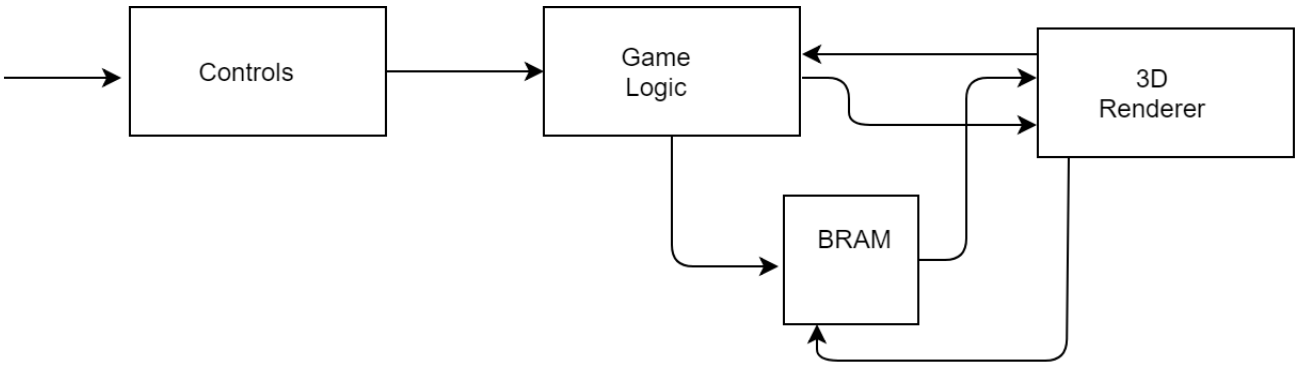
Interactive Minecraft

Chenkai Mao *chenkai@mit.edu*

Alexis Camacho *camacho1@mit.edu*

1 Overview (Chenkai)

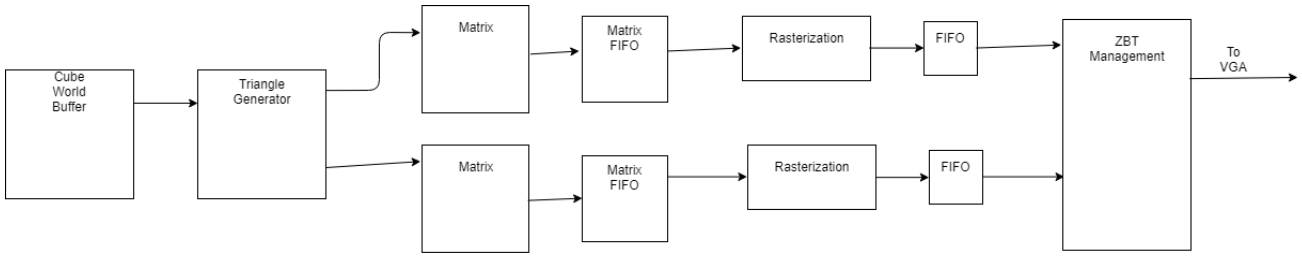
For our project we started intending to implementing a simplified but interactive version of Minecraft on a FPGA, in which we create and display a 3D world made of different blocks, and the player navigates himself around under first person view using controllers like accelerometers and gyros attached to their body, in the same time the user make interactions with the world like mining and moving blocks, high level block diagram is shown below, with the controller for navigation and interaction, game logic responsible for the interaction logic, BRAM for storing the world state and graphics information, and 3D renderer for calculation and displaying the first person view.



It turns out that making a 3D graphics pipeline in FPGA has many difficulties that we didn't foresee at the beginning, and in the end we only finished implementing the graphics pipeline, BRAM and a controller implemented with FPGA buttons. Still, it is quite an exciting project and in this report we'll introduce here our block diagram in section 2, talk about the design and implementation in detail in section 3, and the difficulties we faced and the optmization we used in section 4 and attach some results and reference in section 5.

2 Block Diagram (Alexis)

The very high level block diagram is plotted here:



3 Design and Implementation (together)

As shown in the block diagram, the tipical data flow can be described in the procedures below. Where subsections (a) to (e) are implemented by Chenkai and (f) to (j) are implemented by Alexis.

