Interactive Minecraft

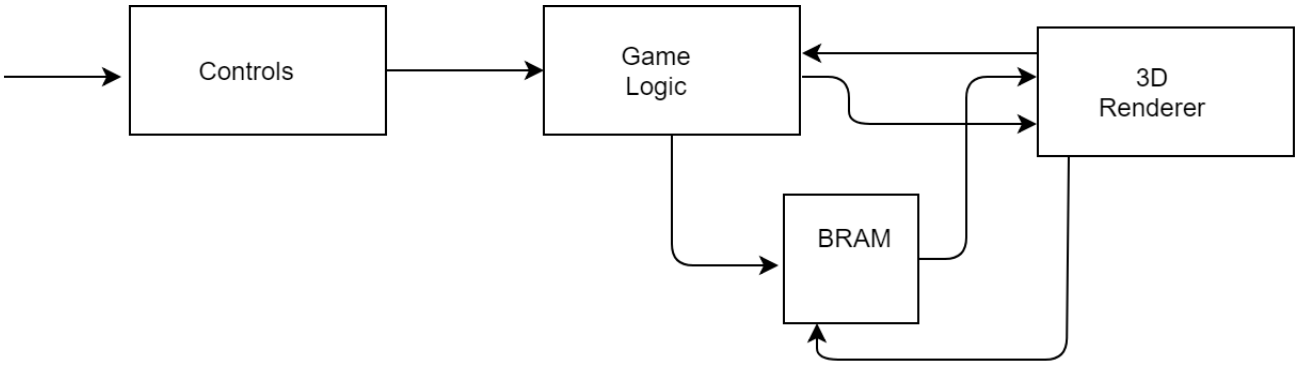
Alexis Camacho, Chenkai Mao

October 2018

1 Overview

For our project we will implement a simplified but interactive version of Minecraft on a FPGA. We aim to build the game in a virtual 3D world made of blocks, where the player navigates interactively using controllers like accelerometers and gyros attached to their body. The players controls consists of sensors in both the hands and legs. Leg sensors act as a position controller, while hand sensors control the players mining tools.

At the core of the project are three main modules that will be used, the controller, game logic, BRAM, and 3D renderer. Each of seperated to think of this project at a higher level.



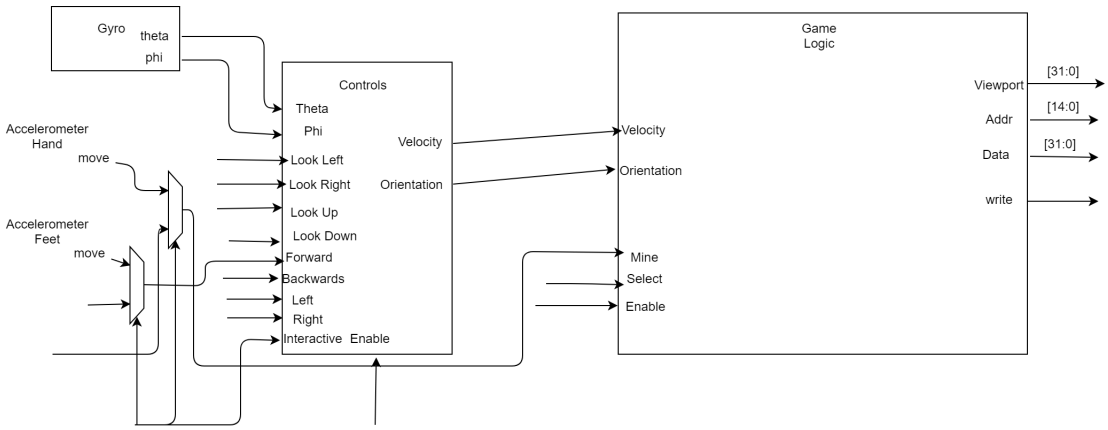
Player Controllers

The player can move in two different ways, either through the buttons on the FPGA or through interactive controllers. The mode of play can be controlled via switch.

Inputs to the controsl include an UP, DOWN, LEFT, and RIGHT inputs for moving, a MINE input to mine blocks, as well as a LOOKUP, LOOKDOWN, LOOKLEFT, and LOOKRIGHT input to control the camera, and gyro scope, accerometer, and mode infomation.

