Team Light Speed
Adam Carpentieri **AC4409**
Souryadeep Sen **SS6400**

FPGA Raycasting Spring 2022

Final Report

1 OVERVIEW

FPGA Raycasting is a project to implement raycasting techniques originally developed in the mid 1990's with games such as Wolfenstein, in FPGA hardware. Pioneered by John Carmack, who took cutting edge research papers and turned the idea into a working game in the span of several weeks. The technique allowed a full 3d game to run on low end PCs of the time.

Our ultimate goal was to replicate the visual style of https://js-dos.com/Wolf/.

The result of our efforts has seen us progress from a low resolution software renderer to a high(er) resolution hardware renderer running in smooth 60 frames per second with full visual fidelity.

Our project benefits from substantial memory savings by avoiding writing to the frame buffer, a speed up since all display operations are done in hardware, and the perfect timing of the hardware which is not prone to scheduling or interrupts. This all leads to a very smooth experience for the user.

Our Github repo is https://github.com/4840-Raycasting-Project/raycasting-prj/.

2 RAYCASTING ALGORITHM

Raycasting is a technique to transform a limited form of data such as a simplified 2D floor plan into a 3D projection by tracing rays from a viewpoint (in our case, the player), to the viewing volume (the VGA screen). Ray casting determines the visibility of surfaces by tracing imaginary rays of light from the viewer's eye to the object in the scene (which will be the textured walls in our implementation). Ray casting sounds much like ray tracing, but must be noted, that it is a special case implementation of ray tracing, due to geometric constraints, which makes the algorithm much faster and simpler compared to ray tracing, but at the same time, images appear blocky and less accurate.

The geometric constraints mentioned above are

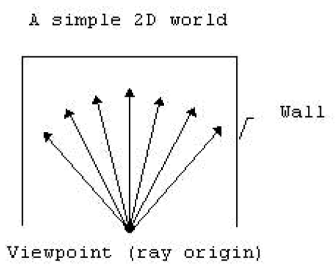
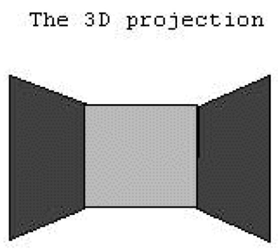
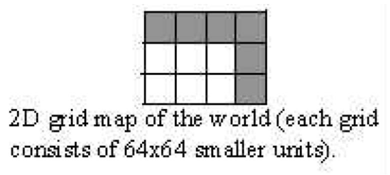
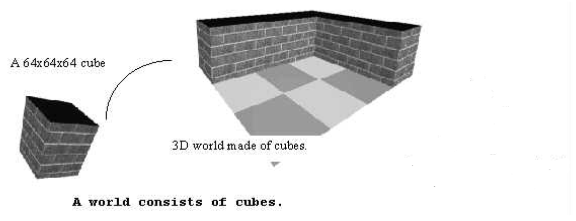
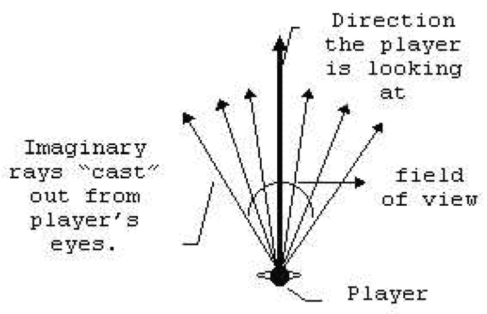
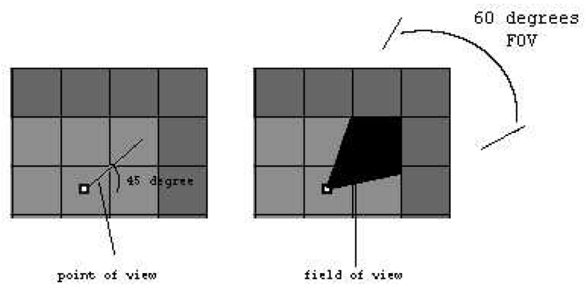
1. Walls are at 90 degree angle with the floor
2. Walls are made of cubes that have the same size
3. Floor is always flat

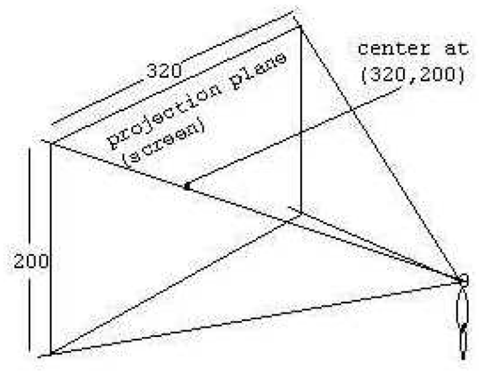
The entire algorithm is based on basic trigonometry, that computes distances to intersections on the grid, distance to next intersections on the grid, height of walls, distance to walls. As a result, we pre-compute these math operations for tan, cos, sine and their inverses in static arrays that are indexed based on player position, field of view and viewing angles.

