



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 7, 2017

Contents

| | | |
|-----------|--|----------|
| 1 | Introduction | 2 |
| 2 | Related Work | 2 |
| 3 | Minecraft Renderer | 2 |
| 3.1 | RenderChunks | 2 |
| 3.2 | Other rendering | 3 |
| 4 | Minecraft Forge | 4 |
| 4.1 | Setting up a mod development environment | 4 |
| 4.2 | Differences between development and release builds | 4 |
| 4.3 | Changes to Minecraft source code | 4 |
| 4.4 | Creating a mod | 4 |
| 5 | Hardware | 4 |
| 6 | C++ and CUDA | 5 |
| 6.1 | Java Native Interface | 5 |
| 6.1.1 | javah | 5 |
| 6.2 | CUDA | 5 |
| 6.3 | CUDA/OpenGL interoperability | 5 |
| 6.4 | Creating a Visual Studio project | 6 |
| 6.4.1 | Reloading C++ | 6 |
| 6.4.2 | Setting up the Visual Studio debugger | 6 |
| 6.5 | Kernel overhead on Windows | 7 |
| 7 | Ray Tracing OpenGL Vertex Buffers | 7 |
| 7.1 | Forge Events | 7 |
| 7.2 | Using Java ASM | 8 |
| 7.2.1 | IClassTransformer | 8 |
| 7.3 | C++ | 8 |
| 7.4 | CUDA | 8 |
| 7.5 | Results | 8 |
| 8 | Preprocessing Vertex Buffers | 8 |
| 8.1 | Obtaining the buffers | 8 |
| 8.2 | C++ | 8 |
| 8.3 | CUDA | 8 |
| 8.4 | Results | 8 |
| 9 | Conclusion | 8 |
| 10 | Future work | 8 |
| 11 | References | 8 |



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

1 Introduction

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 Related Work

3 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.

3.1 RenderChunks

See figure 2.

There are four geometry groups, based on texture properties.

Solid Uses textures that are fully opaque. Most world blocks are in this group.

Cutout Uses textures that have transparent texels.

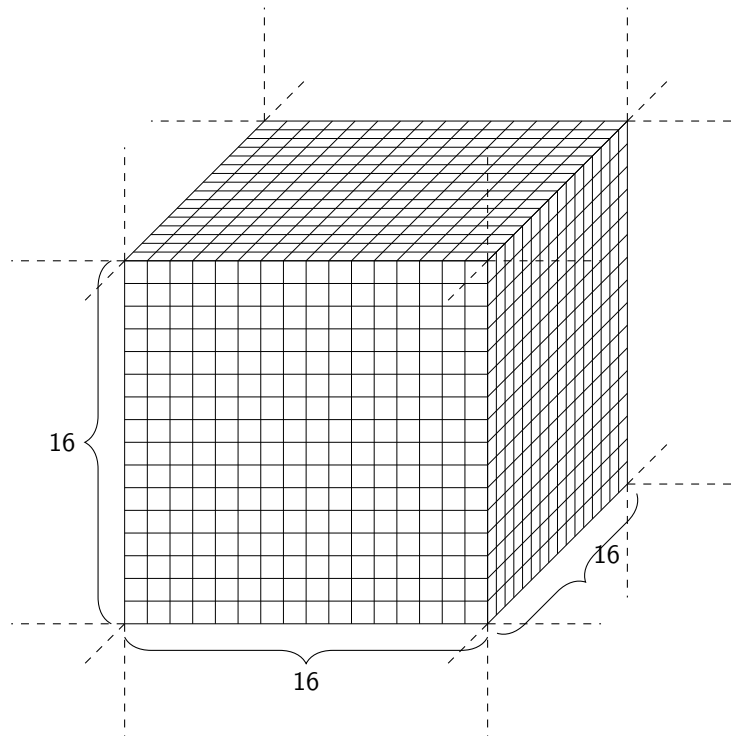


Figure 2: A single, completely filled RenderChunk.

