Voxel Ray Tracer Final Report

Yohan Guyomard
Massachusetts Institute of Technology
Cambridge, MA, USA
yohang@mit.edu

Win Win Tjong
Massachusetts Institute of Technology
Cambridge, MA, USA
winixent@mit.edu

***Abstract*—We propose a ray tracing engine capable of rendering vast voxel worlds, in realtime and implemented entirely in hardware on a Field Programmable Gate Array (FPGA). Our intended use-case is as an alternative graphics front-end for the video game *Minecraft*; our system leverages the existing gameplay mechanics and world generation/storage infrastructure of the game, and focuses entirely on computer graphics. Our renderer is highly-parallelized and as such, we present a novel memory hierarchy built specifically for this problem.**

***Index Terms*—Digital Systems, Field Programmable Gate Arrays, Computer Graphics, Ray Tracing, Voxel**

I. Introduction

A voxel world is a discrete representation of 3D space, where each unit cell is axis aligned and associated with some information. In our case, each voxel is a *Minecraft* "block" and the overall world depicts user-traversable sceneries like mountains, lakes, plains, etc. Using algorithms from computer graphics, snapshots of this data structure can be conveyed visually through a monitor. Our system does just that; we use the Xilinx Spartan-7 Urbana Board to render *Minecraft* worlds to an HDMI display.

Ray-tracing is a rendering technique which simulates the path of light by shooting rays from the camera into the scene. It is trivially parallelize-able as each pixel is computed independently from its neighbors. However, millions of pixels must be computed for each frame and this technique is generally reserved for high-end hardware. To achieve this, we introduce the *Voxel Traversal Unit* (VTU) – analogous in function to computing exactly one pixel (at a time), and instantiated many times to take full advantage of FPGAs' reconfigurable hardware.

We utilize a *Minecraft* server as the source-of-truth for the world's state, and stream voxel data as needed using a custom plug-in and the Urbana board's built-in USB/UART bridge. This necessitates some arbitration scheme, as N VTUs cannot simultaneously access the one UART bus lest our system be severely bottlenecked by the arbiter. We overcome this through a series of caches, ranging in size and access-speed. Our system aims to minimize the FPGA/server exchanges and VTU stall time, hence rendering frames faster.

II. Voxel Traversal Unit

A crucial aspect of our project, and ray tracing overall, is scene traversal — given some starting point and direction, what is the first object that a ray has hit? An efficient implementation of this procedure is necessary for "acceptable" frame-rates. The VTU is responsible for this, and steps through the world one unit voxel at a time until it has collided with a non-empty block (i.e. "not air"). Note that the number of steps is indeterminate, and is limited by either the bounds of the world or some render distance limit in the worst-case.

A. Fast Voxel Traversal

We leverage the *Fast Voxel Traversal* algorithm proposed by [1], which requires only two numeric comparisons and one addition to go from a voxel to its neighbor. The setup for this method is a little more involved, but we expect that minimizing the time per-iteration is absolutely critical.

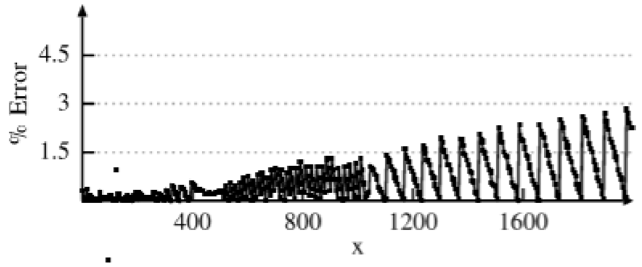
At each step, the VTU queries the voxel at its current location and breaks if it's hit a non-transparent block. In SystemVerilog, this is encoded through a `valid` signal which other subsystems use to orchestrate N VTUs' operation. We require each iteration to take a minimum of two clock cycles, which is a limitation of our memory subsystem. In some cases, we are able to traverse four blocks at a time (see Section III), and this extra cycle is used to compute eager-lookaheads without needing to alter [1] or risking negative slack.

B. Fixed Point Arithmetic (Yohan)

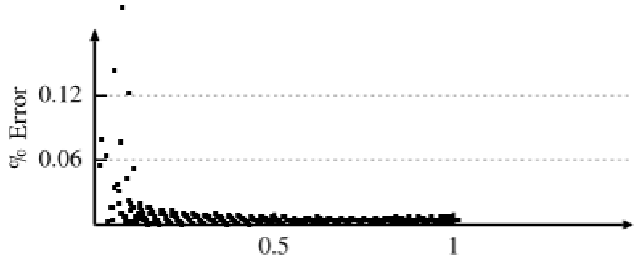
The original algorithm proposed in [1] uses floating point numbers for its computations. Implementing IEEE-754 is out of scope for this project, so we opted for a fixed-point representation instead.