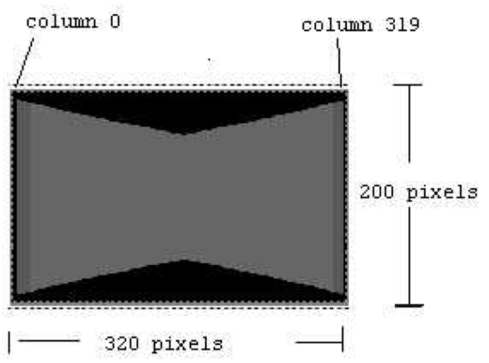
Some projection attributes defined:

1. The map layout is made up of 64x64 pixel grids
2. Players height is 32 pixels tall
3. Wall height is 64 pixels tall
4. FOV (Field of View) is 60 degrees
5. The walls are made of 64x64x64 cubes

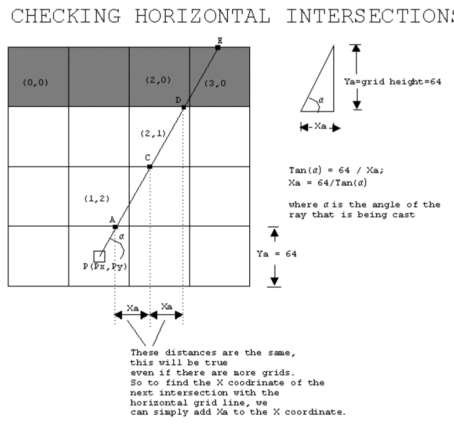
Below are some images of what the ray casting algorithm translates to (these images are taken from https://permadi.com/1996/05/ray-casting-tutorial-table-of-contents/ :

A simple 2D world: 	**The 3D projection:** 
2D grid:  2D grid map of the world (each grid consists of 64x64 smaller units).	**Corresponding 3D world:**  A 64x64x64 cube 3D world made of cubes. A world consists of cubes.
Field of View:  Direction the player is looking at Imaginary rays "cast" out from player's eyes. field of view Player	**Field and point of view on the grid:**  60 degrees FOV 45 degree point of view field of view
The projection plane:	**The ray cast on the screen (projection plane)**





Horizontal intersection with walls on grid:

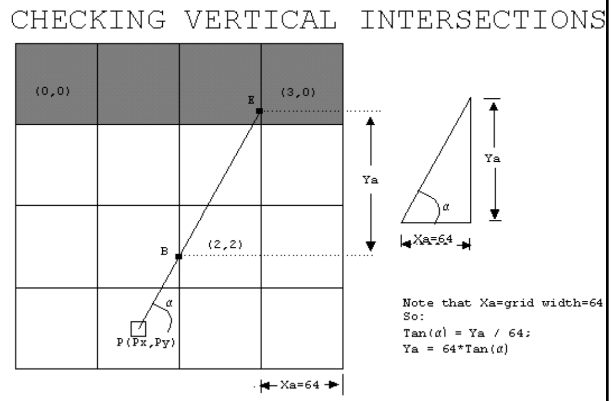

CHECKING HORIZONTAL INTERSECTIONS

Tan(α) = 64 / Xa;
Xa = 64/Tan(α)

where α is the angle of the ray that is being cast

These distances are the same, this will be true even if there are more grids. So to find the X coodrinate of the next intersection with the horizontal grid line, we can simply add Xa to the X coordinate.

Vertical intersection with walls on grid:


CHECKING VERTICAL INTERSECTIONS

Note that Xa=grid width=64
So:
Tan(α) = Ya / 64;
Ya = 64*Tan(α)

The horizontal intersection math:
(based on grid points above assuming alpha is 60 degrees and 64x64 grid size)

Ray facing up:

- A.y = rounded_down(Py/64) * (64) - 1;

Ray facing down:

- A.y = rounded_down(Py/64) * (64) + 64;

X intersection point:

- A.x = Px + (Py-A.y)/tan(ALPHA);

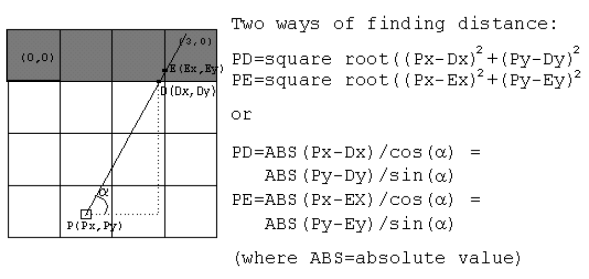
If you were observing closely, the next x intersection would be at 64/tan(alpha), and the next y intersection would be at Ya-64 (facing up) and Ya+64(facing down). So we can conveniently add the values to the current intersection, till we hit a grid that has a wall.

