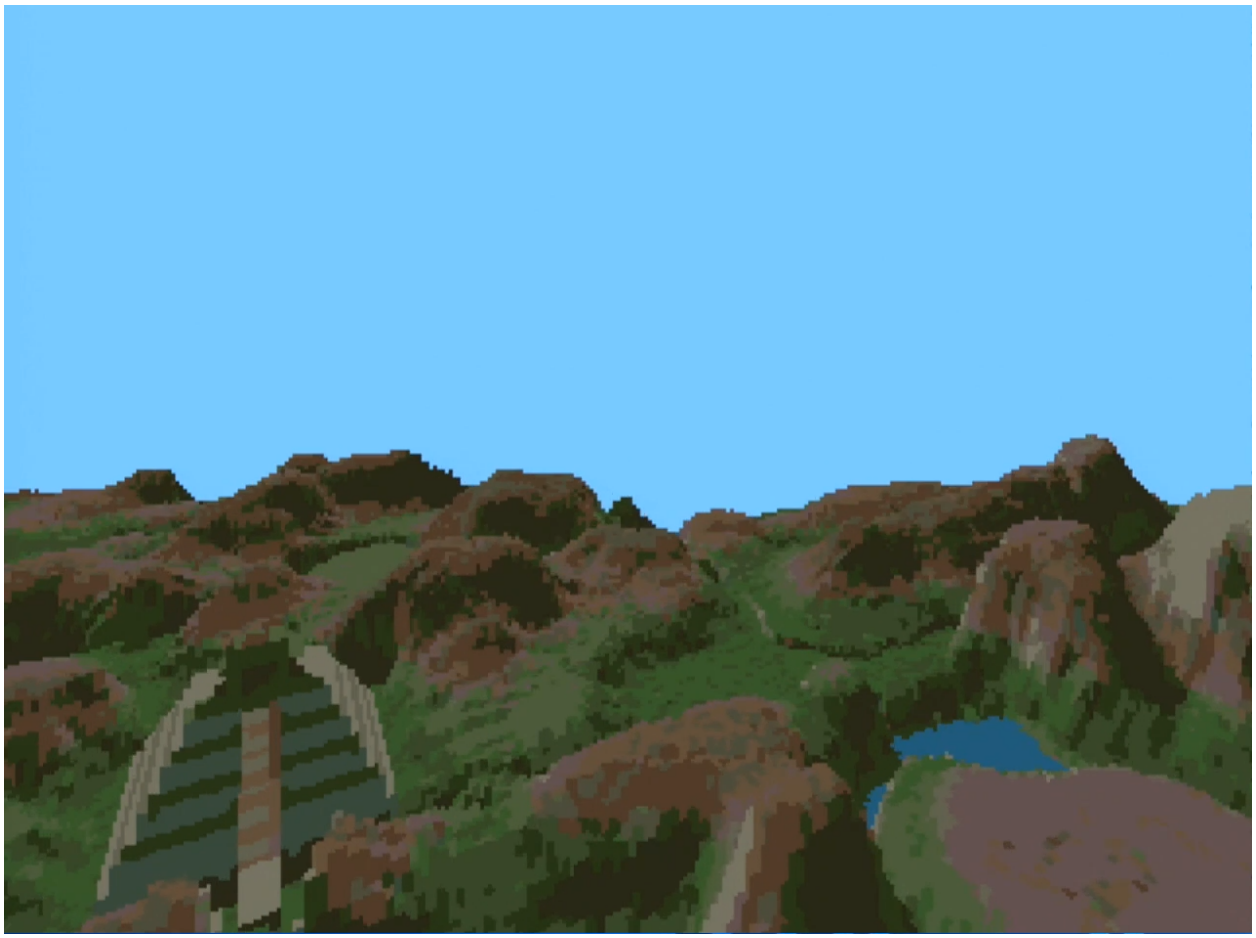
ECE 385

Fall 2021
Final Project



3D Terrain Rendering Engine

Venkat Rao, Patrick Marschoun
Lab Section AB5

Contents

Introduction

We designed and implemented a 3D terrain rendering engine that renders terrain at 60 frames per second at a resolution of 320 by 240 pixels. We are able to view this terrain with full movement and camera control, utilizing a joystick and buttons. The map size is 256 by 256 tiles with each tile allowing for 32 different height values and 15 colors. The user has the choice of walking on the surface of the terrain, or flying above it at a constant height. Our rendering algorithm is based on the "Voxel Space" rendering engine, developed by NovaLogic, an engine designed to render convex terrain quickly. The pre-generated map used in our project was extracted from NovaLogic's Comanche series of games.

