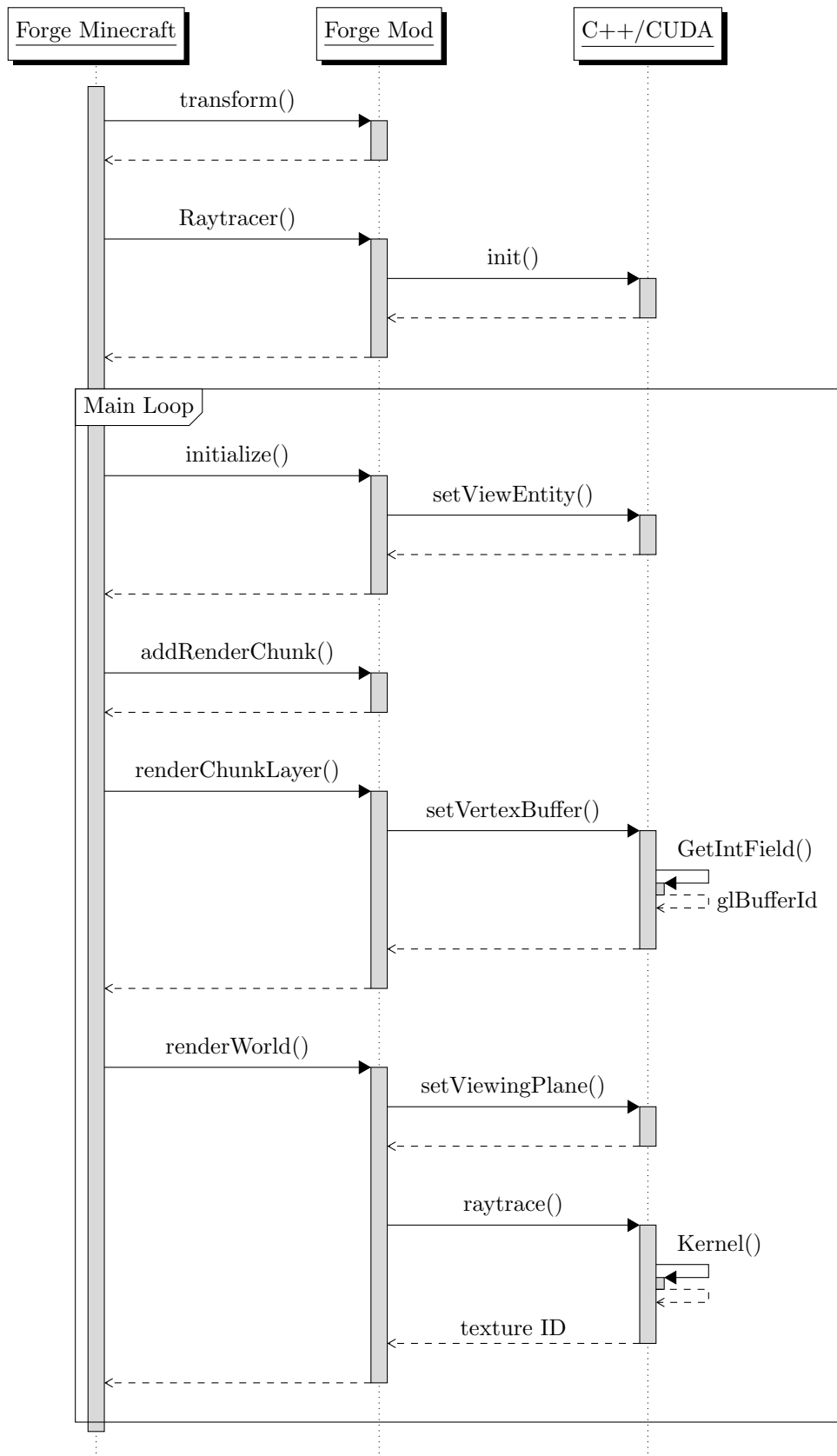


Figure 1: Sequence diagram showing interaction between the modules

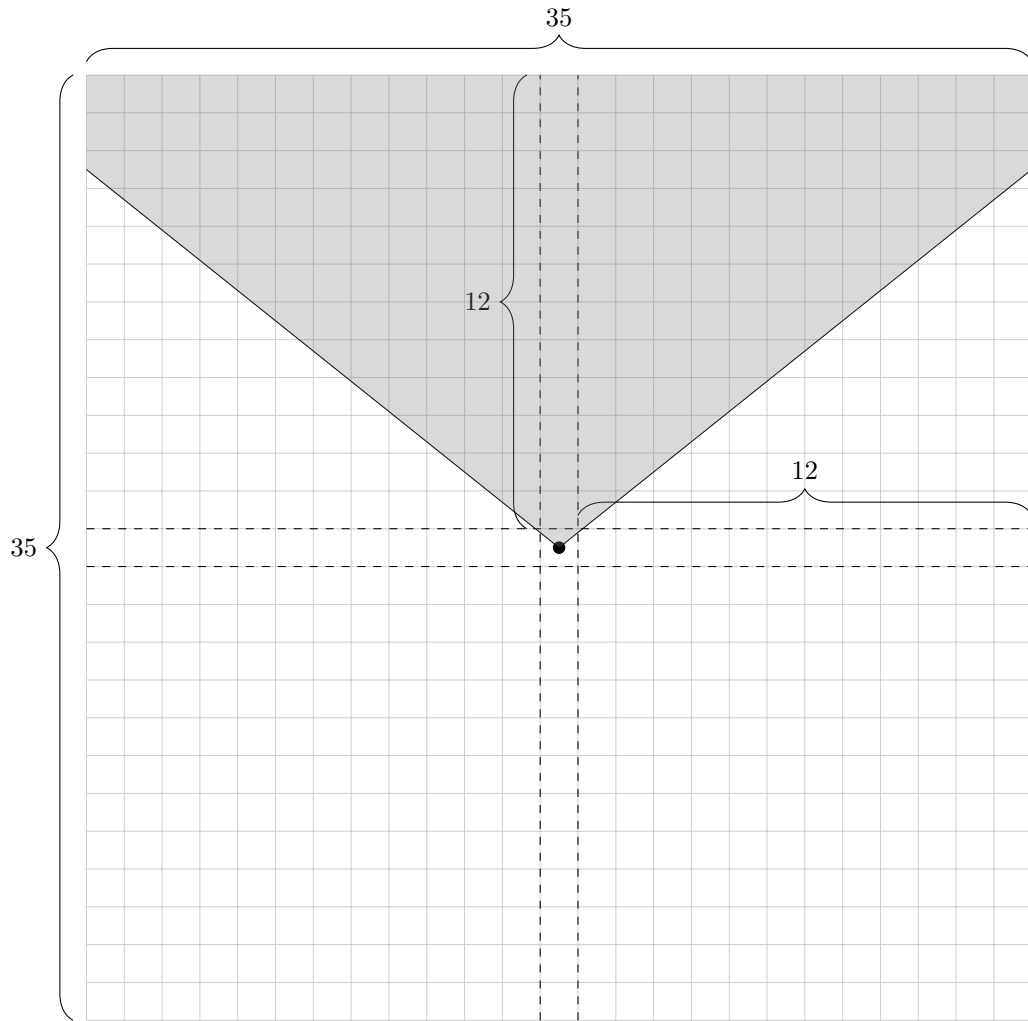


Figure 2: A top-down view of the vertex buffer array, assuming a render distance of 12. The player's view cone is shown facing north. Each cell contains 16 RenderChunks. When the player crosses a horizontal chunk boundary, every buffer in the grid is move once space.