The vertical intersection math:
(based on grid points above assuming alpha is 60 degrees and 64x64 grid size)
The math is the same as for the horizontal intersection, except now the role of X and Y gets swapped in finding the intersection points. Hope you see it :)

Distance to the wall:

Height of the wall:

$$\frac{\text{Projected wall height}}{\text{distance of player to the projection plane}} = \frac{\text{actual wall height}}{\text{distance to the wall}}$$



Two ways of finding distance:

$PD=\text{square root}((Px-Dx)^2+(Py-Dy)^2$

$PE=\text{square root}((Px-Ex)^2+(Py-Ey)^2$

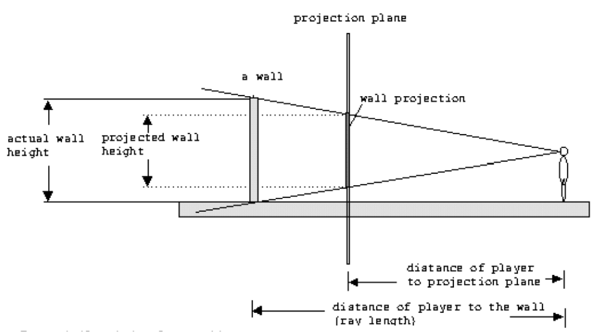
or

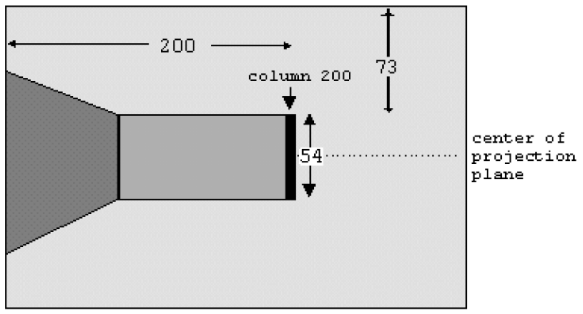
$PD=ABS(Px-Dx)/\cos(\alpha) = ABS(Py-Dy)/\sin(\alpha)$

$PE=ABS(Px-EX)/\cos(\alpha) = ABS(Py-Ey)/\sin(\alpha)$

(where ABS=absolute value)

Example of wall appearing smaller the further it is from the player >>>>>>>>>>


projection plane
a wall
wall projection
actual wall height
projected wall height
distance of player to projection plane
distance of player to the wall (ray length)



This highlights the algorithm. There are additional manipulations to prevent fish bowl effects, adding textures to the walls, drawing floors and the sky, moving forward, backward, but the rendering algorithm continues to use the above computations.

3 SYSTEM DESIGN AND COMPONENTS

3.1 SOFTWARE

The software runs a game loop, and once it renders the scene (creates the column data from the geometry and state of the game) it waits until the display signal is in between frames (V_BLANK). It uses a polling routine combined with a very short sleep interval to constantly check the status of the frame.