System Overview

Our system consists of four main modules. First, the movement module checks the input from the joystick and buttons, and updates the player's position and angle every frame. The rendering module uses the player position and map data as outlined in the "Rendering Algorithm" section and writes a raw image (see image captioned "Raw Framebuffer" in "Rendering Algorithm" section) to a framebuffer. The framebuffer module is responsible for switching between two buffers, the one that the render module is writing to and the one that the output module is reading from. It makes sure the buffers are only swapped when the screen is in a blanking period, and that the buffer is cleared before write access is enabled for the render module. Finally, the output module takes the raw image stored in the framebuffer and processes it into the final image. It then scans this image out, along with the sync signals required by the VGA standard.

Rendering Algorithm

Basic Steps

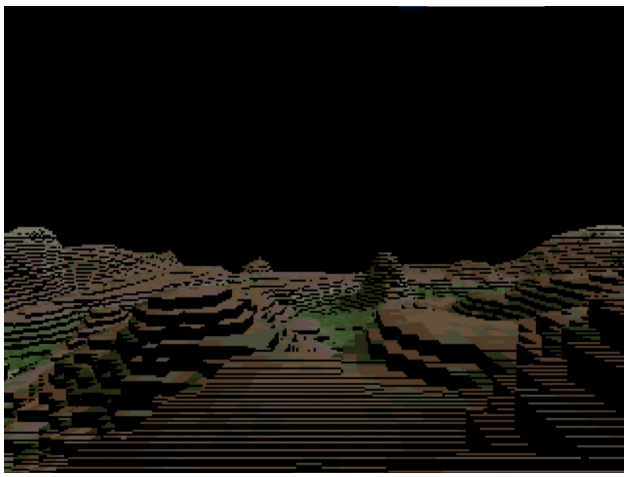
Our algorithm is based on the rendering algorithm employed by the game Comanche by NovaLogic, first released in 1992. The rendering algorithm has since been reverse engineered, and we referenced pseudocode and maps found at https://github.com/s-macke/VoxelSpace.
It follows these basic steps:

1. Starting at the player, draw two rays emanating from the left and right edges of the player's vision. For our code, we chose to make our field of view around 70 degrees, so these rays are about 35 degrees left and right from our player's facing angle.
2. Perform the following loop using different distance intervals from the player. We chose distances ranging from 0.5 to 256, stepping by 0.5 each time. Because our map is 256 tiles wide, this just barely avoids drawing the same object twice. For each distance value:
    a. Get the point on our left and right rays that is that distance from the player.
    b. Now, we have a left point and a right point. Calculate 318 more points at equal steps between these left and right points. This gives us a total of 320 points, all at the same normal distance from the player (they all have the same distance in the direction the player is looking, although not the same Euclidean distance). We now basically have a line of 320 points on our map. For each point:
        i. Fetch the map data at that point. Calculate the difference between the player's eye height and the terrain's height. Multiply this difference by the inverse of the "normal distance" from the player to the point. This serves to perform a perspective correction, where heights on the map are scaled according to their distance. Finally, multiply by a constant factor in order to achieve the desired screen height.
        ii. Draw a vertical line with this color, making sure not to overwrite any previously written data in the framebuffer (so that higher distance terrain doesn't overwrite closer terrain).

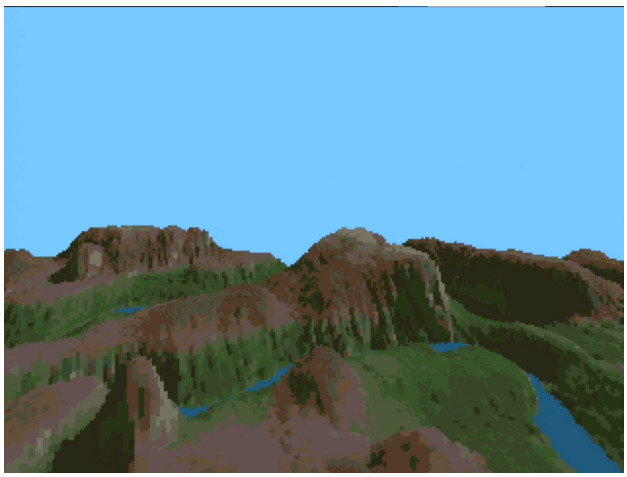
This is a simplified version of what is actually occurring, and some optimization tricks we used are explained in the next section.