(a) World BRAM

We start encoding our world state in BRAM, with size $5 * 32768 = 163840bits = 20.48kB$, where each line is 5 bits, with 3 color bits and 2 state bits, and in total $32^3 = 32768$ lines, representing our world which is 32 by 32 by 32 blocks in size. For color bits, $2'b00$ is default sky blue, and other colors incluide leaf green, grass green, tree trunk brown, stone grey and brick brown. For state bits, $2'b00$ means there is no cube in that position, $2'b01$ means there is a block there that we need to dosplay, we didn't have time to implement but intended to use $2'b10$, $2'b11$, for different states useful in game logic.

(b) Controller

We also have a controller module storing and updating all the user information, like position (three coordinates), orientations (two angles) etc. It takes in the FPGA button signals and update the user state that will change the view accordingly in real time.

(c) Triangle generator

Next step is keep looping through the world and update it on the screen. We have a counter looping through each block address in BRAM. If the state bits are

$2'b01$, which means there really is a block there, we read it from BRAM and pass the cube information(color, state and address which is the position of a vertice) to a module named triangle generater. Based on the user information(position and orientation) and cube information, we can have simple logic calculation that whether the block is in front of the player and lies inside the viewport. If it is then we generate 6 triangles and pass on to matrix module for next step calculation(due to hardware limitation we only have 2 matrix modules so we output 2 triangles each clock cycle and pipeline for three clock cycles for outputing all of them), if not we stop and take in next cube information. At each time the player can only see at most 3 faces of a cube, and each cube can be represeented by 2 triangles, which is why we generate 6 triangles for each cube. Another trick here is when we're generating triangles, we know the normal direction of that cube and we can encode it into a 2 bit shader that determines the brightness of that face in the end to show simple shader effert.

(d) Matrix and mat-ras-fifo

Then each triangle is passed to the matrix module, which applies matrix operations for coordinate transformation and perspective projection, which transforms the triangle in 3D space onto 2D screen coordinates along with the depth information, which is the z-distance of each triangle vertice to the player. The calculation is pipelined so that it can take in triangles at every clock cycle. Then the transformed triangle is fed to a fifo between matrix and rasterization module, which we call mat-ras-fifo. The reason we need a fifo is that we want to parallelize the design with one matrix connected to multiple rasterization modules, so we need a fifo in between to hold the triangle data.

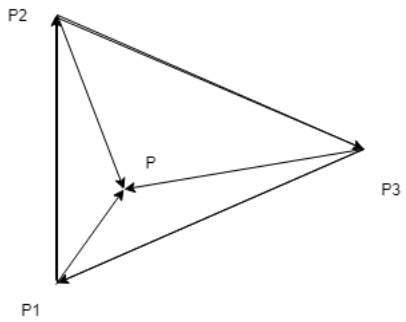
(e) Distributor

In order to distribut the triangle data in fifo to different rasterization modules, we have a counter or we call it distributer after the mat-ras-fifo that is connected to both the fifo and multiple rasterization modules. It takes in the ready signal of each rasterization module, along with the empty signal from fifo, distributing the triangle data to each rasterization module.

(f) Rasterization

Before beginning talk about rasterization, it is a good idea to know what the process is. We are given three screen points that describe a triangle and an inactive screen, where pixels are turned off. Based on these three points, which

pixels should be turned on? This problem can be solved by iterating through the pixels and only turning on pixels that lie within the triangle. We can determined whether a pixel lies in a triangle through cross products. Consider the following figure.



If $(P - P_1)X(P_2 - P)$ z part is positive number, this means that P lies within the right of the line. If this holds for all three of the vectors, the point must lie within the triangle. It is important to know that this only holds if the points are defined in a clockwise fashion, otherwise the cross product will be negative. This is important in our implementation of Rasterization, since we must consider both scenarios.

When we iterate through the pixels it is important to be efficient since 3D rendering is costly. Therefore we only iterate through the bounding box of the triangle, that reduce our area to the ranges of possible candidates.

Now that we have the mathematics down, we can implement Rasterization.

