NHTV Breda University of Applied Sciences

Personal project

Minecraft: Ray Traced

Marco Jonkers

Supervisor: David Hörchner

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 5, 2017

Contents

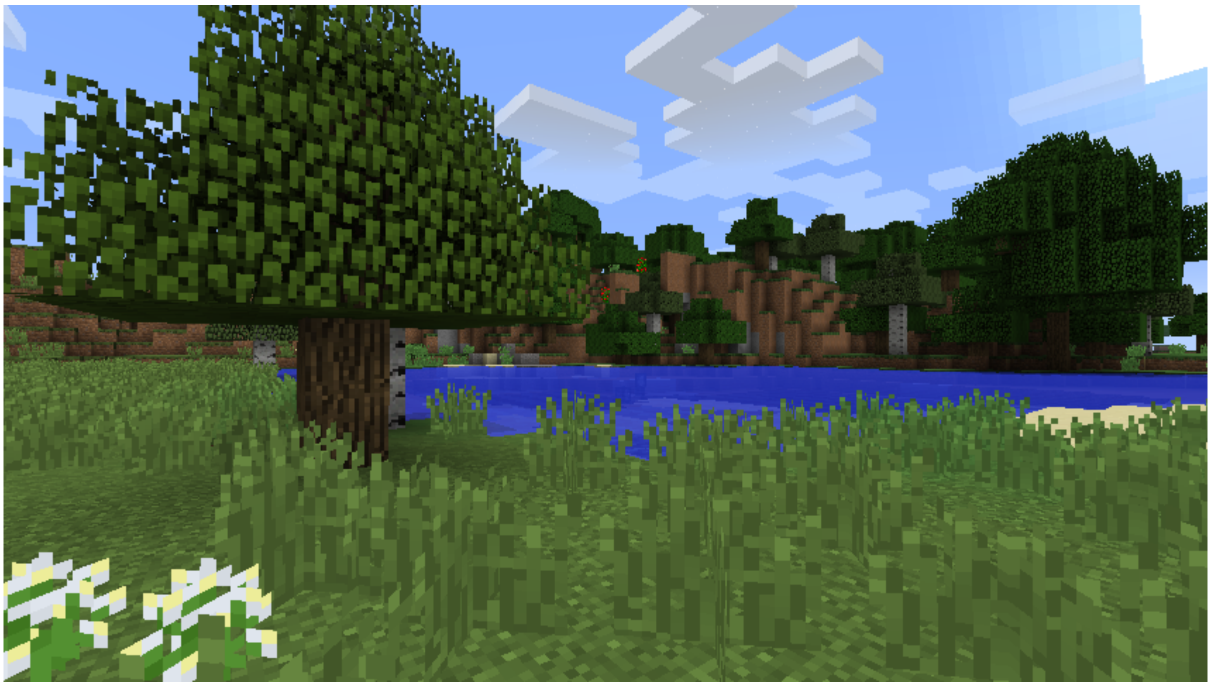


Figure 1: In-game screenshot of Minecraft, without heads-up display (HUD) or viewmodel.

1 Minecraft

Minecraft is a 2011 first-person sandbox video game. Originally created by Markus "Notch" Persson, it is currently maintained by Mojang.

Minecraft comes in multiple editions, for various platforms. This paper focuses on the PC version of Minecraft, released for Windows, macOS, and Linux. The PC version is written in Java.

In Minecraft, the game world consists of a three-dimensional grid. The world is procedurally generated, using a mostly comprised of unit blocks, as shown in Figure 1.

2 Minecraft Renderer

Minecraft uses multi-threaded chunk generation. Minecraft uses OpenGL. Minecraft has two options for rendering static geometry:

Display Lists An OpenGL 1.0 core function. Display Lists are a group of OpenGL commands that have been compiled and sent to the GPU. An object can then be drawn by calling the list. The list can be reused, which means you do not have to send the data over again.

Vertex Buffer Objects Vertex Buffer Objects ("VBOs") were available in OpenGL 1.4 through an extension, and were later added to the core specification in version 2.1. This feature allows you to have a buffer with vertex information, and telling the driver where and how the attributes are stored in the buffer.

My project makes use of Minecraft's VBO rendering, because I can extract the vertex data from the buffers.