Optimizations

1. Our implementation doesn't actually draw vertical lines, as that would require up to 240 cycles per point to draw a vertical line, which is simply too long to achieve a desirable framerate. Instead, we only draw the first pixel of the vertical line, and the output module interprets an unwritten pixel in the framebuffer to mean "the same as the pixel above it", so writing a single value to the framebuffer will draw a vertical line. If we see an unwritten pixel before anything else, we just draw the sky/background color instead, as it is above any terrain drawn in this column. This strategy greatly reduces the load on the rendering module, as it only has to draw a single pixel per point. Below is what the framebuffer looks like if we draw it directly to the screen. In the final output, the black pixels are replaced by sky, or by the last color seen.



Raw Framebuffer



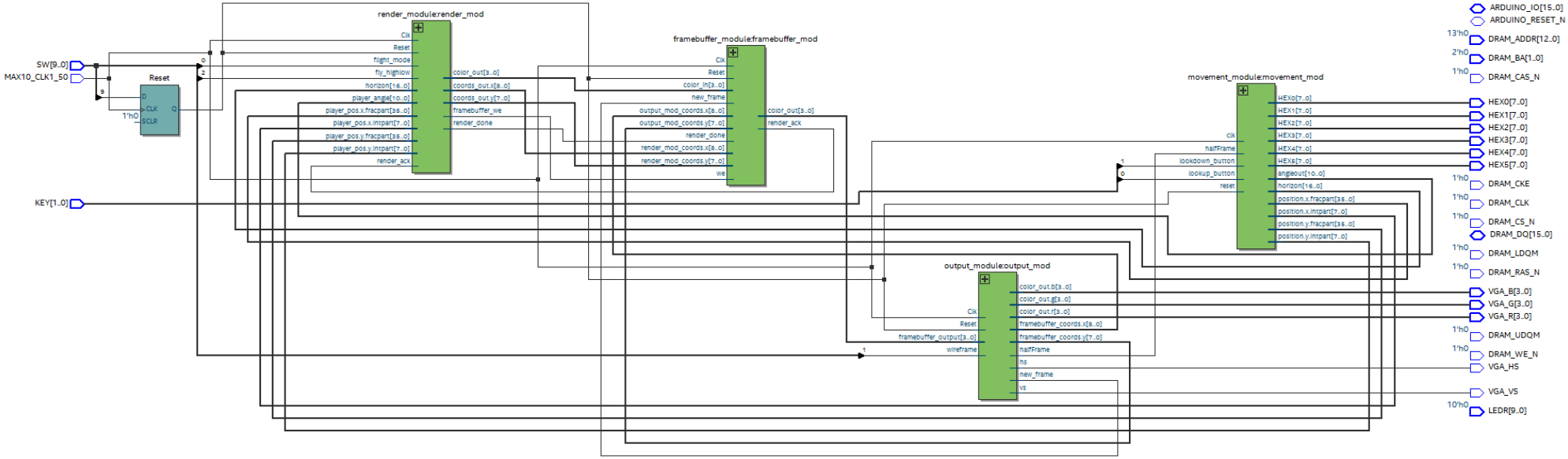
Final Output

2. In addition, because colors are 12 bits, we used a palette of 16 colors instead, meaning our framebuffer uses 4 bits per pixel as opposed to 12. We also assigned palette index 0 to the sky, meaning we could not use it in our map (as then we could not do the trick above with using color 0 as “no change”). So, we ended up with a 15 color map, which we quantized down from the original 256-color map using a Python script. We lose some color fidelity, but the map is still recognizable, and much smaller in size. Examples of an original map and our 15-color version are shown below. We also downscaled height values, from 0-255 down to 0-31.