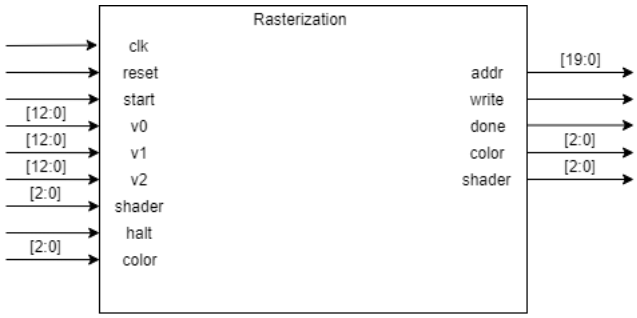
The rasterization module was implemented as a state machine, whose main states included performing initial calculations, and returning pixel information at every clock cycle. Initially, the Rasterization module is in a waiting mode if there is no information being processed. Once a signal is given to the Rasterization Module to start, inputs to the module are processed to be used in the iteration of pixels. These initial calculations include, the bounding box of the triangle, the edge equation coefficients, and pseudo-depth information.

We store edge equation coefficients in a register, and set h-count and v-count values to begin with the minimum x and y values and soon after move into the iteration phase.

During this phase, we iterate through each pixel every clock cycle to determine whether the pixel should be filled or not, and give a registered output of the

pixels color. We must also consider if the points were declared counterclockwise or clockwise and apply the right edge equations. This is done by checking the normal-z and edge equations. Furthermore, we consider whether a h-count has gone past x-max, so we can iterate through the next row or if we can stop iterating (if v-count is past y-max).

Based on our decision in iteration phase, we output color information, pixel location, depth, and whether to write (ie in triangle). Another output, done, is asserted only after we finished iteration.

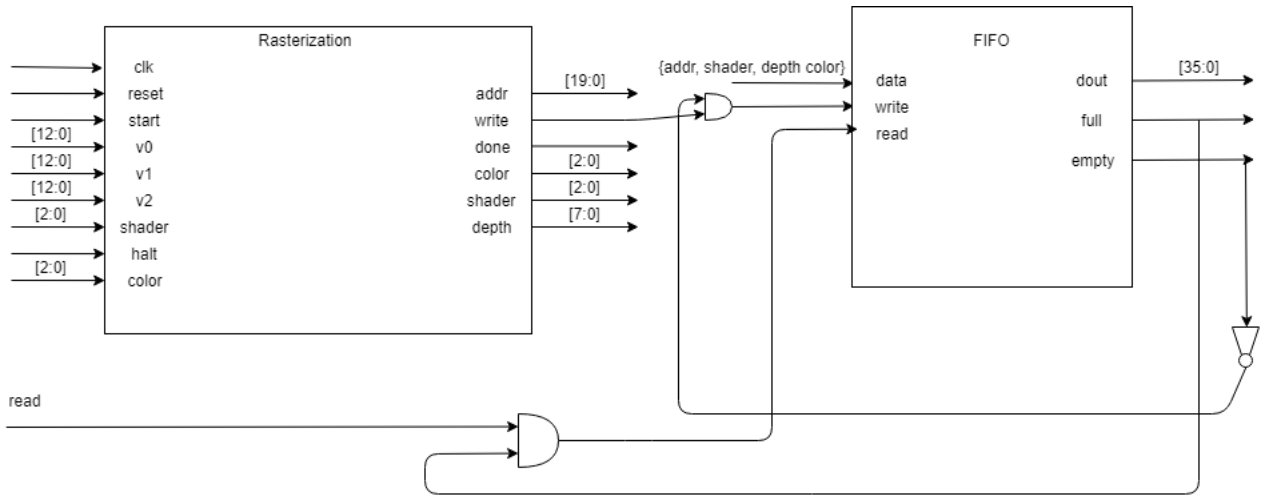


(g) FIFO RASTERIZATION Design

At every clock cycle, the rasterization module gives information about a pixels information (address, color, depth, and shade). Since the ZBT can only read one address at a every clock cycle and there are many rasterization modules, a method of reading many rasterizations modules is needed. Along with that, we only want to read rasterization modules that have pixels who are on.

Considering this, we thought to improve the performance (frame rate) through FIFO structures. Rasterization results would push data into the FIFO only when write is asserted by rasterization. Every Rasterization has a FIFO, so the output of the FIFO is like that of the rasterization, except only values that we care about (filled pixels) are in.