Basic operations like addition, comparisons, etc. are trivial to implement, if at all changed from the underlying representation. We also optimize two common operations: `fixed_fast_inv_sqrt` and `fixed_fast_recip_lte1`. These methods are used in the setup routine of voxel traversal, but are also ubiquitous to computer graphics and will likely manifest elsewhere. The optimizations are motivated by the fact that addition, subtraction and multiplication are cheap in hardware, but long division or otherwise is unfeasible.

`fixed_fast_inv_sqrt` is taken directly from Quake [2] and uses a lookup table instead of floating point hacks; two iterations of Newton's method are used. `fixed_fast_recip_lte1` builds on this idea, allowing for division by numbers ≤ 1 in a fixed number of cycles (untested whether this can be done in one cycle).



Graph 1: Fast inverse square root approximation



Graph 2: Fast reciprocal (<= 1) approximation

As shown above, these approximations can perform quite well and should be sufficient for our use cases.

C. Shading (Yohan)

Once a VTU has intersected with a block and finished traversal, it passes its result to a shader module for computing the pixel's final color. We use simple lambert shading as discussed in [3]. This involves a dot product between the normal of the ray's intersected voxel and some incident light source – while the results aren't photorealistic, it's convincing enough considering the simplicity of this method.

Better results can be achieved by computing more rays per pixel. For instance, determining whether a voxel is illuminated involves casting a ray from it to the light source, and checking whether something is in the way. This yields direct-illumination shadows, and is a major improvement over Lambert shading. Reflections can be computed in a similar fashion by setting the secondary ray's direction to the surface normal of the first hit. Global illumination, radiosity and path tracing are possible extensions of this approach, but are out of scope for this project.

The diffuse color of each voxel is encoded in a lookup table, mapping 8-bit `BlockType`'s to `RGB565` values. We currently do not support texture mapping, however the process would be similar with a LUT.

D. VTU Orchestrator (Yohan)

We introduce the orchestrator which, given N VTUs, assigns a batch of pixels to compute in parallel; calculates ray's directions; and writes to a frame-buffer.

This is implemented as a simple arbiter that prioritizes VTU `#1`, then VTU `#2`, VTU `#3`, etc — each cycle, it chooses the highest-priority VTU that *isn't* busy (i.e. its `valid` flag is asserted). The orchestrator then writes that VTU's (valid) output to the frame-buffer, changes its ray origin/direction inputs and increments the index of the next pixel in line. This introduces some throttling, however it is unlikely that any given VTU can significantly outpace the orchestrator so this approach is sufficient.

Pixels are computed in row-major order, and are combinatorially assigned a ray origin and direction (note that this computation only occurs for the "next pixel in line", *not* all at once). We use the pinhole camera model and an orthographic projection with configurable focal length. Thus, every ray has the same origin – the camera's position – and we're pre-computing the camera's orthonormal basis each frame (world orientation $\longrightarrow$ screen space) such that computing each ray's direction involves only two additions and multiplications.

The frame-buffer is stored in BRAM, which limits the resolution of our renderer. We choose to embrace this as a stylistic choice, and render at 256 by 128 pixels, `RGB565` bit-depth.

III. Voxel Caches

We originally conceived a novel memory architecture inspired by modern CPUs, but tailored to our specific use case. It consisted of three levels of caches:

- `L3`: The largest, and slowest, it stores 128^3 voxels in DDR3 RAM. This is the FPGA's "copy" of the *Minecraft* world, and is updated each time the player moves.
- `L2`: This has N ports for simultaneous queries, and supports single-cycle accesses. Cache misses are resolved with the `L3` cache.
- `L1`: Each VTU has a sparseness representation of the world (1-bit "is air or not air?") stored in its own BRAM.

A. L3 Cache (Win Win)

Rather than directly streaming the voxel data from the server client into the L2 cache, we decided to first store those incoming data into a buffer first. Due to the sheer amount of data being streamed through, updating the entire cache for every time a frame updates would cause output to refresh at a rate nowhere near 30 frame per second (FPS). Instead, an effective strategy we decided to proceed is to utilize a version of the ring buffer [4]. The version we implemented has a shape that is cubic instead of a ring as it makes easier to visualize. However, the concept is the same in the way it updates: instead of shifting a whole set of data to make room for a new one, we use a pointer to indicate the front-facing data to indicate where and how the next state should update.