Original Map (256 colors) Quantized/Compressed Map (15 colors)

3. The system also does not actually calculate the point on the right ray in step 2b, or the 318 points in between. Instead, it calculates what the distance between these evenly spaced points would be, and then produces a vector in the proper direction with that distance. (This is the purpose of the RenderStepLUT: It produces sines that are scaled to what the step should be at our first distance of 0.5, which we then add for subsequent distances). Then, during the loop, we start at the left point, and after each iteration, add the vector to our point. Thus, we will still visit each of the 320 evenly spaced points. This saves us from having to do division, which is very expensive on an FPGA.

Block Diagram



Module Descriptions

Note: fp44, posXY, RGBcolor, screenXY, angle, mapheight, palcolor, and maptile are structures defined in the *structs.sv* module, described below.

Top-Level Module: finalproj.sv

Input: MAX10_CLK1_50, [1:0] KEY, [9:0] SW
Output: [9:0] LEDR, [7:0] HEX0, [7:0] HEX1, [7:0] HEX2, [7:0] HEX3, [7:0] HEX4, [7:0] HEX5, VGA_HS, VGA_VA, [3:0] VGA_R, [3:0] VGA_G, [3:0] VGA_B
Inout: [15:0] ARDUINO_IO, ARDUNIO_RESET_N

Description: This module instantiates the main submodules - *framebuffer_module, render_module, output_module,* and *movement_module*. It also contains a synchronizer for the reset switch and internal wires which connect the different submodules together. The KEY[1:0] inputs are connected to the *movement_module* allowing the character to look up and down with the DE10-Lite's buttons. The player position and angle are outputs from the *movement_module* and are connected as inputs to the *render_module*. Coordinates from the *output_module and render_module,* labeled as *output_module_coords* and *render_module_coords* respectively, are connected to as inputs on the *framebuffer_module.* The *frame_buffer_coords* and *framebuffer_out* output values are connected to the output module. The *output_module* outputs the five VGA signals, which connect to the VGA pins on the board. Further detail about how each of the modules connect can be found in their respective descriptions. Other than the reset synchronizer, there is no logic in the module.

Purpose: This is the top level module for the design. It is a bare-bones module with the express purpose of connecting the other submodules and integrating them into a completed design. It also connects the physical inputs and outputs of the DE10 Lite board to the submodules.

Module: structs.sv

Inputs/Outputs: N/A

Description: This module defines the following structs - *fp44, posXY, RGBcolor, screenXY, angle, mapheight, palcolor, maptile. fp44* is our fixed point representation, containing 44 bits with the top 8 bits assigned to the integer portion and the lower 36 bits assigned to the fractional part of the number. *posXY* represents a set of X and Y coordinates, and is also used to represent vectors. It is made up of two *fp44* numbers. As each side of the square map contains 256 map tiles, each X and Y coordinate can be represented by a *fp44* number. *RBGcolor* is a 12 bit value corresponding to an RGB color. *screenXY* represents a position on screen. As the design uses a 320 by 240 resolution, the X coordinate is represented with a 9 bit number and the Y coordinate is represented by an 8 bit number. *angle* is an 11 bit number with 0 representing 0 radians, and 1024 representing π radians. Thus, the maximum value of 2047 wraps back around seamlessly to $2048 = 2\pi = 0$. *mapheight* is a 5 bit struct that can store 32 different height values. *palcolor* stores one of 16 possible palette indices. *maptile* is the struct stored for each map tile that contains a *mapheight* and *palcolor*.

Purpose: The purpose of this module is to make the code more human readable by assigning recognizable type names to the commonly used types. Using packed structs also means that we can easily perform operations such as getting the integer part of a fixed point number, and if we update our structs we do not have to rewrite much HDL.

Major Submodules

Module: movement_module.sv

Input: clk, reset, halfFrame, lookup_button, lookdown_button
Output: angle angleout, posXY position, [7:0] HEX0, [7:0] HEX1, [7:0] HEX2, [7:0] HEX3, [7:0] HEX4, [7:0] HEX5, [16:0] horizon

