Voxel Ray Tracing

Yohan Guyomard
yohang@mit.edu

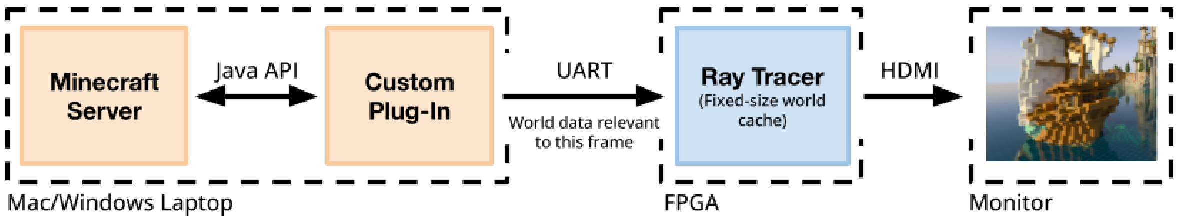
Win Win Tjong
winixent@mit.edu

29 October 2024

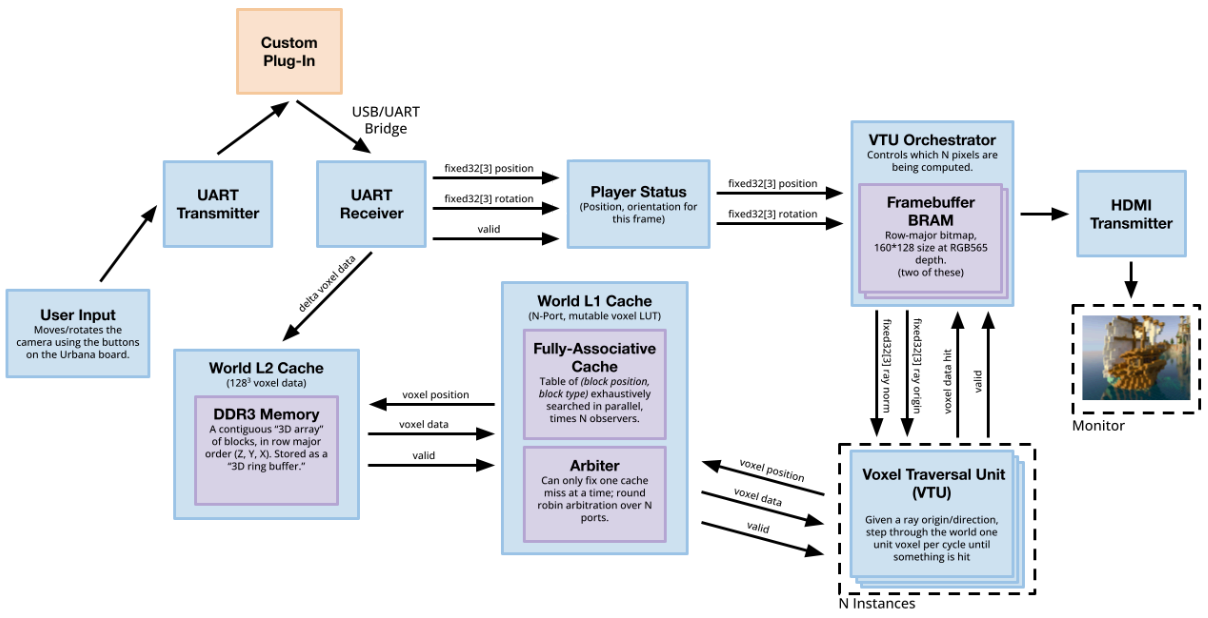
Abstract

We propose a minimal Minecraft client running on an FPGA, capable of streaming world data from a server and rendering it. We intend to use ray tracing for our graphics pipeline, taking full advantage of FPGAs to parallelize and optimize (in hardware) voxel traversal. We will author a plug-in for the Minecraft server that communicates with our FPGA directly over UART; implementing a TCP stack and Minecraft's handshake protocol is *not* within the scope of this project. This allows us to cherry-pick what is exchanged between the server and our system — namely, the blocks around the camera for this frame, the camera's orientation and user inputs.

Block Diagram



At a (very) high level, this is what we're proposing. In more details:


Custom Plug-In
USB/UART Bridge
UART Transmitter
UART Receiver
fixed32[3] position
fixed32[3] rotation
valid
Player Status
(Position, orientation for this frame)
fixed32[3] position
fixed32[3] rotation
VTU Orchestrator
Controls which N pixels are being computed.
Framebuffer BRAM
Row-major bitmap, 160*128 size at RGB565 depth.
(two of these)
HDMI Transmitter
Monitor
User Input
Moves/rotates the camera using the buttons on the Urbana board.
delta voxel data
World L2 Cache
(128³ voxel data)
DDR3 Memory
A contiguous "3D array" of blocks, in row major order (Z, Y, X). Stored as a "3D ring buffer."
voxel position
voxel data
valid
World L1 Cache
(N-Port, mutable voxel LUT)
Fully-Associative Cache
Table of (block position, block type) exhaustively searched in parallel, times N observers.
Arbiter
Can only fix one cache miss at a time; round robin arbitration over N ports.
voxel position
voxel data
valid
fixed32[3] ray norm
fixed32[3] ray origin
voxel data hit
valid
Voxel Traversal Unit (VTU)
Given a ray origin/direction, step through the world one unit voxel per cycle until something is hit
N Instances