At first, we thought about sampling 128^3 worth of voxels with each voxel be 5 bits wide. However, due to our intended resolution which is much lower than most average display resolution, we decided to reduce it to a smaller dimension, which allowed us to fit it all into the block ram (BRAM) instead. The intended type of storage we planned on using was the DDR3, and although the advantage of a DDR3 storage is its size, we decided to use the BRAM instead due to its flexibility to read and write simultaneously. This way for every new frame, the buffer can start to write the oncoming plane of voxels and also read from the buffer on the same cycle. According to the AMD Product Guide, the Spartan S7 50 chip has BRAM capacity of approximately 2.7 Mbit. If we reduce the sample size to at least 64^3, with a 5-bit representation of voxels, we estimated about 1.3 Mbit of the 2.7 Mbit will be used, which is enough to fit the entire cache.

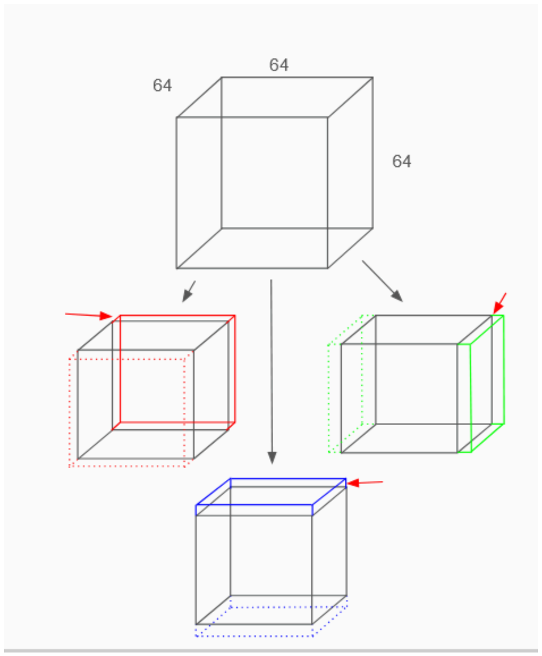


Fig. 1: The three possible ways the buffer updates depending on the axis where the player's location is displaced towards

B. L2 Cache (Yohan)

The `L2` cache supports simultaneous single-cycle accesses from the N VTUs. It is based on the assumption that only a subset of the world is visible each frame and that rays will frequently query the same blocks, especially if they are computed in parallel. Thus, it doesn't need to be large and we are able to use a fully associative cache as its underlying representation.

```
module l2_cache #(PORTS, CACHE_SIZE) (
  // -- snip--

  // N-Ports Interface
  input  BlockPos  [PORTS-1:0] addr,
  output BlockType [PORTS-1:0] out,
  output logic     [PORTS-1:0] valid
);

  BlockPos [CACHE_SIZE-1:0]  tags;
  BlockType [CACHE_SIZE-1:0]  entries;
```

Above is the SystemVerilog code for an N port interface. The entire cache can be thought of as a list of `(voxel position, voxel type)` which is combinatorially brute-searched each cycle (i.e. all at the same time).

In practice, this behaves quite well. Although this paradigm did not make it to our final product, simulation results showed around 50% cache-hits on average (4 ports, 32 entries in cache).

C. L1 Cache (Yohan)

An additional cache is motivated by the fact that we can't scale the `L2` cache to store an arbitrarily large amount of voxels. Moreover, we assume that *Minecraft* worlds are sparse (mostly air), and that the `L2` would ultimately be filled with just air blocks — it should only require 1 bit to store whether a voxel is air or not, what a waste!

Ultimately, ray traversal is only interested in the voxel it's intersected, not the intermediary ones. Thus, we compute a sparseness representation of the world and assign a copy to each VTU. This is stored in BRAM, so we have to down-sample the 128^3 voxels in `L3` by a factor of 4, yielding $\left(\frac{128}{4}\right)^3 = 32,768$ bits, or just small enough to fit.

In this cache, each `0` signifies that all of the voxels in that 4^3 sector are air and can be skipped. A `1` means the VTU should fallback to the `L2` cache.

Another common approach to sparseness is the voxel octree data structure. It is generally more compact, but more difficult to compute. Since we are regenerating the `L1` cache each frame, we opted for this simpler approach.