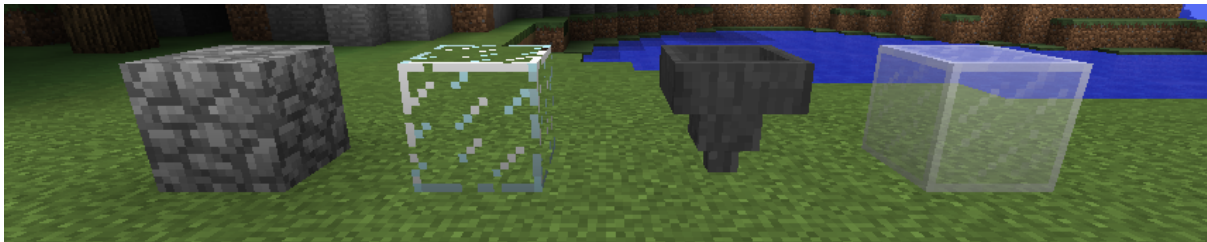


Figure 3: One object from each render layer. From left to right: Cobblestone (Solid), Glass (Cutout), Hopper (Cutout Mipped), and Stained Glass (Translucent).

Mipped Cutout Identical to Cutout, but mipmapped.

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

Vertex format:

| Attribute | Type | # | Bytes |
|-----------|-------|---|-------|
| Position | float | 3 | 12 |
| Color | byte | 4 | 4 |
| Lightmap | float | 2 | 8 |
| Texture | short | 2 | 4 |
| Total | | | 28 |

3.2 Other rendering

TileEntitySpecialRenderer etc.

Every geometry group has its own vertex buffer.

4 Minecraft Forge

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

4.1 Setting up a mod development environment

The Mod Development Kit ("MDK") can be downloaded from the Forge website². The MDK distribution includes ForgeGradle³. ForgeGradle is a plugin for the Gradle⁴ build system. The Minecraft binaries are downloaded, and subsequently decompiled, deobfuscated, The classes are decompiled into srg names.

The Mod Coder Pack⁵ 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.

The path to my DLLs is passed to the virtual machine.

4.2 Differences between development and release builds

During development, Java loads my classpath directly. In release mode, my class files would have to be compressed into an archive first.

4.3 Changes to Minecraft source code

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

4.4 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. Use class transformers in custom mod loading

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.

5 Hardware

This project is developed on the following machines:

¹Modification 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/>

| | | |
|--------------------------|---------------------------------|------------------------|
| | Dell XPS 15 L502X | Desktop |
| Operating System | Microsoft Windows 10 Pro 64-bit | |
| Processor | Intel Core i7-2630QM | Intel Core i7-4790 |
| Memory | 4 GB DDR3 | 8 GB DDR3 |
| GPU | NVIDIA GeForce GT 540M | NVIDIA GeForce GTX 970 |
| - CUDA Cores | 96 | 1664 |
| - Shader Multiprocessors | 2 | 13 |
| - Compute Capability | 2.1 | 5.2 |
| - Memory Clock | 900 MHz | 7010 MHz |
| - Graphics Clock | 672 MHz | 1140 MHz |
| - Memory | 2 GB | 4 GB ⁶ |

6 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 LWJGL⁷ 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.

6.1 Java Native Interface

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

6.1.1 javah

`javah` is a tool that generates C header and source files for Java classes that declare native methods. I use the generated header file in my Visual Studio project. The header file is compatible with C++ code bases as it includes an `#ifdef __cplusplus` directive which prepends an `extern "C"` to the function declarations. Function parameters are passed as the following⁸:

| Java | C++ | represented type |
|---------|----------|------------------|
| boolean | jboolean | unsigned 8 bits |
| byte | jbyte | signed 8 bits |
| char | jchar | unsigned 16 bits |
| short | jshort | signed 16 bits |
| int | jint | signed 32 bits |
| long | jlong | signed 64 bits |
| float | jfloat | 32 bits |
| double | jdouble | 64 bits |
| void | void | not applicable |

Additionally, Java objects are passed as (subclasses of) `jobject` instances, as can be seen in Figure 4.

6.2 CUDA

CUDA is a parallel computing platform and programming model developed by NVIDIA. It is only compatible with NVIDIA hardware. CUDA has two APIs: Runtime and Driver.