2.1 RenderChunks

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.

3 Minecraft Forge

Minecraft Forge ("Forge") is a community created platform for developing and using modifications ("mods") for Minecraft.[1]

3.1 Setting up a mod development environment

The Mod Development Kit ("MDK") can be downloaded from the Forge website[2]. The MDK distribution includes ForgeGradle[3]. ForgeGradle is a plugin for the Gradle[4] build system. The Minecraft binaries are downloaded, and subsequently decompiled, deobfuscated, The classes are decompiled into srg names.

The Mod Coder Pack[5] is a package which is used to decompile, change, and recompile Minecraft Java classes.

The Gradle tasks include:

1. Download Minecraft .jar files.
2. Decompile the Minecraft
3. Generate the Forge Minecraft binary

Forge explicitly supports the Eclipse and IntelliJ integrated development environments ("IDEs"). I used IntelliJ IDEA Community. ForgeGradle gives you the option of debugging and building both client and server sides of Minecraft.

3.2 Changes to Minecraft source code

Forge adds some new features to the Minecraft source code, in order to support multi-mod functionality.

3.3 Creating a mod

In general there are three approaches to creating a mod:

1. Build a mod on top of the Forge Minecraft code
2. Edit the Minecraft source directly
3. Change the Minecraft bytecode at runtime

For most mods, the first approach is sufficient. This also enables the developer to freely distribute their mod. Editing the Minecraft source directly means that the mod cannot be redistributed, because the original Minecraft code is copyrighted.

4 C++ and CUDA

I wanted to create a GPU ray tracer. There are various methods to achieve this. I wanted to take advantage of the vertex buffers in CUDA. Because Minecraft uses OpenGL through the LWJGL[6] library, I could have chosen to use OpenGL compute shaders. However, I have no experience with OpenGL compute shaders, whereas I do have experience with NVIDIA CUDA.

There exist Java bindings for CUDA, but using C++ removes my dependency on a binding layer.

4.1 Java Native Interface

Java Native Interface ("JNI") is a programming framework that allows for Java code to interact with native code.

[1]Modification of Minecraft ("modding") is not officially supported by Mojang. For more information, visit `https://account.mojang.com/terms`

[2]`http://files.minecraftforge.net/`

[3]`https://github.com/MinecraftForge/ForgeGradle`

[4]`https://gradle.org/`

[5]`http://www.modcoderpack.com/website/`

[6]`https://www.lwjgl.org/`

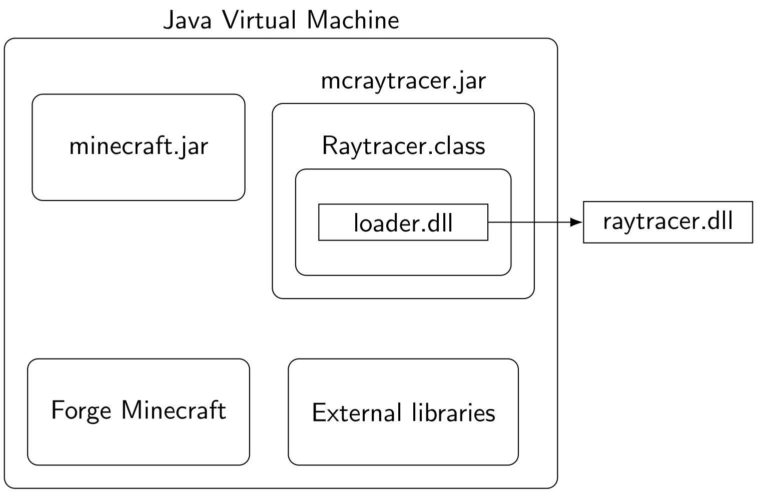


Figure 2: Interaction diagram of the different binaries.

4.2 CUDA/OpenGL interoperability

4.3 Creating a Visual Studio project

For setting up the native part of the mod, I created a Visual Studio CUDA project.

4.3.1 Reloading C++

Java IDEs such as IntelliJ support hot-swapping of method bodies by using the HotSwap^{TM}functionality of the Java Platform Debugger Architecture. I created something similar for my native code by using a technique called DLL reloading. Once a native library has been loaded by the JVM, I cannot make changes to it. Therefore I am using a passthrough DLL, which passes the calls from Java to another DLL. I have shown this in Figure 2.