When playing with interactive controllers a gyro on the players head will control the viewport rotation (through phi and theta), while an accelerometer in the hand will detect whether the player wants to mine and an accelerometer in the feet will detect if the player will want to move forward.

These controls will be fed to the game logic to determine position of the player and viewport information. A rough schematic is depicted below.



Representation of the World

This game world will be made completely out of cubes, where the width of the cube will be the main unit of measurement. The world's dimensions will be 32 by 32 by 32 three-dimensional space. This space will be represented through an x,y,z coordinate system where $x, y, z \epsilon [0, 32]$. The player cannot be allowed to go beyond these bounds, meaning additional logic will prevent the player from going out of bounds, this is handled by collision logic within the game logic module.

We will represent a coordinate in the cube world as a byte of information. This means for a given axis, there are 256 possible discrete steps for the player to move in. Since there are 32 blocks in each axis, each block consists of 8 discrete steps (ie fractions). By making each unit have discrete steps the player can appear to move continuously, when in reality the movement is discrete.

Since an axis coordinate can be described by a byte of infomation, a position in space can be described in 3 bytes of infomation.

Each block can have 4 possible colors, brown, green, light green, and gray. These colors represent wood, leafs, grass and stone.

Representation of World State

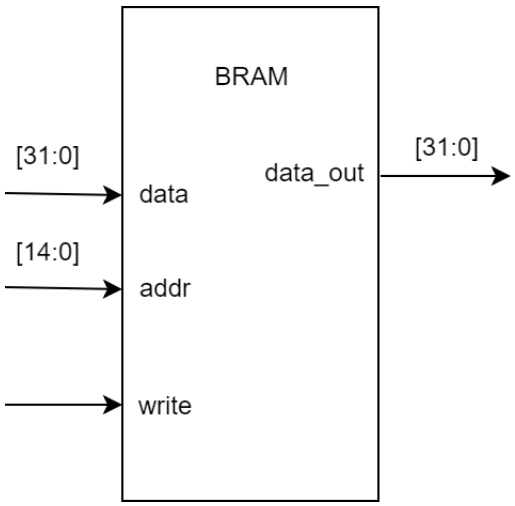
In our game world, there are 32^3 volume spaces where we can insert a block. Each of these spaces can either be filled with a block or not. We can represent these states by using memory with a address space of 32^3, where each entry contains the state infomation about that block.

Each address can contain a three 8 bit numbers indicating the integer position of the cube (where a cube's location is defined by one of its vertex).

Furthermore, each address will contain infomation of whether there is a block or not, so another bit will represent this state. 1 will correspond to a cube is present and a 0 will mean the space is empty.

Along with position and cube state, we also need infomation about color, so another 2 bits are appended to the memory to represent one of 4 colors. The job of the 3D render will then take these 4 bits and give the correct RGB value.

In total each address in the RAM will contain either contain 27 bits. Since BRAM's work best at some 2^n, this corresponds to 32 bits.



Game Logic

The Player is free to move, mine, rotate the camera, and collect blocks. The job of the Game Logic is to handle all of these functions.

Since the player can do all of these things, the player must have states. Therefore we need store the player's position in world space, viewport infomation, and inventory state.

From the controller logic we are given a velocity and an orientation as inputs, these are used to update internal registers as explained below.

The game logic will have an internal register keeping track of the player's position in discrete space, which we will represent using three 1 byte numbers.

Each of these bytes corresponds to the X, Y, Z position of the player.

The view port (ie rotation) can be expressed as two angles, phi and theta. Since we want the player to differ between angles of 1 degree, we will express these angles as two 9 bit numbers, allowing for 512 angles to be represented.

For the player's inventory, we use four 5 bit numbers to represent the number of each type of cube the player has. Since there are 4 types of cubes, we only need to track the amount of each, which at maximum can be 31.

When the player places a cube, the rotation vector picks the first location it collides with where there is a cube, and places the specified player's cube above if the player has enough cubes. An internal counter decrements the inventory.