Description: The main component instantiated by this module is the analog to digital converter module called *adc_qsys*. It takes in analog inputs from the joystick and converts them into a digital 12 bit value. As the joystick has two directions of movement, forwards/backwards and left/right, there is a channel input that is set to either 1 or 2 depending on which joystick direction is being polled. To interact with the ADC module, the *movement_module* implements a basic FSM, shown below. The first state for this FSM is the HALT state. In this state, the input channel, *command_channel*, is set to 1 to read in the forward/backwards direction. When halfFrame input goes high, signaling the start of a new frame, the FSM moves to the YMOVE state. Here, assuming the value from the joystick is higher than a certain threshold, the system calculates the change in the Y coordinate for the player depending on its current angle. It utilizes the trig LUT to perform this calculation. As the trig LUT is a sine function and calculating the change in X coordinate, required in the next state, uses a cosine function, the input address to the trig LUT has 512 added to it. The 512 represents pi over two in our design as our angle has a total of 2048 possible steps. In the next state, XMOVE, the value from the trig LUT has been returned so that change in X coordinate can be used to change the player's position. The FSM then enters the ANGLEWAIT state where the *command_channel* value is set to 2, so the ADC starts to poll the left/right direction on the joystick. As the channel switching takes multiple cycles, the FSM won't move to the next state until halfway through the frame sync cycle. This way we wait the maximum amount of time to allow the ADC value to change to the correct channel. Once the FSM moves to the ANGLEMOVE state, the player angle changes depending on the value on the joystick. It is either increased to turn left, decreased to turn right, or not changed if the joystick value is below a certain threshold. The FSM then enters the HALT state again until the halfFrame input goes back high. There are also two button inputs which increase or decrease the horizon variable allowing for the player to look up and down. The movement module also outputs the current position and angle onto the DE-10 Lite's 7-segment displays for debugging purposes.

Purpose: The purpose of this module is to control all the inputs from the DE10 Lite board and the I/O shield and use them to calculate the player's position and orientation. These values are then connected to the other submodules to perform the screen output and rendering calculations.

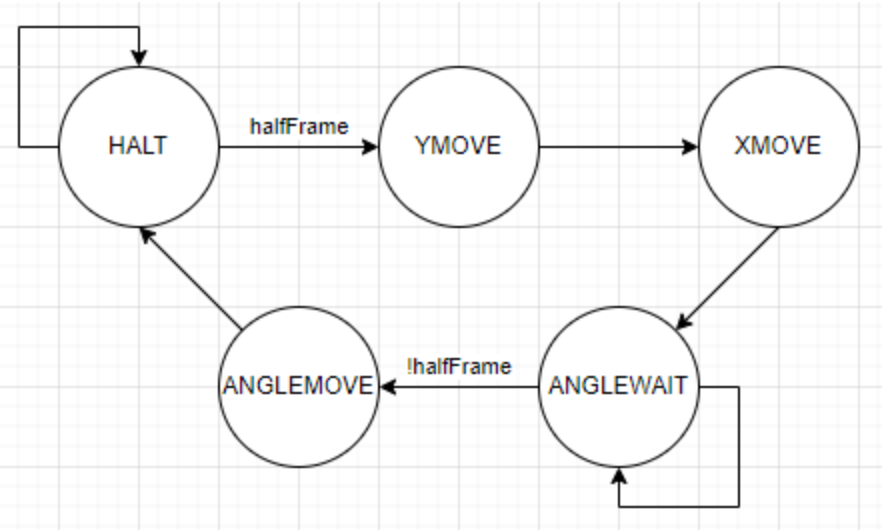


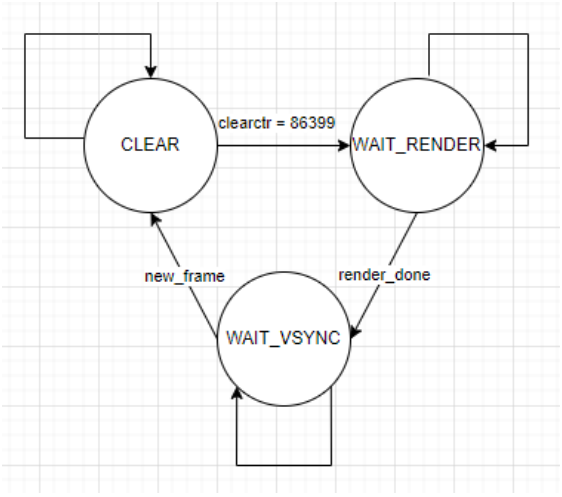
Diagram of Movement Module FSM

Module: framebuffer_module.sv

Input: Clk, Reset, new_frame, screenXY output_mod_coords, screenXY render_mod_coords, palcolor color_in, we, render_done, render_ack
Output: palcolor color_out, render_ack

