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 Minecraft. The ray tracing itself is done in CUDA. First, we attempt to ray trace the Minecraft OpenGL vertex buffers directly. Then, we implement our own data structures for increased performance.

May 1, 2017

Contents

1 List of Figures

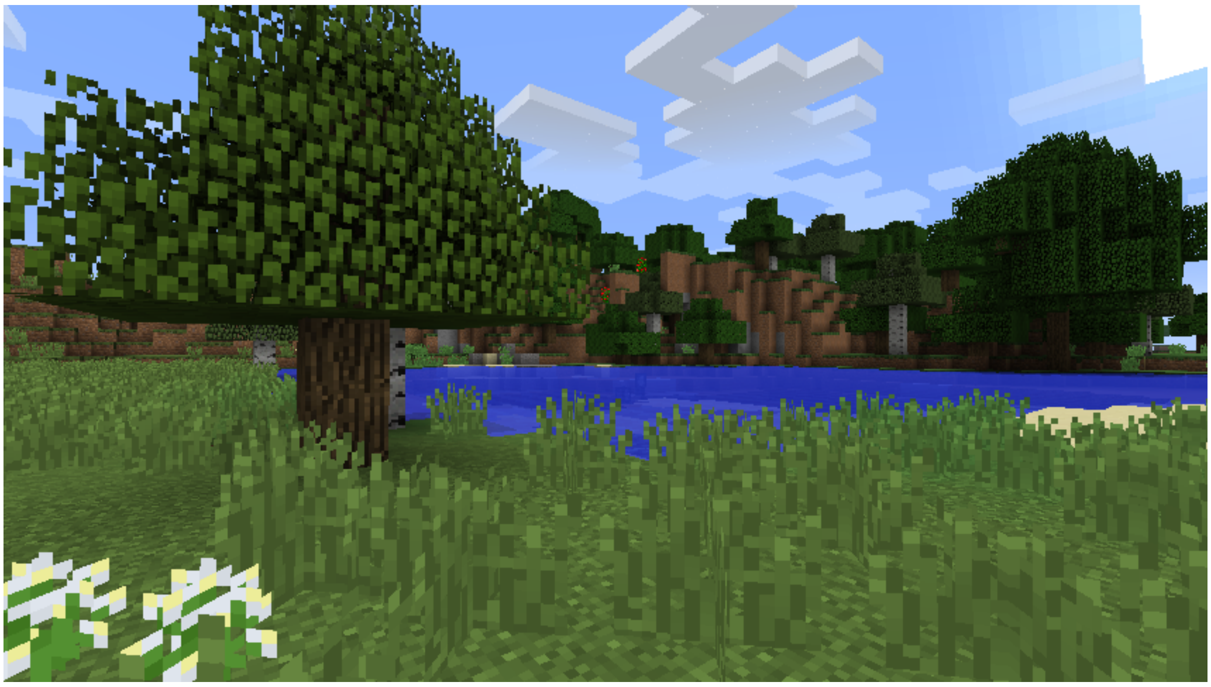


Figure 1: In-game screenshot of Minecraft, without heads-up display (HUD) or viewmodel.

2 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 PC version 1.10.2, 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, and mostly comprised of unit sized blocks, as shown in Figure 1.

3 Minecraft Renderer

Due to memory constraints, Minecraft divides the world up in chunks. These are 16×256×16 columns of blocks. As the player moves, the chunks that are closer to the player are loaded, while those further away are unloaded. The first time a chunk is loaded in a save file, it needs to be generated. Chunks are generated in a multi-threaded system.

For rendering, Minecraft uses OpenGL. There exist two graphics options in the video settings:

Display Lists An OpenGL 1.0 core function. Display Lists are grouped commands which have been compiled and sent to the GPU. An object can be drawn by calling a list. This list can be reused, which means we 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 us to have a buffer with vertex information, and telling the driver where and how the attributes are stored in the buffer.

Our project makes use of Minecraft's VBO rendering, because this lets us extract the vertex data from the buffers.