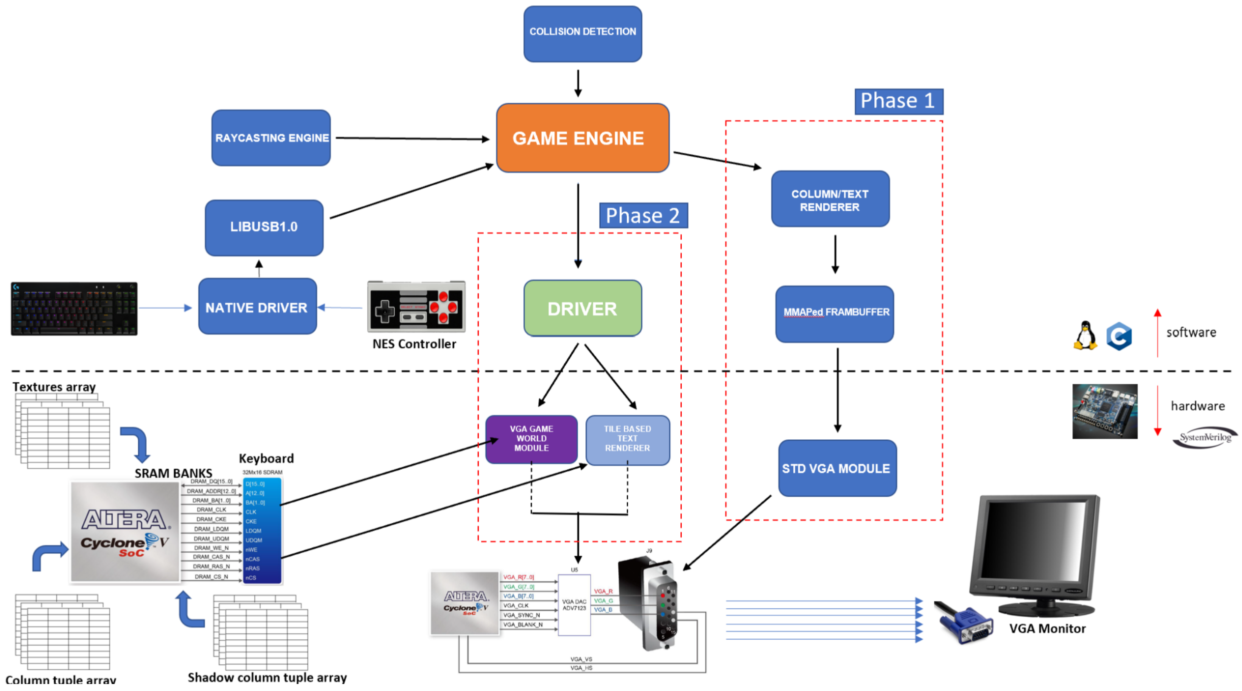
Once the frame is completed, the column data is sent. Because of our triple buffering scheme, this is not necessarily needed, though it does ensure that no frames are missed. The **driver** (kernel module) interacts with the Avalon bus to transmit the data from software to hardware.

Separate threads handle usb input from the keyboard and controller to be read into the game loop.

3.2 HARDWARE

The hardware is broken down into the main module, the vga module, the texture module, and the column modules. The hardware, which we refer to as the **column decoder**, gets written to through the Avalon bus by way of the software driver.

It also accepts commands for text display and blacking out the screen (not showing the rendered scene).



4 HARDWARE

4.1 COLUMN DECODER & RENDERER

4.1.1 Design Assumptions

- 64 unit wall height
- 64 unit “block size”
- 1 pixel column width

4.1.2 Column Data

For each pixel, we grab the column tuple in SRAM corresponding to the row. Then we calculate if the pixel is in the ceiling, wall, or floor. Floors and ceiling are hard coded with some RGB value and also incorporate a subtle gradient for a pleasing effect.

If the pixel is contained in a wall, we will need to calculate the “relative” or perspective adjusted row of the wall we are so that we can grab the appropriate RGB values from the texture, also in SRAM.

The column tuple is encoded as 74 bits:

- **Top of wall** [16 bits signed]: what row the ceiling ends and wall starts. Can be a negative number if the top is higher than the top of the screen (row 0)
- **Projection wall height** [16]: how many rows of pixels for the wall starting with the above value. Can be larger than the amount of rows on screen if the wall is zoomed in.
- **Wall side** [1]: a single bit to say if the wall is along the x or y axis on the map. Using this data you can darken or lighten the wall to create a subtle lighting effect
- **Texture type** [3]: select one of 8 different textures
- **Texture offset** [6]: 0-63 in terms of what column of the block you are currently working with. This is critical for mapping pixels to textures.
- **Scaling factor** [32] Special bit shifted number to indicate the relative pixel position with respect to the 64 pixel texture. We simulate floating point by creating a large number, bit shifted left, by 21 positions. In the hardware, we perform the multiplication, then right shift the 21 positions to correct.[1]

Because we make use of the basic Avalon bus, we limit ourselves to 16 bits of transfer, requiring five separate consecutive write steps in order to write one column. This is then repeated 640 times until one of the write buffers is filled. Such a scheme does necessitate which “substep” of

[1] Technique suggested by Prof Edwards.

the writing process we are currently on. We found this small extra complexity preferable to using 640*5 different write locations to indicate which data we are receiving.

As the hardware requires the data to come in sequentially, it introduces some brittleness to the design, but we have not observed any issues in practice. Nonetheless, we do add a **column reset** register to the hardware in order to ensure we start the software with the hardware expecting column data piece 1 of column 1.