Description: This module implements a double framebuffer. It stores palette indices (“palcolor” structs) to be read to the *output_module* for display onto the screen. The double buffer is essential for our design; as one buffer is read by the *output_module* and has its contents displayed to the screen, the other buffer is cleared and then written to by the *render_module*. The reason why the clearing step is necessary is explained in the *output_module* description. This module implements a FSM with a CLEAR state clearing one of the buffers. When it is done clearing, the framebuffer sends an acknowledge signal using *render_ack*, indicating that the *render_module* can begin rendering again. Then, it enters a WAIT_RENDER state which writes to the cleared buffer according to the coordinates and value from the *render_mod_coords* and *color_in* inputs respectively. When the *render_done* input is high, the FSM waits for the next frame to switch the buffers. It enters a WAIT_VSYNC state which switches the buffers between read and write mode during the next vertical blank period, when *new_frame* goes high. While one of the buffers is getting written to, the other buffer is getting read from according to the coordinates from the *output_mod_coords* with the output value set as *color_out*.

Purpose: This module provides a double buffer to prevent screen tearing and allow for both the *output_module* and *render_module* to each have their own dedicated buffer. This ensures that the same buffer is never being read and written at the same time. This module stores the palette indices representing the colors to be displayed on screen. It is almost a pixel by pixel representation of what will be displayed, except for the last-seen color step used in the *output_module*.



Framebuffer Module FSM Diagram

Module: output_module.sv

Input: Clk, Reset, palcolor framebuffer_output, wireframe
Output: hs, vs, new_frame, screenXY framebuffer_coords, RGBcolor color_out, halfFrame

Description: This is the module that calculates the actual VGA_HS, VGA_VS, and VGA color outputs sent out to the monitor. It starts with instantiating the *vga_controller* module, originally from Labs 6 and 7 of ECE 385. This module creates the two coordinate outputs that keep track of which pixel is currently being displayed to the screen. The output value *framebuffer_coords* is defined using the X and Y coordinates of the *vga_controller*, divided by 2 to decrease our resolution. Using the *framebuffer_coords* as input, the *framebuffer_module* returns the palette color of the specific output pixel. However, the *output_module* doesn't just output the raw framebuffer. Instead, a last seen value is stored for each column on the screen. If the framebuffer value is zero, indicating "no change", the last seen value for the column being written to is used as the output to the screen. If the framebuffer value is non-zero, that value is displayed for that pixel and the last seen value for the column is also updated. In this way, we draw vertical lines based on the framebuffer instead of single pixels. This is a crucial step of the rendering algorithm, and without it we would have to spend much more time rendering in order to actually draw each vertical line. The rendering algorithm is explained further in the Rendering Algorithm section of this report. As with all VGA signals, the *output_module* does not write any color if the screen is currently writing to pixels off the screen. The vertical sync (*vs*) and horizontal sync (*hs*) signals are passed through from the *vga_controller*, with a small delay to account for the time spent calculating our current color. It also produces the *new_frame* and *half_frame* signals. *new_frame* pulses high for one cycle at the start of each frame, and *half_frame* is a signal that is high for half of each frame and low for the other half. *new_frame* is used by several modules for synchronization, and *half_frame* is used by the movement module for timing.

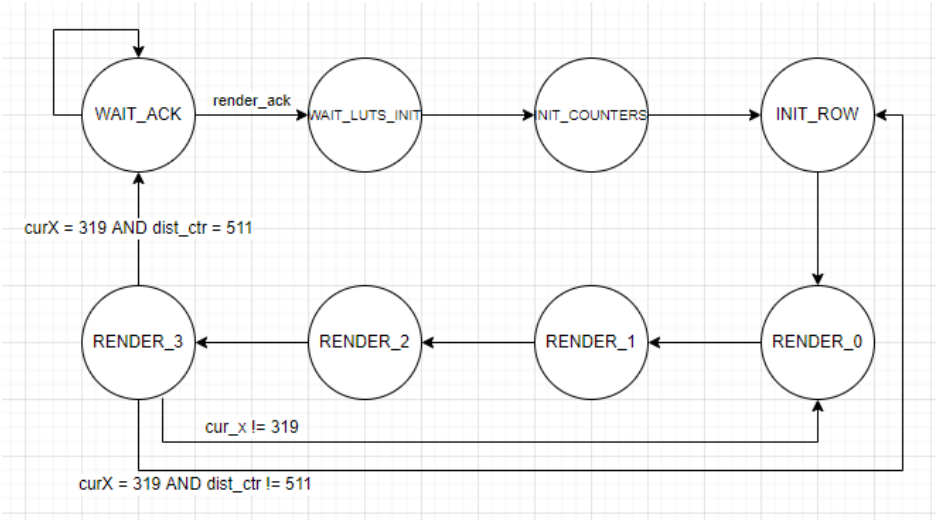
Purpose: The purpose of this module is to produce the five VGA signals - VGA_HS, VGA_VS, VGA_R, VGA_G, and VGA_B - which are outputs in the top level *finalproj.sv* module, sent as a VGA signal to the monitor.