6.3 CUDA/OpenGL interoperability

CUDA provides an API for operating on OpenGL primitives.

1. Run the ray tracing kernel
2. Map the OpenGL texture to a CUDA array
3. Copy the kernel output buffer to the CUDA array

⁷<https://www.lwjgl.org/>

⁸<https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/types.html>

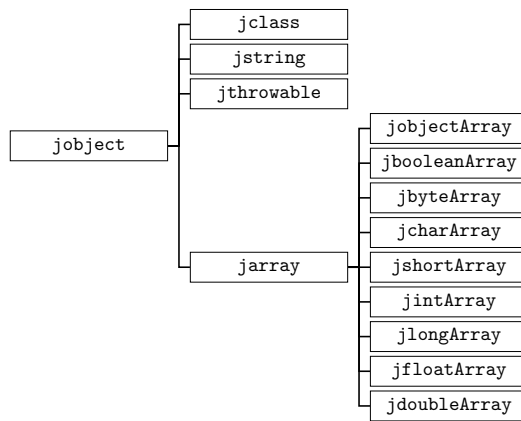


Figure 4: C++ inheritance diagram

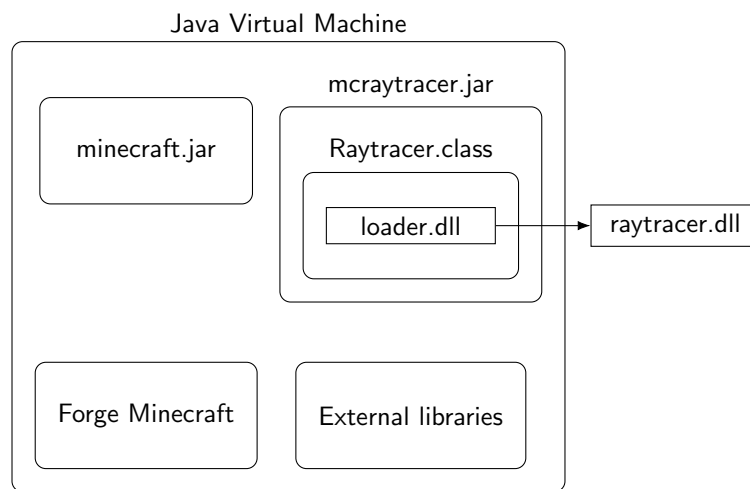


Figure 5: Interaction diagram of the different binaries.

6.4 Creating a Visual Studio project

For setting up the native part of the mod, I created a Visual Studio CUDA project. NVIDIA has a CUDA plugin for Visual Studio called Nsight.

6.4.1 Reloading C++

Java IDEs such as IntelliJ support hot-swapping of method bodies by using the HotSwap 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 5.

6.4.2 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. 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 configured to launch Minecraft by copying the command line arguments from an IntelliJ debug session. The JVM uses the access violation signal internally for its memory exception handling. When an access violation is thrown from the JVM, this can be safely ignored. However, Visual

Studio can easily block hundreds of these signals. Visual Studio 2017 offers an option to ignore certain exceptions per module, allowing me to break on access violations in my own native code while ignoring those coming from the JVM.

6.5 Kernel overhead on Windows

There is a non-trivial amount of overhead attached to the launch of a CUDA kernel on Windows. This is due to the Windows Display Driver Model ("WDDM"). An alternative would be to use the Tesla Compute Cluster driver ("TCC"). Unfortunately, TCC is not available for GeForce GPUs. Additionally, GPUs in TCC cannot be connected to a display.

7 Ray Tracing OpenGL Vertex Buffers

My initial idea was to ray trace the uploaded VBOs from Minecraft directly.