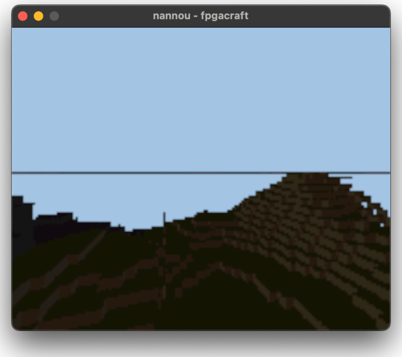
D. In Reality

Unfortunately, the L3 cache never worked as intended. Interfacing with the DRAM proved more difficult than originally planned. Therefore, we decided last-minute to omit our entire "fancy" cache architecture, and the L3 cache would instead be stored in BRAM. As such, we reduced the chunk size from 128^3 to 64^3.

However, with this paradigm, there are very few benefits to `L1` or `L2`, since everything is in BRAM (2 cycle access). It's a shame, since `L2` had already been implemented, however we decided that every `VTU` would directly interface with the `L3` cache (a thin abstraction over a 2-port BRAM). Then, every `L3` is updated at the same time (i.e. their write ports are interconnected) and read by at-most 1 `VTU` (i.e. their read ports are *not* interconnected).

IV. Software Implementation (Yohan)

A software-only version of this project has been implemented in Rust. It is used to characterize real-world behaviors of caches, e.g. hit/miss ratio and whether our architecture is actually beneficial.



Although software is not the focus of this class, this approach was *immensely* beneficial. Graphical and algorithmic quirks were able to be identified using a debugger, and the rapid turn around time helped tremendously. Moreover, the code was organized such that porting to Verilog would be easier:

```
// Very weird to code this way! But it makes
sense, since this ultimately ends up on the
FPGA
impl TopLevel {
   pub fn rising_clk_edge(&mut self) {
      // Propagate signals to owned
submodules
      self.orchestrator.reset = self.reset;
      self.orchestrator.rising_clk_edge();

      if self.reset {

self.orchestrator.camera_heading_in =
Vec3::FORWARD;
         self.orchestrator.camera_pos_in =
Vec3::default();
      }
   }
}
```

The Verilog implementation was tested to give exactly the same results as above. However, the software implementation renders in around `500ms` whereas cocotb can take tens of minutes. We are also characterizing the performance of the caches and our overall architecture (roughly) by logging some statistics each frame:

```
This frame had 3346782 accesses with
49.83148% hit-ratio
Took 3367185 cycles, simulation ran for
501ms
This frame had 3346653 accesses with
49.83146% hit-ratio
Took 3367056 cycles, simulation ran for
491ms
This frame had 3346616 accesses with
49.83144% hit-ratio
```

V. Communication Protocol

A. UART Receiver (Win Win)

Amongst the choices for which communication protocol we can use, we decided to proceed with UART due to the benefits that it provides. We are aiming for an output rate of 30 fps, and for a single frame update, we expect to transmit $5 * 128^2$ bits of data to update the buffer in the `L3` cache. Overall, we expect a transfer rate of $30 * 5 * 128^2 = 2$ Mbit of information per second. Coincidentally, the UART in the Urbana board is capable of transferring at a rate of 2 Mbit/s, which meets our specification.

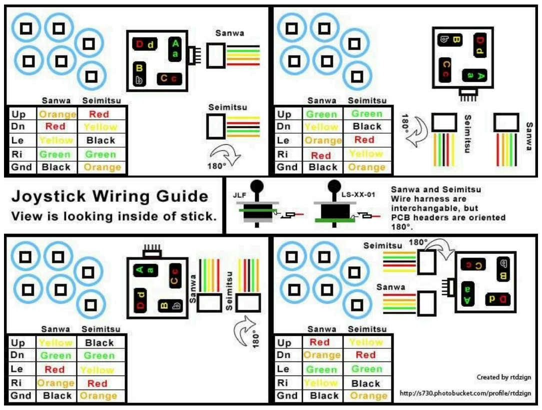
Another feature that drew us to using the UART is the asynchronization rate of transmission and receiving. We expect that the data transmission from the server may not align in time with the rate in which it is received at the `L3` cache, so to ensure that these two rates do not affect one another, setting them asynchronously was the best choice to go with.

B. UART Transmitter (Win Win)

The UART transmitter works in opposite to the receiver, where it will take the user's input commands and send it to the Minecraft server client, where it then will update the current game status. The input command data that will be transmitted will not be as intensive as the UART receiver, and the rate of transmission will most likely not be as fast to it, which is why the asynchronous feature between the two is very helpful for our purpose.

C. User Input (Win Win)

A frame will update only if the player is detected to have been displaced from their current position. To trigger this displacement, an input is needed by the player to determine the direction to move towards. At first, we thought that the built-in buttons on the Urbana board could be used as the input. However, we realized that will not work due to other usage for the buttons. So instead, we chose to use arcade joystick for control input, which will provide us with the same 4 inputs (forward, backward, left, and right).