3.1 RenderChunks

A RenderChunk is a group of 16×16×16 blocks (see Figure 2). Every chunk in the world is comprised of 16 RenderChunks. When a block is added or removed, the corresponding RenderChunks are re-meshed and re-uploaded to the GPU. The size of the RenderChunk results in small GPU buffer updates. As a downside,

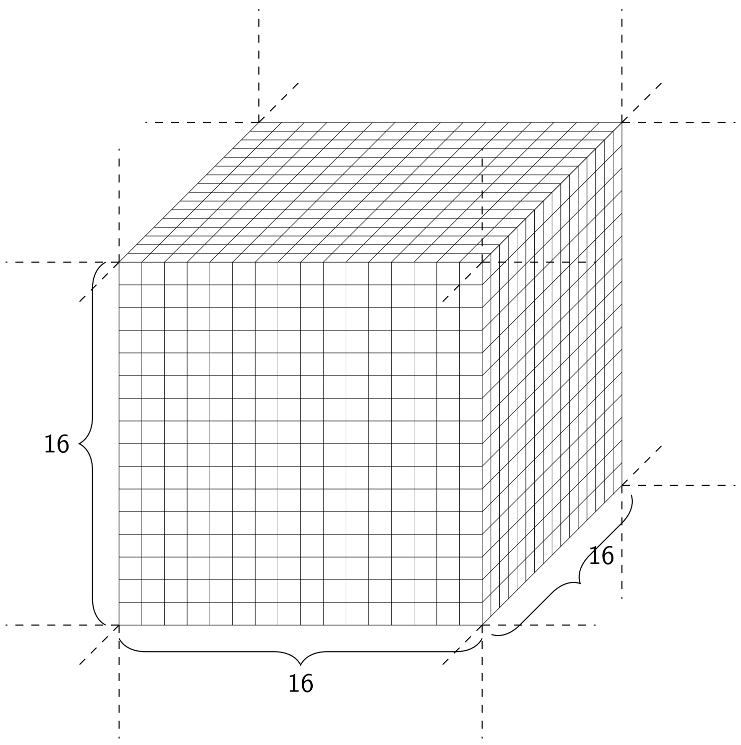


Figure 2: A single, completely filled RenderChunk.

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

Table 1: Vertex attributes.

this means that Minecraft must manage a significant amount of small vertex buffers. The renderer uses frustum culling to remove most RenderChunks from the draw queue.

There are four render layers, based on texture properties (Figure 3).

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

Cutout Uses textures that have transparent texels.

Mipped Cutout Identical to Cutout, but mipmapped.

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

The render layers exist such that Minecraft can do transparency sorting and batching. Note: Our ray tracer has no use for this functionality. A RenderChunk has one vertex buffer for every layer. This results in further fragmentation of the vertex data. However, not all vertex buffers are always in use.

The vertex data from the world geometry contains four attributes (see Table 1).

4 Minecraft Forge

Minecraft Forge ("Forge") is a community created platform for developing and using modifications ("mods") for Minecraft.[1] It was released around November 2011.

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

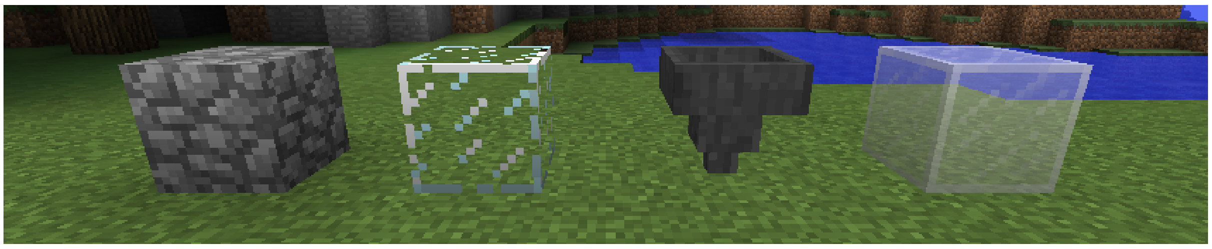


Figure 3: One object from each render layer. From left to right: Cobblestone (Solid), Glass (Cutout), Hopper (Cutout Mipped), and Stained Glass (Translucent).

4.1 Setting up a mod development environment

The Mod Development Kit can be freely downloaded from the Forge website[2]. It includes scripts which set up a development environment for the creation of mods. Forge explicitly supports IntelliJ IDEA as development environment, which is what we will use. The Forge scripts download, decompile, and deobfuscate the Minecraft binaries. The deobfuscation is done with the help of the Mod Coder Pack[3], a community effort which provides useful names to the source code.