Server to FPGA Pipeline (Win Win)

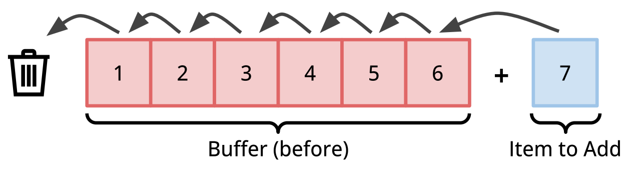
Plugin Protocol / UART Receiver

Amongst the selection of protocols for communication between the game and the FPGA, we have decided to use UART instead. This protocol is simple and straightforward, which allows for ease of implementation and debugging. Moreover, a useful feature of UART is the asynchronization between the receiver and the transmitter, which is applicable in this situation where the transmission and receiving of data may not align at the same rate. Block data will be transmitted in three different scenarios: the first time with no value input, when player movement within one unit of X, Y, and Z direction is detected, and when a single block of a particular coordinate changes to another coordinate.

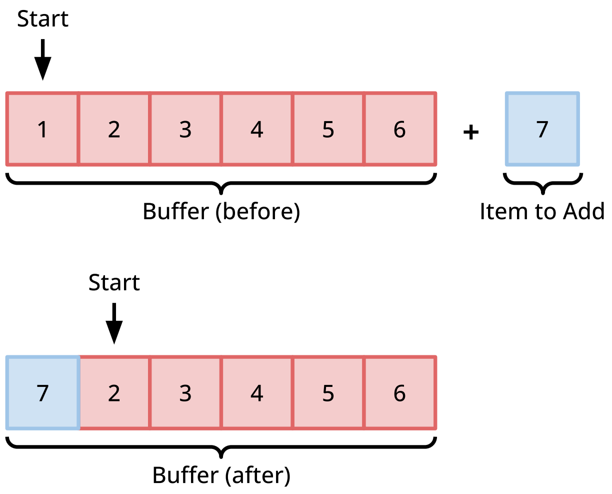
Because there is no direct way for the game (Minecraft in this case) to interact with UART, we will need some form of an intermediary. Our approach for this is to write a custom plug-in in Java that will do this instead.

World L2 Cache

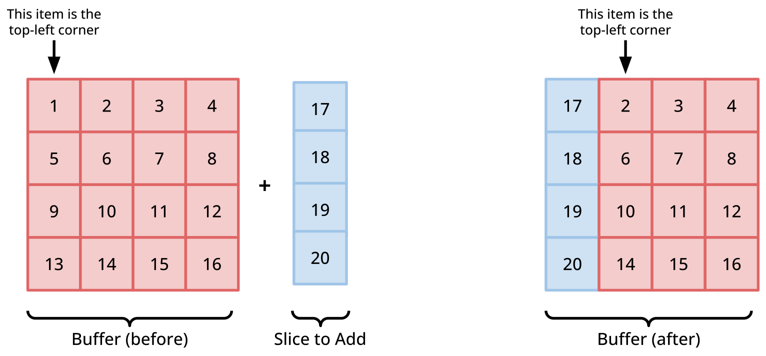
Transferring a whole set of world data through UART poses some problems, with the most prominent one being the rate it performs at. For context, we plan on working with 128^3 voxels, and no matter which baud rate we choose to use, it is not possible to transfer data of this size in under a second. To tackle this challenge, we propose an alternative where instead of updating the entire 128^3, we shift the plane of data over into a unit of certain direction. For example, if the player is detected to have moved one unit in the positive X-direction, the data in the YZ plane will be shifted from X to X+1. This way, instead of transferring 128^3, we instead are transferring 128^2, which is more plausible with a minimum baud rate of 19,200 bps. However, the issue still persists because the UART has a limit on how much data it can transfer, which will make it impossible to render at a stable 30 FPS. Moreover, Real Digital Urbana board has a maximum DDR3 capacity of up to 128 MB, so updating the entire cache will take too long, causing the transmission process to have to wait. To fix this, we decided to use a ring buffer, where new data replaces an old data in the cache, but it still allows other data in the cache to still be accessible. This way, we will not have to wait for the cache to update fully before transmitting it.



A suboptimal ring buffer which shifts every element.



A better ring buffer just tracks the index of the first element.



Same premise as ring buffer, but in 2D. You can imagine what shifting in a slice in the Y direction would look like, and likewise for the 3D case. For our system, we also need to track what "top-left corner" corresponds to in world coordinates.

Player Status

The player's data relative to the world data in Minecraft can be obtained from the game's server data files. We are using the server data files instead because it updates the game's data at a constant interval. Because ray tracing follows the path of lights from the viewer, which in turn means where the player is located at, it is crucial to have the most recent player's data to update the rendering.