When the player mines, they mine the first cube the rotation vector collides with. An internal timer keeps track of how long they should press the button before the cube disappears and the player gets the cube. This timer resets when player stops mining.

Velocity of the player will not be state, but rather the position of the player is updated with each clock cycle with a specified direction indicated by the velocity input.

3D Graphics

In this project, we will only focusing on displaying the 3d scene, without the lighting and shading effects. From the modern 3d rendering pipeline, one basic idea behind rendering a 3d scene is to decompose the world to various little triangles(which is extremely easy since we are in a world made of cubes), transform the triangles from the 3d world coordinates to the 2d screen coordinates, and fill in each pixel with color calculated from the 2d triangulars. To explain this in more detail, the high level idea can be described as below:

1. The 3d rendering engine takes input *cube_data* from memory(BRAM), the user control and viewport data from game logic, and some other enable inputs like start, newdata and ALLFEED. It does coordinate transformation rasterization, and z-buffering, then feed the information to the frame-buffer(essentially BRAM), which will feed to VGA module once ready to be rendered.

2. Since we store all the cude information (32 bits for each)in BRAM, which includes the cube locations, colors and state bits, we can easily generate an array of cube data and feed to the 3d Renderer(through some buffering). Vertex to Triangle module implements simple logic to generate 12 triangles for each cube.

3. We then feed the triangle data to the Matrix transformation module(through 3D buffer). The Coordinate transformation module transforms the 3D triangle to 2D screen coordinates by doing a series of matrix multiplication, where the matrices are based on the viewport information (like where the player, i.e. the camera is and where it's facing). It also calculates the depth (distance of the object to the camera) and discards the triangle if it lies outside of the screen.

4. The rasterization(i.e. filling in the pixels inside the triangles) is done by large amount(about several hundred) computation unit which as we call it "R-unit" (rasterization unit). After each triangle is transformed to 2D, we store the location, color, depth information to BRAM(through 2D buffer), writting to an address that is mapped to one of the R-units. On the other hand, once any one of the R-unit is done rasterizing, it send its mapped address in BRAM to the 2D Buffer, so the buffer knows which address to store next value. And the R-unit keep looking into BRAM to read the new data, if it is set by the 2D buffer (there should be a dirty bit indicating the data is newly written).

5. For each of the R-unit, it takes the data (location, color and depth of the triangle) from its associated address in BRAM, and does the logic to figure out where on the screen should be apdated, and what are the color values and depth for each pixel. It loops through the pixels (only the ones inside or near that triangle) and then send the result to some memory management system(think of it as some buffering for processing and comparing the pixel values.) Once it's done looping, it sends the ready signal with its associated BRAM address to the 2D buffer and start reading for next input triangle.

6. The memory management system(which we have some initial thoughs of but are still improving) takes inputs(pixel location, color, depth) from all the R-units and select the minimum depth one, which should be displayed to the screen. It keeps sending data to the VGA module, which will be displayed when all the pixels has been updated(the signal ALLFEED is asserted and all the R-units finished working.)

Design challenges & potential pitfalls

1. We are using BRAM for storing the cube states, reading and writing for the triangles and zbuffering. Since we're not super familiar with BRAM, there might be some issues like when different parts need to read/write at the same time etc.

2. Most computation intensive parts of the projects lies in the matrix transformation and also the R-unit calculations. For Matrix we could precompute some values (like sin, cos of various angles) to speed it up, but we may need to parallelize the procedure. It is the same with R-units, where the way we rasterizing the pixels really matters.

3. We're using a lot buffering, so almost every module need to send ready signal backwards to the module before it. How to time the clock and reduce the latency should be carefully treated.

4. How to implement the design of the memory management system and how to take the minimum depth from multiple values efficiently.