4.2 Changes to Minecraft source code

Aside from gaining access to Minecraft's source code, Forge inserts some code to make mod creation easier, and to provide mod compatibility. One of the major features is the event system. A class method can subscribe to an event, which causes the method to be called when the event is fired. There are events for a multitude of gameplay features. Another feature is the Ore Dictionary, which provides a compatibility layer for blocks between mods which share the same functionality.

4.3 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. Use class transformers during 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. The Forge development environment prevents the user from editing Minecraft source files directly. The third approach should be used as a last resort. Class transformers allow a mod to change the bytecode of a vanilla Minecraft class at runtime. Mods which have class transformers are called "coremods". This functionality holds a lot of power, but can cause compatibility issues with other coremods.

5 Hardware

This project is developed on the following machines:

[2]`http://files.minecraftforge.net/`
[3]`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 C++ and CUDA

We want to create a GPU ray tracer. There are various methods to achieve this. We want to take advantage of the vertex buffers in CUDA. Because Minecraft uses OpenGL through the LWJGL[4] library, we could have chosen to use OpenGL compute shaders. However, we have no experience with OpenGL compute shaders, whereas we do have experience with NVIDIA CUDA.

There exist Java bindings for CUDA, but using C++ removes our 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. We use the generated header file in our 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. [Oracle, 2016] Function parameters are passed as the following:

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. We will be using the CUDA Runtime API (as opposed to the CUDA Driver API), as it is the easier of the two to use.

6.3 CUDA/OpenGL interoperability

CUDA provides an API for operating on OpenGL primitives. More specifically, it is able to map an OpenGL vertex buffer to a CUDA device pointer, allowing a CUDA kernel to access the contents of the buffer. We can also copy the contents of a CUDA buffer to an OpenGL texture.

Our strategy for ray tracing will be as follows:

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

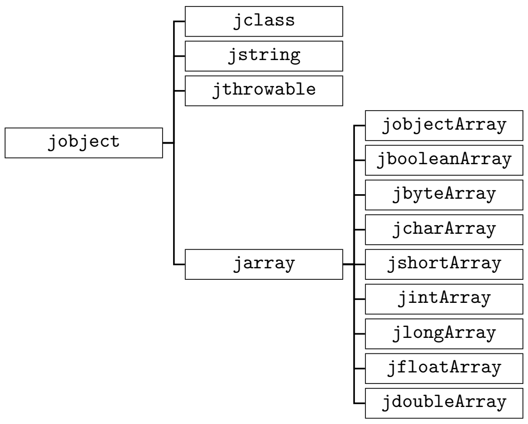


Figure 4: C++ inheritance diagram

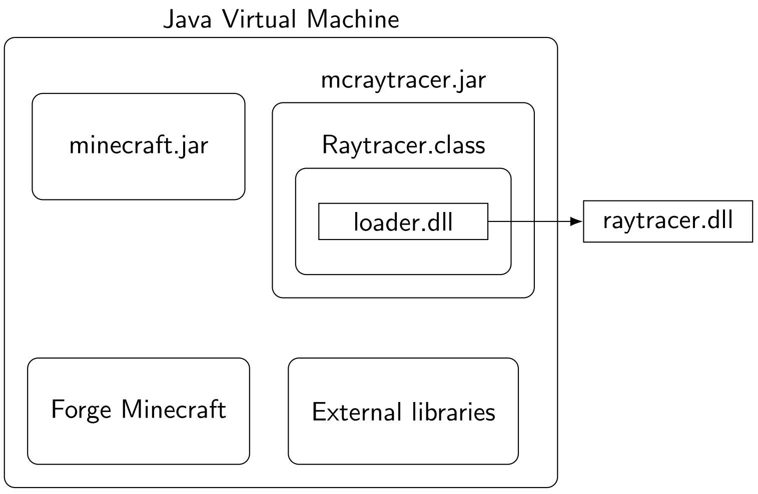


Figure 5: Interaction diagram of the different binaries.