Between the FIFO and Frame-Buffer, is a counter. This counter directs which FIFO the Frame-Buffer should read based on a equality test. This way, the frame-buffer only reads data one at a time.



(h) FIFO RASTERIZATION Implementation

The FIFO modules were implemented using the IP Core Wizard, specified with a 36 bit width write and read, along with a 1024 depth address.

FIFO and Rasterization were combined into one module called fifo_rasterization for abstraction. Thus, the inputs to the fifo_rasterization included the 12 bit integers denoting the coordinates of vertices, 4 bit color input to denote triangle color, 4 bit normal input for shaders, 32 bit output for pixel information, full/empty ports for FIFO information and done output port for external modules to know rasterization is done with processing.

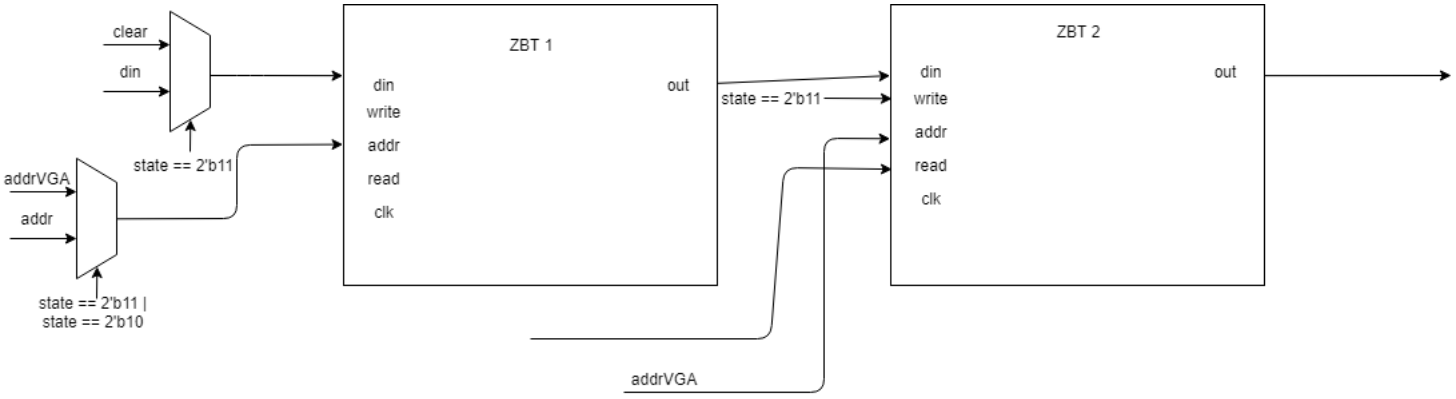
Within this module, this design describe above was implemented. A 36 bit wire was created from the outputs of the rasterization module to connect into the din (data in) port of the FIFO. This din port is a concatenation of the address, shader, depth, and color, in that order.The write port of the FIFO is connected to the output write signal of the rasterization module to ensure only relevant data is written into the FIFO.

Since there is a possibility of the FIFO being full, the FIFO must communicate with the Rasterization module to tell it to halt. Without a halt signal from the FIFO, we can lose some data since data will not be written into the FIFO.

For the frame-buffer to read data, we only read data if the counter corresponds to the FIFOs assigned number and if the FIFO is not empty. Therefore the rd port is a logical and of these two signals.

(i) ZBT-Management Design

We wanted to ensure that every frame had a continuous output of pixels, meaning that every frame had something showing, even if it was laggy. In order to do this, we need to read and write into our frame buffer at the same time since writes to the ZBT are at arbitrary locations and reads are based on the xvga module. However, ZBT cannot read and write at different addresses at the same time. To get around this, we use one ZBT for writing, while the other is used exclusively by the xvga module to display pixel information.



(j) ZBT-Management Implementation

ZBT has an address depth of 512K, with 36 bit write depth, however our screen size requires at least 786432 addresses, the amount of pixels on screen, to store its 14 bits information (depth, shade, color). Since we only use 14 bits for each pixel, we can therefore write information about two pixels into 1 ZBT address. This will only take up 28 bits of information out of the 36, leaving room for expansions like more colors.