4.4 Setting up the Visual Studio debugger

IntelliJ debug sessions are good for testing if my Java ASM works correctly, and if my JNI bindings are set up properly. However, if an error occurs a native part, the debug session ends immediately. The crash output from IntelliJ is unable to use the debug information from Visual Studio. I wanted to use the Visual Studio debugger for the native part. Getting that working was quite tricky. I had to launch java.exe with the right command line arguments and working directory. I looked at the launch parameters given to java.exe when starting a debug session from IntelliJ. The Visual Studio debugger can be started by copying the command line arguments from an IntelliJ debug session.

5 Ray Tracing OpenGL Vertex Buffers

5.1 Using events

I wanted to see if I could make my mod work using the first approach. Forge adds hooks to the game loop which I could use to intercept the rendering code.

5.2 Using Java ASM

Here I explain how the project is set up and the technologies involved.

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.

6 Preprocessing Vertex Buffers

6.1 Obtaining the buffers

6.2 Acceleration structure

7 Future work

I only got to work on the world rendering of the game. Viewmodels (player-held items) are not ray traced. Non-playable characters ("NPCs") are not ray traced.

8 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.

9 List of Figures

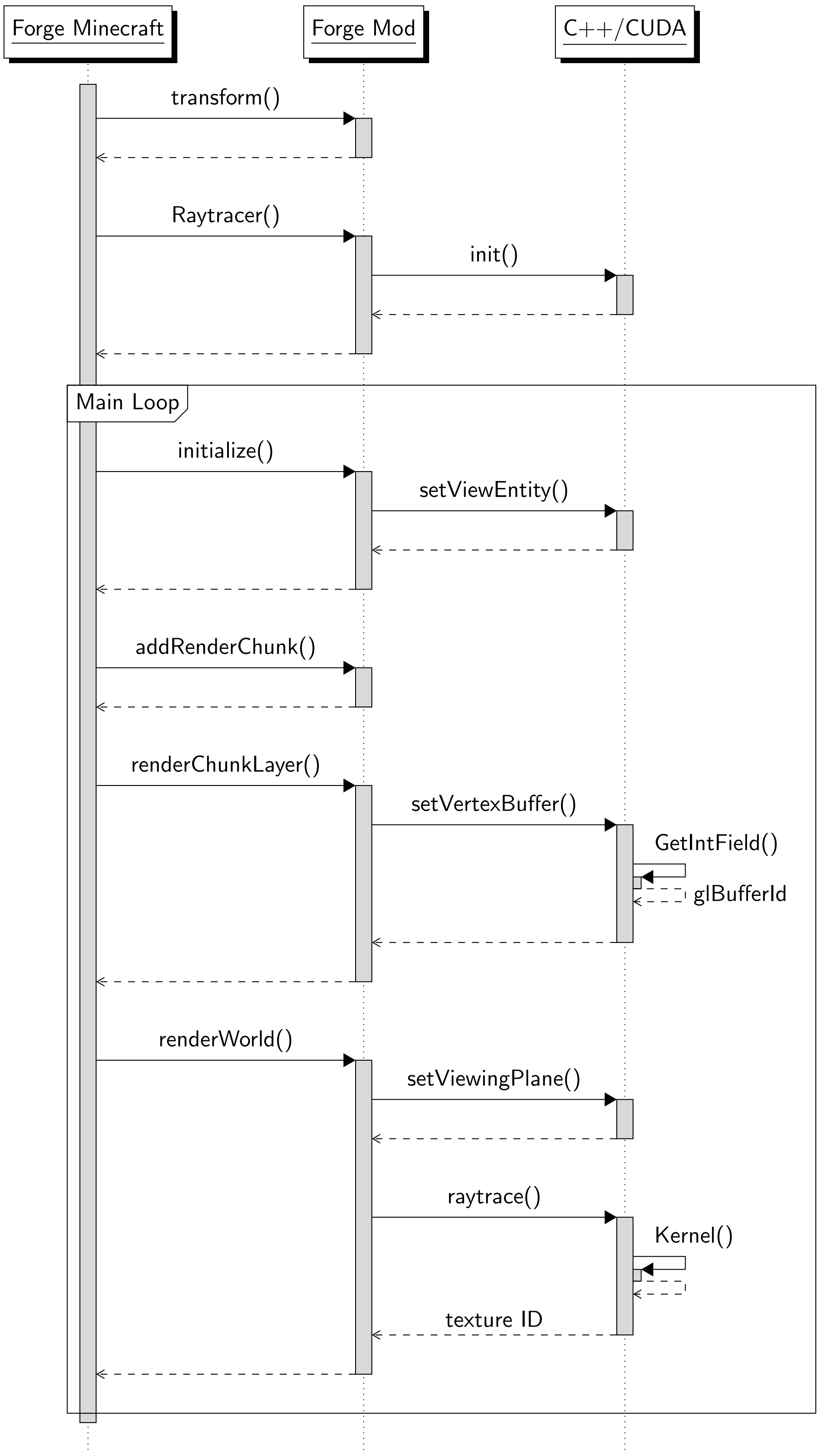


Figure 3: Sequence diagram showing interaction between the modules

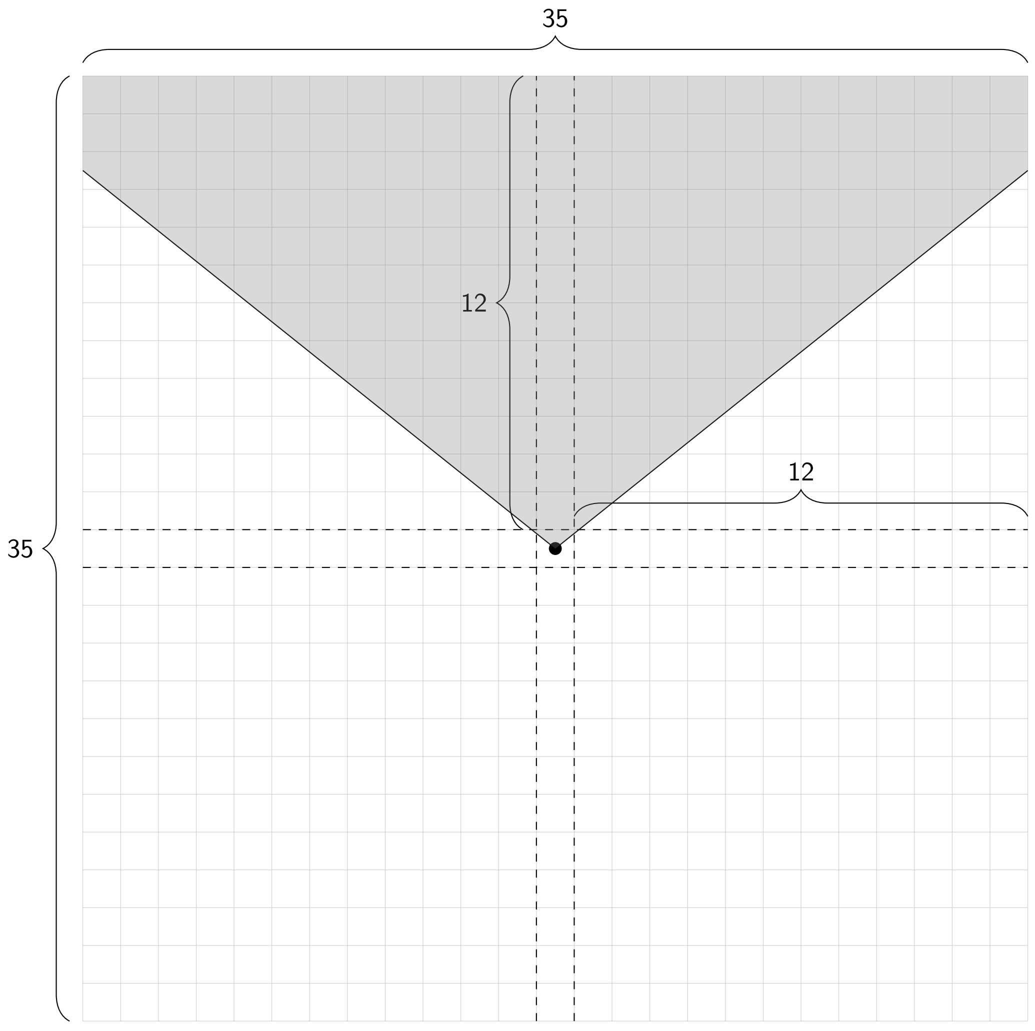


Figure 4: A top-down view of the vertex buffer array, assuming a render distance of 12. The player's view cone is shown facing north. Each cell contains 16 RenderChunks. When the player crosses a horizontal chunk boundary, every buffer in the grid is move once space.