1. Upload world data to CUDA
2. Run the ray tracing kernel
3. Map an OpenGL texture to a CUDA array
4. Copy the kernel output buffer to the CUDA array
5. Pass the OpenGL texture to Minecraft

6.4 Creating a Visual Studio project

For setting up the native part of the mod, we create a Visual Studio CUDA project. NVIDIA has a CUDA plugin for Visual Studio called Nsight. This sets up Visual Studio so we can write CUDA kernels.

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. We created something similar for our native code by using a technique called DLL reloading. Once a native library has been loaded by the JVM, we cannot make changes to it. Therefore we are using a passthrough DLL, which passes the calls from Java to another DLL. We have shown this in Figure 5.

6.4.2 Setting up the Visual Studio debugger

IntelliJ debug sessions are good for testing if our Java ASM works correctly, and if our 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. We use the Visual Studio debugger for the native part. We launch java.exe with the right command line arguments and working directory. We 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 our 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. The measured overhead per kernel launch on both machines is roughly half a millisecond, which is not negligible. Therefore, we avoid executing multiple kernels per frame.

7 Ray Tracing OpenGL Vertex Buffers

Our initial idea is to ray trace the uploaded VBOs from Minecraft directly.

7.1 Forge Events

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 we 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. We 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 we track it manually. The resolution information in the `Minecraft` class is public, so we 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 we have done the ray tracing, we 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. We manually draw the game overlay after drawing the ray tracing result to the screen.

- `GuiScreenEvent.DrawScreenEvent.Pre` & `GuiScreenEvent.DrawScreenEvent.Post`
  We 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 we previously made it `null`,

we now have to restore it on the `GuiScreenEvent.DrawScreenEvent.Pre` event in order to draw the background. We set it back to null at the `GuiScreenEvent.DrawScreenEvent.Post` event.

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

7.2 Using ASM

Forge provides the `IFMLLoadingPlugin` interface for mods which have customized loading procedures. It provides a method for adding class transformers, which can edit the Minecraft bytecode before it is loaded into the game. An example of bytecode can be found in Figure 6. We create a class transformer by implementing Forge's `IClassTransformer` interface, which contains a single method: `transform`. Every time a Minecraft class is initialized, the method is called with its name and bytecode. This is the general procedure if we want to change a method:

1. Check the name of the class
2. Parse the bytecode of the class to a `ClassNode`
3. Find the method in the `MethodNode` list of the `ClassNode`
4. Find the instruction in the `InsnList` list of the `MethodNode`
5. Insert new instructions
6. Remove old instructions
7. Return the modified class as a byte array

We changed the following classes:

- `EntityRenderer`
  Contains the method `updateCameraAndRender` that calls `renderWorld` to render the world. We replace it with a call to a `Raytracer` method. An easy way to do this is by having a Singleton. This removes the need to introduce a new field in the modified class. It involves adding two instructions:
  - `INVOKESTATIC` to put the Singleton on the stack
  - `INVOKEVIRTUAL` to call the static function
- `RenderGlobal`
  Contains a `ChunkRenderContainer` field that is initialized using a `VboRenderList`. It is a list of RenderChunks which is rebuilt every frame. We replace the construction of the `ChunkRenderContainer` by modifying the `NEW` and `INVOKESPECIAL` instructions to use our `RaytracerRenderList` instead. `RaytracerRenderList` overrides `ChunkRenderContainer` such that the list of `VertexBuffers` is sent to C++ instead of being drawn using OpenGL.

The sequence diagram can be found in Figure 7.

7.3 Camera construction

We unproject the corners of the screen to world space by applying the inverse view-projection matrix from the game. These coordinates, along with the player position and the vertical field of view ("FoV") from the game settings are passed to C++. With this information, we can calculate the ray origin and view pyramid, as shown in Figure 8.

```
// Java
public Renderer(Raytracer raytracer) {
    this.raytracer = raytracer;
    this.mc = Minecraft.getMinecraft();
}

// Bytecode
public <init>(Lcom/marcojonkers/mcraytracer/Raytracer;)V
ALOAD 0
INVOKESPECIAL java/lang/Object.<init> ()V
ALOAD 0
ALOAD 1
PUTFIELD com/marcojonkers/mcraytracer/Renderer.raytracer : Lcom/marcojonkers/
    mcraytracer/Raytracer;
ALOAD 0
INVOKESTATIC net/minecraft/client/Minecraft.getMinecraft ()Lnet/minecraft/client/
    Minecraft;
PUTFIELD com/marcojonkers/mcraytracer/Renderer.mc : Lnet/minecraft/client/Minecraft;
RETURN
LOCALVARIABLE this Lcom/marcojonkers/mcraytracer/Renderer; 0
LOCALVARIABLE raytracer Lcom/marcojonkers/mcraytracer/Raytracer; 1
```

Figure 6: Example bytecode for the constructor of our `Renderer` class. Note the use of Java VM signatures. `ALOAD` puts a `LOCALVARIABLE` (listed at the bottom) on the stack. Method parameters and `this` are implicit local variables.