The ZBT management system is a state machine with 4 states. Reading, Writing, Migration, and Clearing. When we are given pixel location (ie addr, depth, shade, color), we cannot just write the information directly into the ZBT since two pixels share the same location. Therefore we must read the address, and replace the appropriate pixel.

Since reading a pixel takes 2 clock cycles, we read for one cycle. Then two clock cycles later we write into the machine. In the meantime, we write previous data using registers indicating the past data and read new data. Therefore the state switches from reading to writing to reading etc. We switch back and forth

until ZBT-management gets a frame done signal.

Once frame done is enabled, the ZBT finishes it last write and enters the migrate state. During this state, we read the future address that VGA will read (to account delay in read) and to write the output of the first ZBT into the other ZBT reading at the same address. Once the ZBT has looped through its addresses, the machine enters the clearing state.

At this state, memory is cleared such that the depth of pixels is maximal and color is default.

4 Difficulties and solutions (together)

Various obstacles pop up while we're implementing the whole graphics pipeline, we list the major ones below. Subsection (a) to (c) is written by Chenkai and (d) to (f) is written by Alexis

(a) When we started to implement the matrix module at the beginning, because we're using fisrt person view which is implemented by perspective projection, there is division operations in the matrix module, but the build in division takes too many clock cycles. So we implemented a look up table that takes in a divisor(9 bits) and return a multiplier and a rightshifter which makes an equivalent operation as divide by the divisor. We also need to do *sinusoidal* calculations for our orientation and viewport calculation, which is also hard for hardware to calculate in real time, so we had another loopup table for sin and cos values.

(b) Another big problem is that since we have multiple modules and FIFOs pipelined precisely, we need to make sure the pipeline and different state works under different situations, in perticular when the mat-ras-fifo or the rasterization FIFO in the end is full, we need to halt all the procedure before the rasterization and wait for the pixel data passed into ZBT and free the FIFOs. In order to halt and restart all the procedures without loosing any information and pipeline mismatch is quite a challenging task since we're first time implementing the framework. We came up with multiple debugging approaches including unit test each module using modelsim, output signals through hex LED display, and drawing a small 2D map on the corner of screen for place information. We finally made the pipeline work by debugging each module back and forth. This is probably the most time-consuming part of the whole project.

(c) The other problem that is deterministic for the whole project is the hardware limit. Since we mentioned before that we need to display 6 faces for each triangle, we started planning to have 6 matrix modules, for which we can process each block in one clock cycle, but we don't have enough logic for that and it took too long to compile (more than 10 minutes), which makes it harder to debug. So we decided on only having 2 matrix modules and write additional logic for triangle generator to ouput 6 triangles in 3 cycles.

(d) Rasterization: During the rasterization processes there were edge cases that were tricky to figure out. If a triangle lay partly out of the screen, we had to consider this case while not iterating through the non-existent part of the screen. When rasterization was first made it only ran at 27 mhz, I had to pipeline the module to get it to run at 65mhz. Since most of the heavy computation was done in the beginning, I devoted some more clock cycles for initialization. This seemed to have fixed the problem.