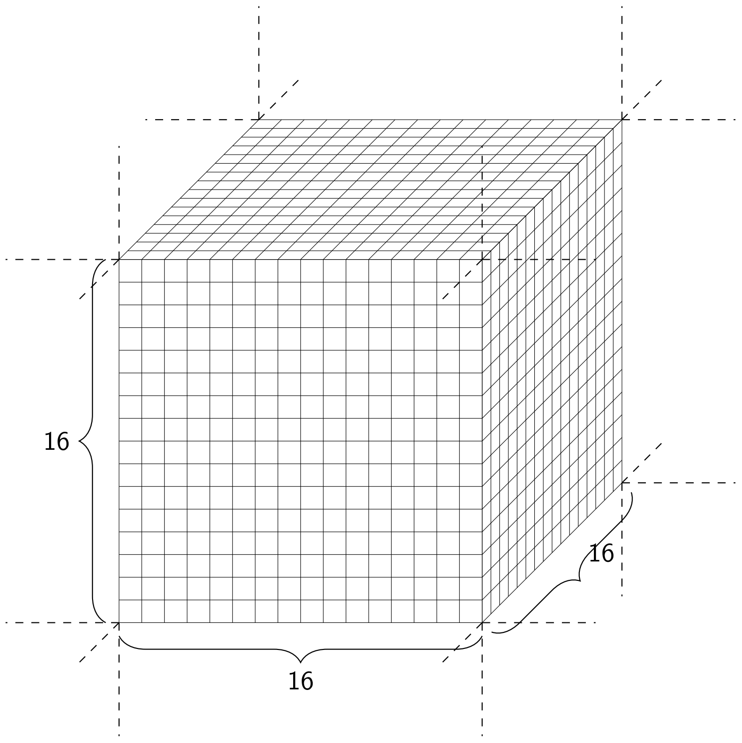
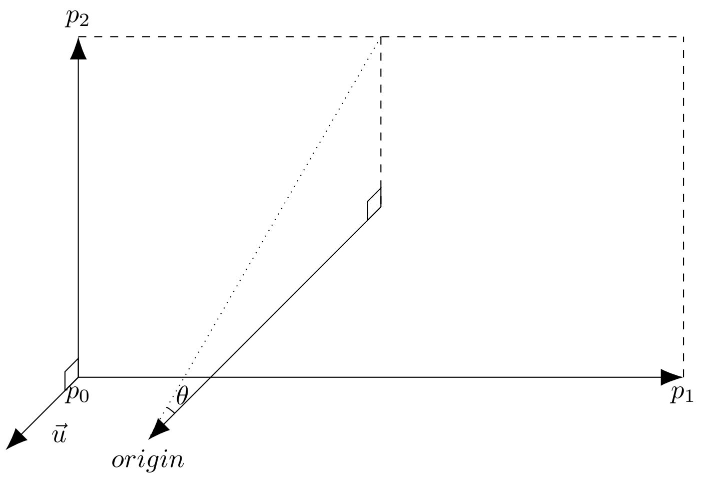


Figure 5: A single, completely filled RenderChunk.



$$\begin{aligned}\theta &= \frac{vertical\ field\ of\ view}{2}\\ \vec{u} &= \overrightarrow{p_0p_1} \times \overrightarrow{p_0p_2}\\ origin &= \frac{p_1 + p_2}{2} + \frac{\vec{u}}{||\vec{u}||} \cdot \frac{\frac{||\overrightarrow{p_0p_2}||}{2}}{\tan(\theta)}\end{aligned} \tag{1}$$

Figure 6: Viewport and ray origin calculations