User Input

We plan on utilizing the buttons provided on the board as player's input. Similar to how the user's input was utilized in Pong, the input for this will control the player's next state, which will then update the cache.

Rendering Pipeline (Yohan)

Voxel Traversal Unit

Ray tracing involves broadcasting rays into the scene and, in its simplest form, returning the albedo of the object hit. This is performed per-pixel, for each frame, so we need an efficient algorithm for traversing voxel worlds. We will be implementing one such algorithm proposed in *A Fast Voxel Traversal Algorithm for Ray Tracing*; its selling point is that "Going from one voxel to its neighbor requires only two floating point comparisons and one floating point addition." (Amanatides and Woo).

We propose a "voxel traversal unit" (VTU) module which, given a ray origin and direction, advances through the world one unit voxel at a time. It does this until a solid voxel (i.e. *not* air) is hit. This is essentially a pipelined version of the original algorithm, meaning that each step involves at most one voxel query (as opposed to e.g. eager look ahead or attempting to combinatorially trace the entire ray). We intend to instantiate *N* of these modules to compute *N* pixels in parallel.

Implementing floating point arithmetic is a non-goal of this project. Instead, we'll opt for a fixed point representation of world coordinates.

VTU Orchestrator

The orchestrator is both our swapchain and, as the name suggests, VTU manager. We will use two frame buffers, one for display and the other for compute, to avoid screen tear. Whenever a frame is computed, the orchestrator waits for a *vsync* signal and swaps the two buffers.

Given *N* VTU's, the following need to be controlled:

- ❖ What pixel should the ith VTU compute?
- ❖ What is the ray origin/direction, in world space, of that pixel?
- ❖ Once the ith VTU has finished its computation, which pixel should it be reassigned to, if any?

The orchestrator ensures that the entire framebuffer is populated, while avoiding duplicate work (i.e. two rays for the same pixel). This will be implemented as a counter going from the top-left to bottom-right pixel; it increments each time a VTU has finished its computation. It will also account for the possibility of multiple VTU's finishing at the same time.

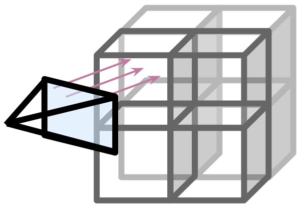
The orchestrator computes the ray origins/directions from the camera's location controlled by the *Player Status* module.

World L1 Cache

Each "step," a VTU will query one voxel. The most straightforward way of doing this would be using a BRAM (or DDR3) to store the voxel data. However, this would limit us to *N=1* VTU's, *maybe N=2* if we use a two-port BRAM. Any higher, and we would have more observers than our memory has ports; so, an arbiter would be introduced and our entire system would be bottlenecked by that.

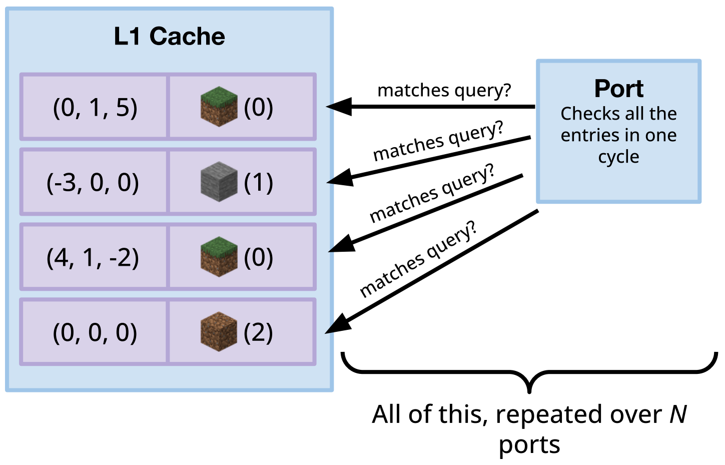
Another approach is having one BRAM cache *per* VTU, which queries some larger DDR3 RAM upon cache misses. This approach is actually completely pointless, since a ray will only ever move forward and voxels are visited at most once per VTU per ray (i.e. cache hit rate = 0%).

We propose a shared cache, across all VTUs. This works on the assumption that nearby pixels (and thus rays) will need to query the same voxels. In the best-case scenario, it is like querying the voxels for each ray all at the same time.



All the nearby rays (computed in parallel) need to query the same voxel. If it's a cache miss at first, it will eventually be resolved to a cache hit for all of them at the same time.

The world L1 cache supports *N* observers and resolves cache misses with the *L2 cache* module. This will be implemented as a fully-associative cache — an array of *(voxel position, voxel data)* that's exhaustively searched (combinatorially) for each request.



In the event of one or more cache misses, the L1 cache replaces its oldest entry. Only one cache miss can be resolved at a time (since there is one port to the L2 cache), so we plan to experiment with different arbiters (e.g. round-robin, fixed-priority, etc.).

HDMI Transmitter

We intend to use the same HDMI module as used in class. It will be routed appropriately to the correct framebuffer by the *VTU Orchestrator* module.