Module: render_module.sv

Input: Clk, Reset, render_ack, posXY player_pos, angle player_angle, flight_mode, [16:0] horizon, fly_highlow
Output: screenXY coords_out, palcolor color_out, framebuffer_we, render_done

Description: This module uses an FSM to render terrain. It starts in the WAIT_ACK state, where it waits for the framebuffer to be cleared and for the framebuffer to send an acknowledge signal. After this, it initializes internal variables using the player's current position and angle provided by the *movement_module* using the *player_pos* and *player_angle* inputs. It also determines the player's eye height, based on either the map height (if the *flight_mode* is 0), or using a constant flying height based on *fly_highlow* if the *flight_mode* is 1 (enabled). Then, in the INIT_ROW state, more variable initialization occurs. Finally, in states RENDER_0 through RENDER_3, a single pixel is rendered. Our render module looks at each tile within a certain distance and within the player's field of view. Then, it determines how high or low relative to the player the tile is. Finally, it scales this difference by the inverse of the distance (found using our *HeightScaleRom*) from the tile to the player, and draws this value on screen, offset by the *horizon* input's value. In this way, it essentially draws the top of each column of the map, and the *horizon* input controls how high or low on the screen the output is. The output module draws the rest of the column by remembering the last seen color for each color and using it if no color has been drawn to the current pixel. After finishing 320 pixels for each row, it goes back to the INIT_ROW state if there are more rows to draw; otherwise, it sends a *render_done* pulse to the framebuffer and waits for the input buffer to be cleared by the framebuffer, moving to the *WAIT_ACK* state.

Purpose: This module renders the current terrain to the framebuffer based on the player's position and angle.



Render Module FSM Diagram

LUT Modules

Module: TrigLUT.sv

Input: clk, angle inval
Output: fp44 outval

Description: This module uses the ROM megafunction as a LUT to store one quarter of a sine wave. The input to the LUT is 11 bits so there are 2048 possible angles and 512 of them are stored in the ROM. The remaining angles are calculated by flipping the input if the required angle exists in the second or fourth quadrant and by inverting the output if the angle exists in the third or fourth quadrant. This way, it is possible to store an entire sine lookup table in a quarter of the space.

Purpose: The purpose of this module is to speed up trigonometric calculations as having the FPGA calculate sine values is expensive and would slow down the final design significantly. Given an angle, it outputs the sine of that angle in fixed-point format.

Module: MapLUT.sv

Input: clk, [7:0] x, [7:0] y
Output: maptile data

Description: This module stores the map data. It takes 8 bit inputs for the X and Y coordinates, converts them to a ROM address, and returns the maptile struct stored at that address. The only logic in this module is the concatenation of the two X and Y addresses into a single address that is used to address the ROM.

Purpose: This module stores the map data and is used extensively by the rendering module while the data is converted into a 3D rendered image.

Module: RenderStepLUT.sv

Input: clk, angle inval
Output: fp44 outval

Description: This module is similar to the TrigLUT, but the sine values are scaled by a constant factor. These are used to decide how far to step through the map. At each distance from the player, we start at the left edge of the screen (so, forward and to the left of the player) and step towards the right edge of the screen (exactly perpendicular/to the right of the player). The values in RenderStepLUT are scaled by a constant factor so that we can simply use its outputs to form a vector that tells us how far we need to step before drawing the next pixel. We could also use a TrigLUT combined with a multiplier, but that uses more resources and is much slower than looking it up in a separate table. (See “Optimizations” section, #3)

Purpose: This module stores scaled sine values that are convenient for part of the rendering code, similarly to TrigLUT.