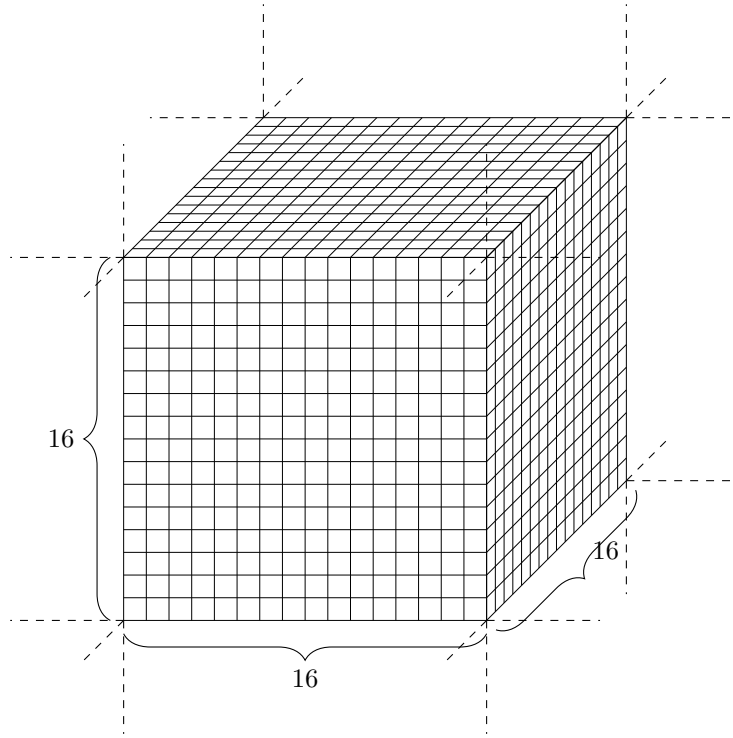
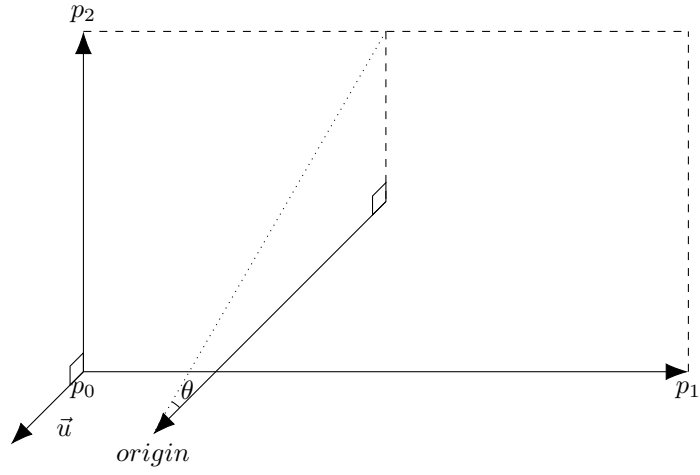


Figure 3: A single, completely filled RenderChunk.



$$\begin{aligned}
 \theta &= \frac{\text{vertical field of view}}{2} \\
 \vec{u} &= \overrightarrow{p_0 p_1} \times \overrightarrow{p_0 p_2} \\
 \text{origin} &= \frac{p_1 + p_2}{2} + \frac{\vec{u}}{\|\vec{u}\|} \cdot \frac{\frac{\|\overrightarrow{p_0 p_2}\|}{2}}{\tan(\theta)}
 \end{aligned} \tag{1}$$

Figure 4: Viewport and ray origin calculations

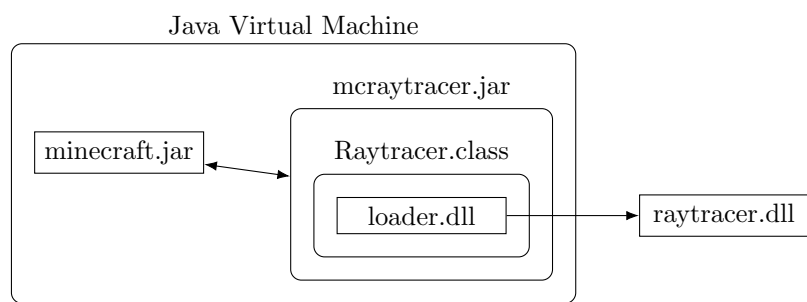


Figure 5: Interaction diagram of the different binaries.