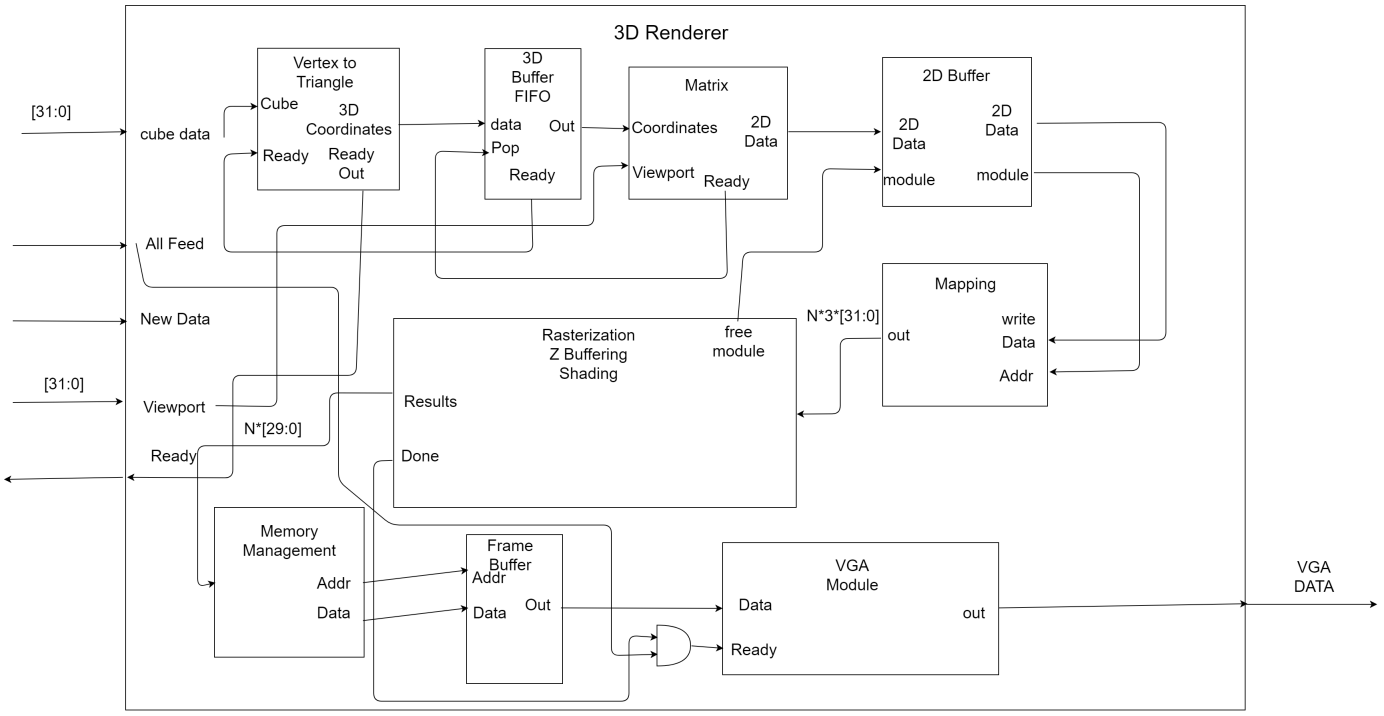


Figure 1: The graphics pipeline

Goals, Milestones and Timeline

There are many uncertain and exploratory things for this project, so we list the following goals and milestones in increasing challenge order:
1. (MVP)Render the 3D world with resonable rendering rate (could start with a small plain ground) and the player navigates in the world using the buttons on FPGA.
2. Incorporate controllers with proper filter.
3. Add interactions like picking and placing objects, add tools and inventories.
4. Add sound, UI and other creatures, opponents etc.

We aim to achieve these milestones throughout the approximate timeline(the tasks are split to 2 people but each member should help and coorporate throughout the whole process):

11/04: Figure out the matrix multiplication and linear interpolation for rasterization and also BRAM related stuff.
11/11: Finish module matrix multiplication and rasterization modules and testing, optimizing.
11/18: Finish buffering and simple game logic (control). Display simple setups like plain world.
11/25: Start incorporating hardware, being able to transfer data. Improve game logic design, e.g. adding interactions like mining and placing objects, adding user inventories.
12/02: Finish controller design, namely add necessary filters to the data stream, incorporate it into the game logic. If have time add some UI.
12/09: Debugging and optimization. If have time, potential things to do: add some sound effects; add shading and texture for the blocks; add tools to use(sword, axe); add other creatures in the world.