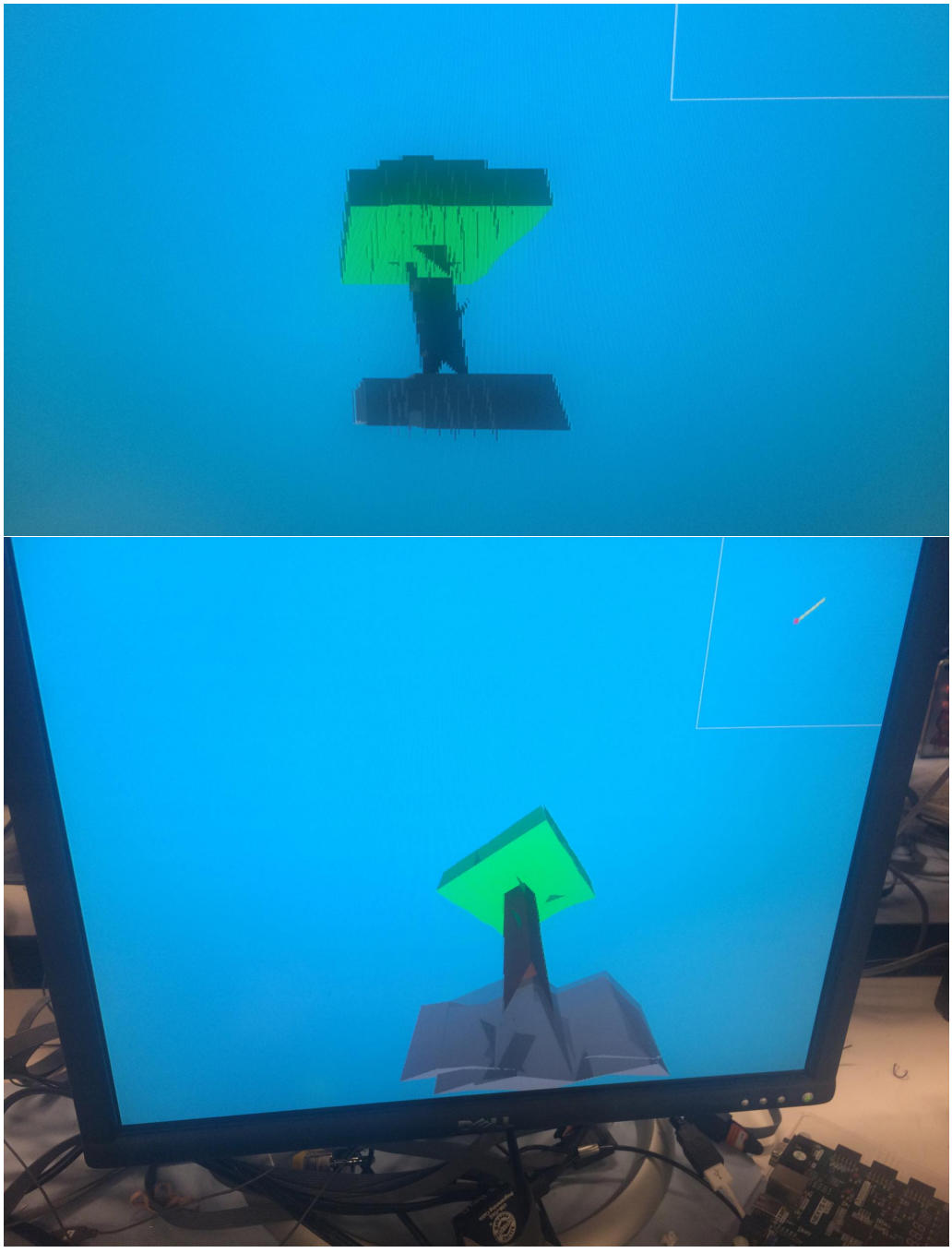
(e) FIFO Rasterization: There was this problem on the screen where the triangles would have appeared to have lines on them. It took a while to figure out. The problem did not lie in the rasterization module or the configuration of the rasterization - fifo design, but rather that I was reading from an empty fifo at times. After figuring this out and making the correct adjustments, the problem disappeared. Lesson: do not read from an empty FIFO.

(f) ZBT Management: So actually, there were three iterations of the design. The first iteration with dual port BRAM. This way we could read and write at different locations. However, since we used additional BRAM for LUT and FIFOs, this led to the project running out of space. So we had to switch to ZBT.
Since we wanted parallelism, we decided to use two ZBT modules and run them at twice the clock speed so simulate four separate ZBTs (to emulate real parallelism). However, we ran into the problem of the 65mhz clock and the 130 mhz clock being offset of one another. This issue could have been fixed, however the bigger problem was the fact that during the writing phase we could not display any pixels during that time. I had originally thought that this flickering would not be noticeable since the clock was so fast, however there was flickering.
As a result, using a ZBT for writing and another one for reading, while migrating based on the address of the VGAs pixel seemed like a better option. After implementing it, the flickering problem was solved.

5 Appendix

(a) Results

We displaying some results we have here:



(b) Verilog

We list our major verilg modules here:

(b).1 labkit.v (part)

*