7.1 Forge 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. The Raytracer class listens to the following events:

- `FMLInitializationEvent`
As the name indicates, this event is fired when Forge is initializing Minecraft. This event is used to initialize the mod. Here I also obtain a reference to the `Minecraft` singleton class. The mod needs to be registered in the `ClientRegistry` in order to receive further Forge events.
- `TickEvent.ClientTickEvent`
This event is fired 20 times per second, and can be regarded as the client-sided update function. I listen to this event to check for keyboard input changes.
- `GuiScreenEvent.InitGuiEvent.Pre`
This is the first event that is fired when a window resize is detected. The event does not contain any resize information, so I track it manually. The resolution information in the `Minecraft` class is public, so I extract it from there. The information is passed to C++, such that the CUDA and OpenGL resources can be resized if they need to be. OpenGL supports resizing by calling `glTexImage2D` with the new information, without having to call `glGenTextures` again. In CUDA, it is required to unregister and re-register the associated graphics resource. The kernel output buffer is also resized. This uses a pair of `cudaFree` and `cudaMalloc` calls.
- `TickEvent.RenderTickEvent`
This event is fired when Minecraft starts rendering the next frame.

In order to prevent Minecraft from drawing the world after I have done the ray tracing, I copy the `WorldClient` reference from `Minecraft`, and set the value in `Minecraft` to `null`. This effectively causes Minecraft to skip the rendering of the game world, because it has an internal check for the `WorldClient` there. If it is `null`, nothing is rendered. However, this also disables rendering of the game overlay, which contains elements like the player inventory and menu's. I manually draw the game overlay after drawing the ray tracing result to the screen.
- `GuiScreenEvent.DrawScreenEvent.Pre` & `GuiScreenEvent.DrawScreenEvent.Post`
I noticed that the background of the game overlay was solid instead of transparent. This is due to another check for the `WorldClient` instance during drawing. Because I previously made it `null`, I now have to restore it on the `GuiScreenEvent.DrawScreenEvent.Pre` event in order to draw the background. I set it back to `null` at the `GuiScreenEvent.DrawScreenEvent.Post` event.

I was able to draw a full-screen texture using CUDA and copy it to Minecraft using OpenGL. However, I am not able to obtain the Minecraft vertex buffers using Forge Events alone. That is where I either needed to change the Minecraft source directly, or use Java ASM. I chose the second option.

7.2 Using Java ASM

7.2.1 IClassTransformer

7.3 C++

`cudaGraphicsGLRegisterBuffer`

7.4 CUDA

First, we calculate the player's position relative to the chunk it is currently in:

$$f(x) = \frac{x}{16} - \left\lfloor \frac{x}{16} \right\rfloor \quad (1)$$

This maps the player's position to $[0, 1)$.

See equation 2.

$$f(x, dx) = \begin{cases} \frac{\lceil x \rceil - x}{|dx|} & dx > 0 \\ \frac{x - \lfloor x \rfloor}{|dx|} & dx < 0 \\ \infty & dx = 0 \end{cases} \quad (2)$$

7.5 Results

8 Preprocessing Vertex Buffers

8.1 Obtaining the buffers

In order to get the raw vertex buffers from Java to C++, I had to change three Minecraft classes. My approach was to replace the built-in `VertexBuffer` class with my own class, called `CppVertexBuffer`. Confusingly, the decompiled version of Minecraft has two classes called `VertexBuffer` in two separate packages.

RenderChunk The class that represents a 16x16x16 chunk of blocks. It contains one OpenGL vertex buffer per render layer.

ChunkRenderDispatcher

VertexBufferUploader

8.2 C++

8.3 CUDA

8.4 Results

9 Conclusion

10 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.

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.