Joystick Wiring Guide
View is looking inside of stick.
Sanwa
Seimitsu
180°
JLF
LS-XX-01
Sanwa and Seimitsu Wire harness are interchangable, but PCB headers are oriented 180°.
Created by rtdzign
http://s730.photobucket.com/profile/rtdzign

	Sanwa	Seimitsu
Up	Orange	Red
Dn	Red	Yellow
Le	Yellow	Black
Ri	Green	Green
Gnd	Black	Orange

	Sanwa	Seimitsu
Up	Green	Green
Dn	Yellow	Black
Le	Orange	Red
Ri	Red	Yellow
Gnd	Black	Orange

	Sanwa	Seimitsu
Up	Yellow	Black
Dn	Green	Green
Le	Red	Yellow
Ri	Orange	Red
Gnd	Black	Orange

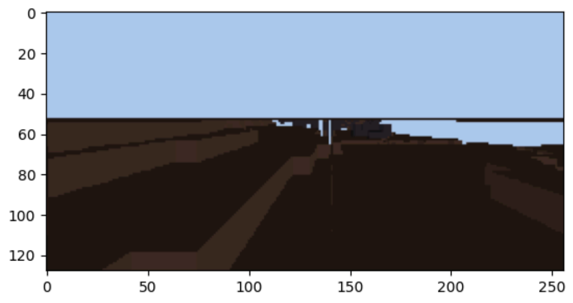
	Sanwa	Seimitsu
Up	Red	Yellow
Dn	Orange	Red
Le	Green	Green
Ri	Yellow	Black
Gnd	Black	Orange

Fig. 2: The wiring schematic of an arcade joystick comprised of 4 limit switches.

A joystick is commonly comprised of two potentiometers, but an arcade version of the joystick is instead comprised of 4 limit switches. The convenience of this type of this switch compared to the potentiometers variant is that this one provides digital signal without the need to have analog to digital converter (ADC). By connecting the joystick to the first four pins of PMOD A I/O pins, each button input in the joystick can concatenate into a 4-bit value. By padding it with four additional 0s, this can form a byte packet which is then transmitted through UART into the plugin to update the server.

VI. Evaluation

A. Voxel Traversal Unit + Orchestrator (Yohan)



The voxel transversal unit (VTU) and orchestrator had been tested to work as intended. The following is an image rendered from our system, in simulation (cocotb). Using 1 VTU, it took 38 milliseconds.

B. L3 Cache (Win Win)

We decided to change our approach for the L3 cache from using DDR3 to BRAM instead solely for its doable implementation and simultaneous read and write of data. While cache was able to do as intended on the BRAM, the ring buffer style state change never came to fruition. Although it was tested to make sure it does as it was intended, it did not translate well to the final implementation.

VII. Retrospetive

A major area of development would be identifying the key elements of the project and devise contingencies in place in case it does not work. We did not consider, at first, that the L3 cache could work with the BRAM under reduced sample size. The late change that took place for the implementation has costed us the time that could have been used to ensure intended behaviors translate when it is integrated with the rest of the modules.

Another possibility to worked on was to utilize a second FPGA hardware for its BRAM. This way, the size limitation are expanded to allow more modules – especially the three different levels of caches – to take advantage of its features.

VIII. Contribution

Yohan handled the main ray tracing portion of this project, which includes the algorithm and any dependency for it. His knowledge of Rust and Java programming language allows for the implementation of the game's server and plugin to set up serial communication.

Win Win worked on communication and data storing between the game and the FPGA. He also worked on HDMI to stream output of the ray traced algorithm on to an external device.

IX. Resource Code

The resource codes can all be found in the following repository link: https://github.com/yohandev/fpga-final-project.git.

Acknowledgements

We would like to thank Joe and the staff team of 6.2050 for providing insightful guidance and support throughout the project.

References

[1] J. Amanatides and A. Woo, "A Fast Voxel Traversal Algorithm for Ray Tracing," in *Eurographics*, 1987. [Online]. Available: https://api.semanticscholar.org/CorpusID:60696902

[2] Wikipedia contributors, "Fast inverse square root — Wikipedia, The Free Encyclopedia." [Online]. Available: https://en.wikipedia.org/w/index.php?title=Fast_inverse_square_root&oldid=1256998505

[3] "Introduction to Shading." [Online]. Available: https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/diffuse-lambertian-shading.html

[4] Wikipedia contributors, "Circular buffer." [Online]. Available: https://en.wikipedia.org/wiki/Circular_buffer