7.4 C++

In C++, we use JNI calls to obtain the OpenGL buffer ID and index count of the `VertexBuffer`, as the fields are marked `private` in Java. Empty buffers are discarded. We register the buffer in CUDA to obtain a `cudaGraphicsResource`. Each buffer must be registered exactly once. We store the `cudaGraphicsResource` handles in a three-dimensional array (see Figure 9). The size of the array is defined by the render distance option in the game settings. The render distance is defined in chunks (minimum = 2, maximum = 16 on 32-bit Java, 32 on 64-bit Java).

The grid is indexed x, z, y. The handles are placed in the grid such that the player chunk is in the center of the x–z plane:

$$\begin{aligned} x_{grid} &= \left\lfloor \frac{x_{player}}{16} \right\rfloor - \frac{x_{chunk}}{16} + \texttt{MAX\_RENDER\_DISTANCE} \\ y_{grid} &= \left\lfloor \frac{y_{player}}{16} \right\rfloor \\ z_{grid} &= \left\lfloor \frac{z_{player}}{16} \right\rfloor - \frac{z_{chunk}}{16} + \texttt{MAX\_RENDER\_DISTANCE} \end{aligned}$$

When the player moves to a new chunk, we shift the grid along the x–z plane. This allows us to keep the vertical `RenderChunks` contiguous in memory.

After all `RenderChunks` have been passed to C++ for the current frame, we obtain a device pointer for every `cudaGraphicsResource`. These point to vertex data on the GPU and can be used in a CUDA kernel. We use `cudaMemcpy` to copy the device pointers and array sizes from host to device memory. Before we send the player position to the kernel, we transform it to chunk space:

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

This maps the player's position to $[0, 1)$, which is suitable for grid traversal.

7.5 CUDA

We define the CUDA block size to be 8x8 threads. Every pixel (assuming Minecraft's default resolution of 854x480) will run the kernel to fire a ray. The implicit CUDA variables `blockIdx` and `threadIdx` are used