Module: HeightScaleRom.v

Input: clock, [8:0] address
Output: [26:0] q

Description: This module takes as input the distance from the player to the current point being rendered. The output of this ROM is the inverse of this distance multiplied by a constant scaling factor, to help get the correct y position on the screen to draw to.

Purpose: The purpose of this LUT is to increase the efficiency of the design as calculating inverses and multiplication is expensive on an FPGA. By calculating the value beforehand, we save the FPGA from having to perform the expensive division.

Design Resources and Statistics

Usage of important FPGA resources is listed in the following table:

Resource	Used	Available	% Used
LUTs	1,824	49,760	4
DSP Elements	4	288	1
Block RAM / Memory	1,427,712	1,677,312	85
Flip-Flops / Registers	690	49,760	1
Maximum Frequency	52.8 MHz	N/A	N/A
Static Power	96.56 mW	N/A	N/A
Dynamic Power	79.26 mW	N/A	N/A
Total Power	187.93 mW	N/A	N/A

Due to the basic rendering algorithm used, we did not end up using many FPGA resources. Most of our modules are state machines that only perform a few operations, requiring few LUTs and registers. However, we did use many lookup tables for things like trig functions, map data, and finding the inverse of numbers, giving us high Block RAM usage. The maximum operating frequency of our design is 52.8 MHz, but could probably be increased with better pipelining of the multiplier and some other time-consuming elements. However, as the frequency of our design is tied to the VGA clock frequency, we did not see a need to get the maximum frequency much higher than double the VGA pixel clock. We used 4/288 of the available 9-bit multipliers. The calculation of the height to draw on screen uses these multipliers. It requires a multiplication of an 8-bit signed number, the difference in height between the player and the surface being rendered, and a 36-bit unsigned number, the inverse of the horizontal distance to the surface. This uses four 9-bit multipliers internally.

Development Paradigm/Debugging

While working on previous lab experiments, we realized that visual output is an invaluable debugging tool, and that once visual output was working, it was easy to create different test patterns that we could identify problems with. Simulation is much harder to perform, especially for complex outputs like VGA, and so we decided to do our debugging visually. We did this by first creating the output module, and ensuring that we were able to display simple striped patterns to our monitor. Then, we tested each of the lookup tables by rendering them to the monitor (for example, drawing a sine wave using the TrigLUT). We were able to identify an error with our lookup table having longer hold times than were necessary. Then, we built the framebuffer module, and a dummy rendering module that would simply draw different patterns. Because we were able to see all of the different patterns produced by the rendering module, we knew that the framebuffer was successfully switching and clearing the buffers at the right time. We also were able to debug the rendering module using itself by drawing information, like the map data being read, to the screen. This made the rendering module much easier to debug, as we could see whether different variables were being set correctly by drawing them to the screen. We also used 7-segment displays to view the player's current position and angle, which helped us debug the movement module.

Conclusion

Overall, the entire final project was finished successfully. We were able to accomplish all of our stated minimum objectives. The screen resolution of our project is 320 by 240 pixels, which matches the resolution we stated in our proposal. We are able to control a character that can move forwards and backwards and turn left and right. We also quadrupled our target frame rate of 15 FPS and achieved the maximum frame rate of our monitor, 60 FPS. This made our movement on screen seem extremely smooth and responsive. We were also able to implement another one of our additional features, allowing the user to move the camera up and down. Finally, we added a flying mode to allow the user to observe the terrain from a distance.

Achieving more additional features using the DE10-Lite would have required extensive redesigning of our program. For example, increasing the output resolution to the full 640 by 480 pixels would have increased the size of the frame buffers by a factor of four. This would not be feasible with the size of the on-chip memory on the FPGA, so the SDRAM would have had to be used. As this memory takes longer than 1 clock cycle to return a value, almost all of the state machines would need to be altered and give enough time for the memory to return a value. With the extra delay, the FPS of our design would have also decreased drastically, as rendering each frame would take much longer than 16 ms.

The entire final project process was run well and there was no confusion on what was required to get full credit. The point breakdown for each section was explained in lecture and feedback for the proposal and mid-project checkpoint was helpful.