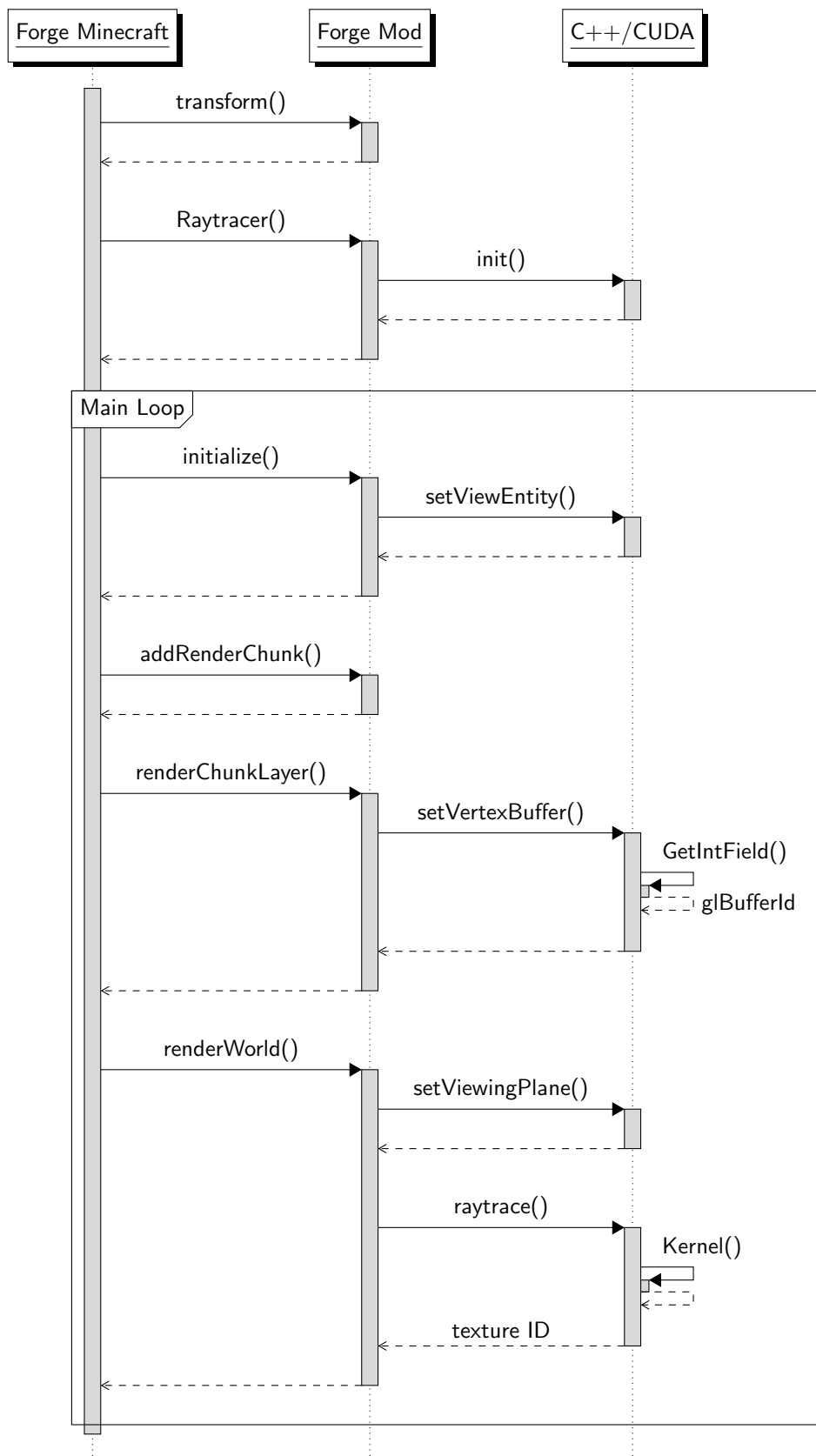


Figure 6: Sequence diagram showing interaction between the modules

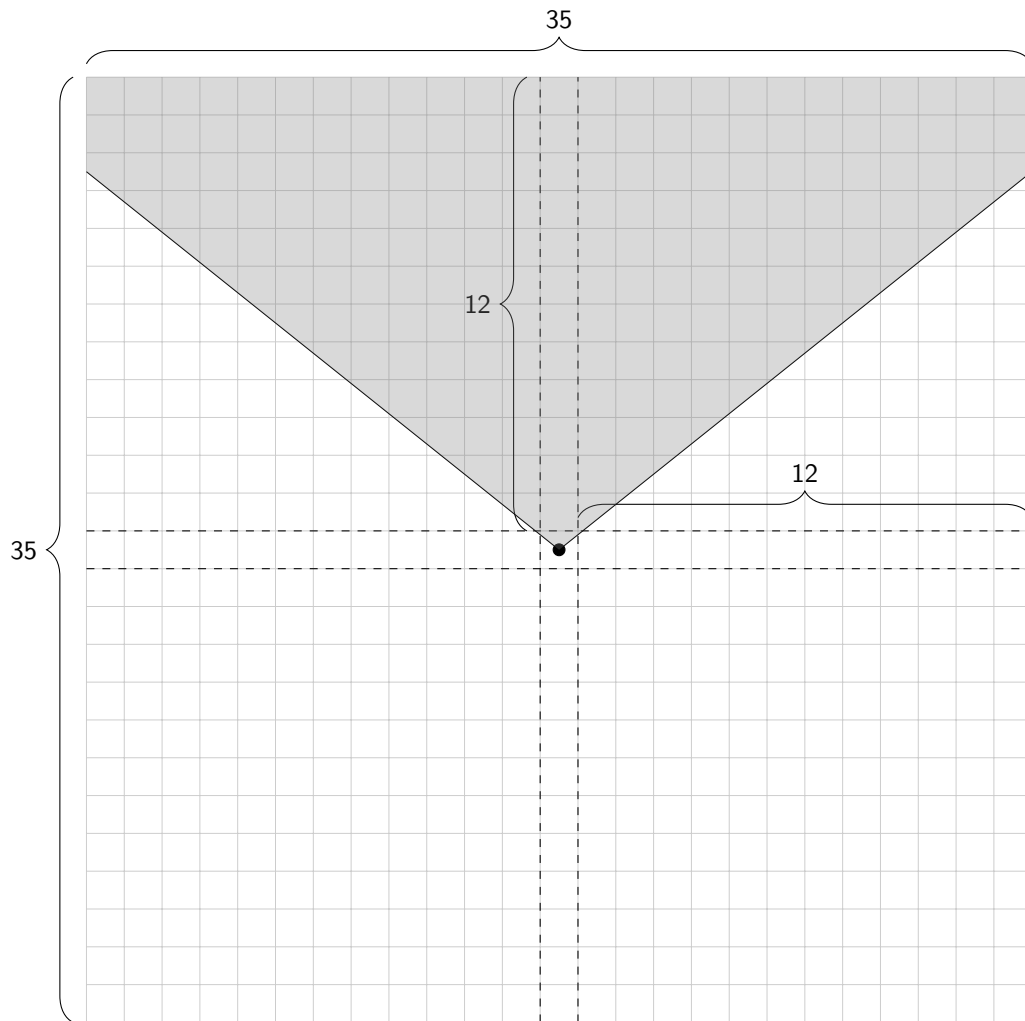
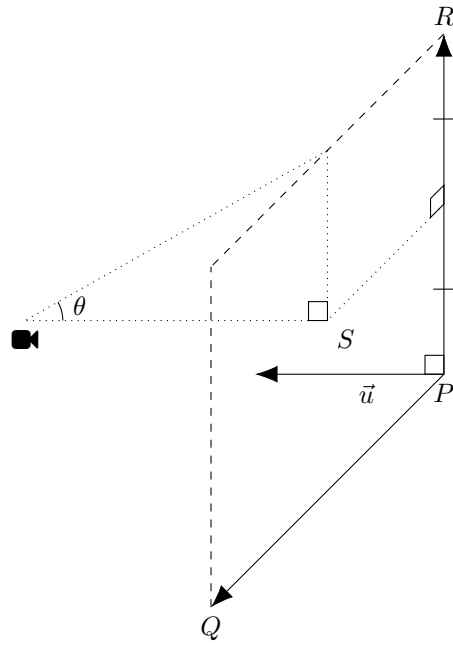


Figure 7: A top-down view of the vertex buffer array, assuming a render distance of 12. The player's view cone is shown facing north. Each cell contains 16 RenderChunks. When the player crosses a horizontal chunk boundary, every buffer in the grid is moved once space.



$$\begin{aligned}
 \theta &= \frac{\text{vertical field of view}}{2} \\
 \vec{u} &= \overrightarrow{PQ} \times \overrightarrow{PR} \\
 S &= \frac{Q + R}{2} \\
 \blacksquare_{pos} &= S + \frac{\vec{u}}{\|\vec{u}\|} \cdot \frac{\|\overrightarrow{PR}\|}{2 \tan(\theta)}
 \end{aligned} \tag{3}$$

Figure 8: Ray origin calculations