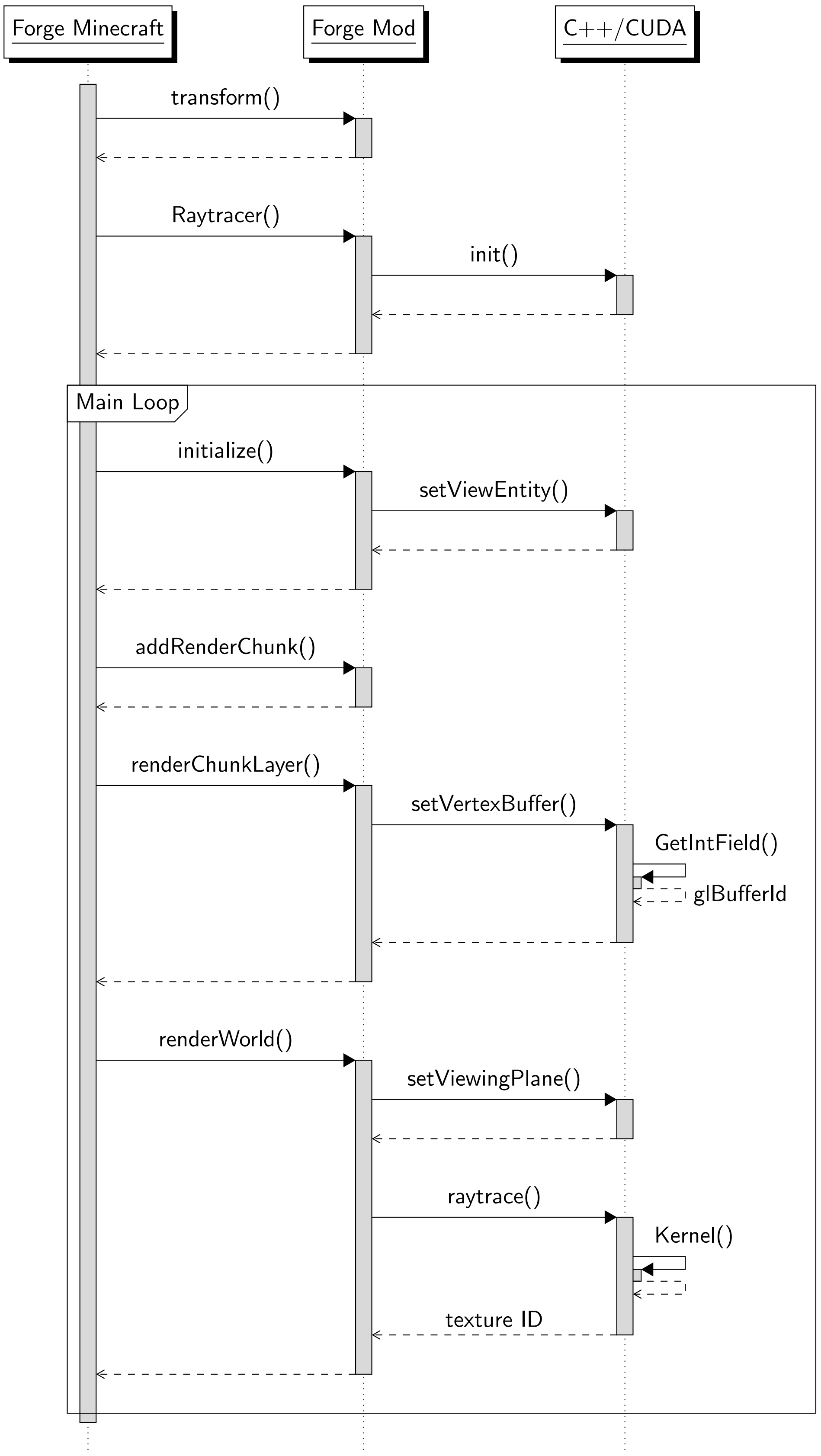
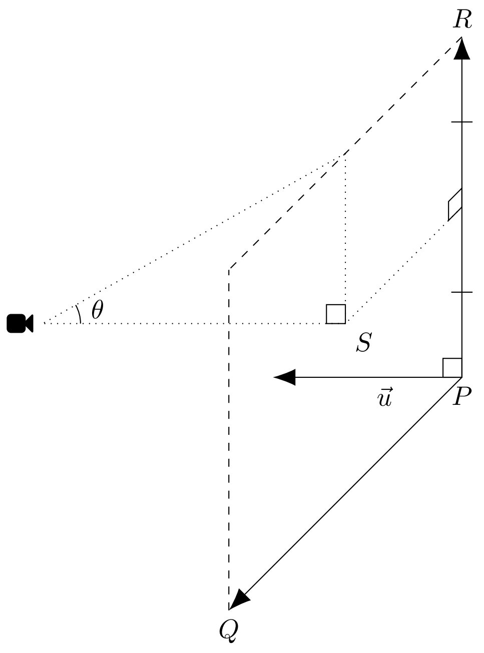


Figure 7: Sequence diagram showing interaction between the modules



$$\theta = \frac{vertical\ field\ of\ view}{2}$$
$$\vec{u} = \overrightarrow{PQ} \times \overrightarrow{PR}$$
$$S = \frac{Q + R}{2}$$
$$\text{🎥}_{pos} = S + Normalize(\vec{u}) \cdot \frac{||\overrightarrow{PR}||}{2\tan(\theta)}$$

Figure 8: Ray origin calculations

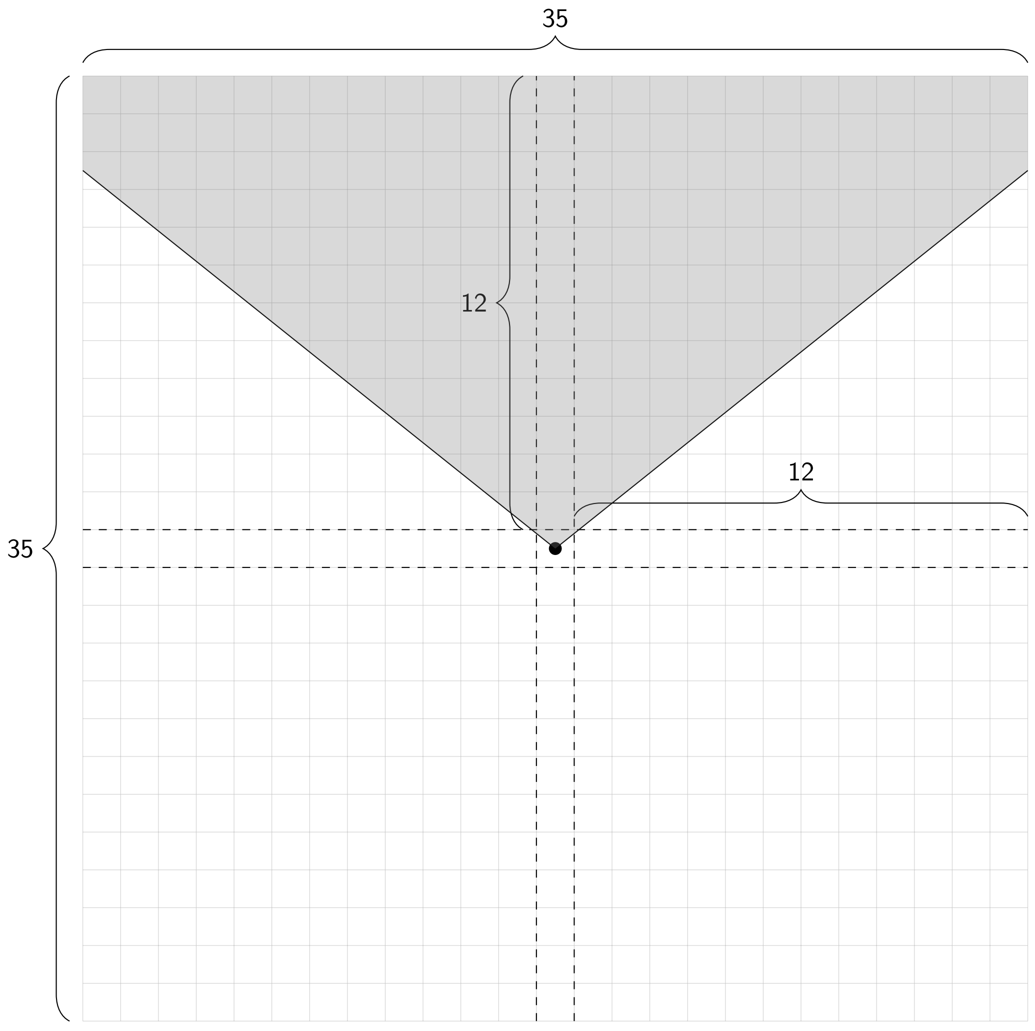


Figure 9: A top-down view of the vertex buffer array, assuming a render distance of 12. The player's view cone is shown facing north. Each cell contains 16 `RenderChunks`. When the player crosses a horizontal chunk boundary, every buffer in the grid is moved once space.

in the kernel to calculate which pixel is being processed. We calculate the ray direction for every pixel by linearly interpolating the corners of the viewport (refer to Figure 8):

$$Lerp(a, b, t) = a + t \cdot (b - a) \tag{2}$$

$$Normalize(\vec{v}) = \frac{\vec{v}}{||\vec{v}||} \tag{3}$$

$$dir_{ray} = Normalize(Lerp(P, Q, u) + Lerp(P, R) - P - pos_{camera})$$

We use this in a 3D voxel traversal algorithm from [Amanatides et al., 1987]. Amanatides et al. do not explain how to obtain the value of t at which a ray crosses the first voxel boundary. We have put the method in equation 4.

$$f(x, dx) = \begin{cases} \frac{\lceil x \rceil - x}{|dx|} & dx > 0 \\ \frac{x - \lfloor x \rfloor}{|dx|} & dx < 0 \\ \infty & dx = 0 \end{cases} \tag{4}$$

We traverse the grid until we hit a non-empty cell. The device pointer and array size from that cell are read from device memory. We perform two triangle intersection tests per quad, using the technique from [Möller and Trumbore, 2005]. If we hit the back-face of the first triangle, we can skip the test for the second triangle because they share the same normal. We can also skip the second triangle if our first hit was successful.

7.6 Results

We initially only tested for hit vs non-hit. No texturing or lighting techniques were applied. Results were disappointing. The NVIDIA Nsight profiler showed that the kernel is mostly dependent on memory reads.

8 Preprocessing Vertex Buffers

In our second attempt, we obtain the vertex buffers before they are uploaded to OpenGL, and insert the quads in an acceleration structure fit for ray tracing.

8.1 Obtaining the buffers

In order to get the raw vertex buffers from Java to C++, we change three more classes.

We created our own version of `VertexBuffer`. `VertexBuffer` contains a method called `bufferData` which uploads a `ByteBuffer` to OpenGL. We modify the `bufferData` function such that the buffer is sent to C++ instead. However, we need extra information aside from the vertex data, such as position and an id. These can be obtained from `ChunkRenderDispatcher`.

- `RenderChunk`
  We replace the `VertexBuffer` field with our `CppVertexBuffer`. There is no inheritance, so we replace all references to `VertexBuffer`.
- `ChunkRenderDispatcher`
  In charge of uploading `RenderChunks` to OpenGL. We pass an additional position parameter to the uploading routine.
- `VertexBufferUploader`
  Passes calls from `ChunkRenderDispatcher` to `RenderChunk`. We change this class to support `CppVertexBuffer`.

8.2 C++

Not ray tracing the VBOs directly means we do not have to map and unmap them every frame, which saves us quite a bit of overhead. As acceleration structure, we have chosen to use an octree. An octree lets us divide a RenderChunk evenly, while culling a lot of nodes per collision hit. It is also cheap to build, as we can iterate over the quads and fit them to a leaf. There will be one octree for every RenderChunk, which means the octree will be 4 levels deep ($16^3 = 8^4 = 4096$).

We upload the vertex buffer to CUDA instead of OpenGL.

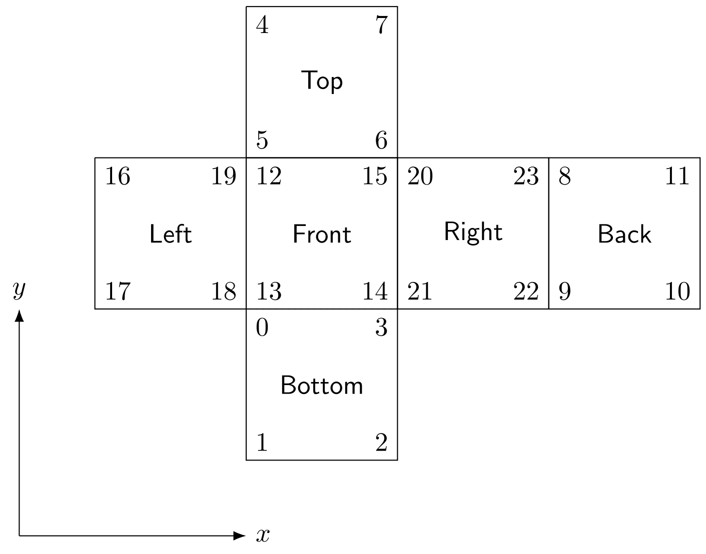


Figure 10: The net of a single cube in a RenderChunk, after Minecraft finished meshing. Numbers indicate the order of vertices in the vertex buffer. There is no index array, so no vertices are shared. Face order: bottom, top, back, front, left, right. The vertex order is counter-clockwise and starts at the top-left.

8.3 CUDA

8.4 Results

9 Conclusion

10 Future work

We 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

[Amanatides et al., 1987] Amanatides, J., Woo, A., et al. (1987). A fast voxel traversal algorithm for ray tracing. In *Eurographics*, volume 87, pages 3–10.

[Möller and Trumbore, 2005] Möller, T. and Trumbore, B. (2005). Fast, minimum storage ray/triangle intersection. In *ACM SIGGRAPH 2005 Courses*, page 7. ACM.

[Oracle, 2016] Oracle (2016). JNI Types and Data Structures. `https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/types.html`. Accessed: 2017-04-10.