4.1.3 Six Stage Pixel Pipeline

We are constantly calculating information for pixels ahead of the one currently being displayed.

Our VGA module runs at half speed so we can perform the work of two clocks for every pixel displayed. We start by determining (i.e. row and column) the pixel, three pixels in the future, will be. This is trivial when you are in the middle of a particular row, but can become tricky when you are towards the end of a row, or the end of the last row as well. We have special logic to deal with this in the verilog code.

Once this is determined, we fetch the column data for the pixel in question. We then wait two cycles for the column data to arrive.

In stage 4 of the pipeline, we perform the calculation to determine which texture data we require. The register in the texture module is changed and we then wait 2 cycles (due to double barrel design) to get the correct texture information in the form of RGB values.

By stage six we have the rgb values ready at our disposal and set the VGA output values as necessary. These values will be the same every two cycles and the picture is properly displayed.

When we are dealing with ceiling or floor pixels, the texture data is effectively garbage from the prior time it was retrieved and is discarded.

4.1.4 Triple Buffering

Although not necessary, it was an interesting experiment to have our data writing scheme be asynchronous. The software does not necessarily need to wait for the time in-between frames to change the column data. We have a total of 3 separate column buffers which hold the 640 columns of data that make up one frame. At any given time, one is being read from (front buffer), and the other two can trade off being written to (back buffers)[2]. At the end of every frame, the module asks “which column buffer has the freshest, **and fully written** column data”.

This scheme ensures:

1. Column data can be written to at any time (asynchronous)

[2] https://www.anandtech.com/show/2794/2

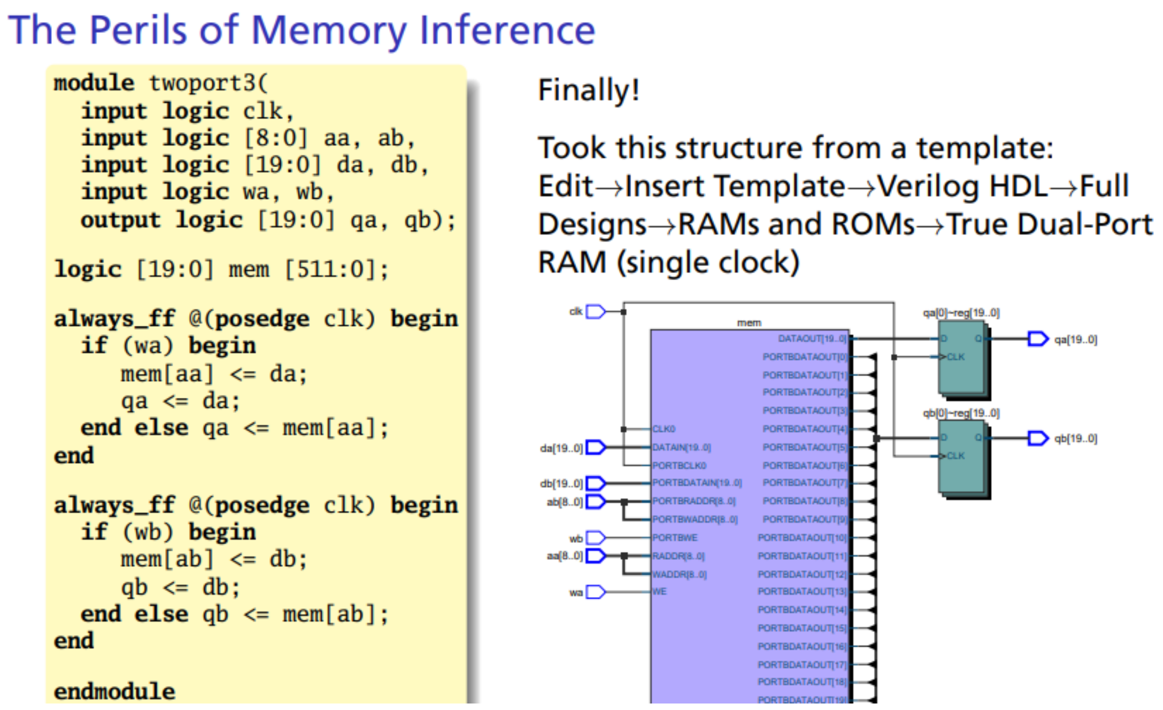
2. There are no changes to column data mid frame which would lead to screen tearing
3. We always display the most recent frame

If there was only one back buffer, and we wanted to ensure that we never read from a buffer which was in the midst of being written to, there is a scenario where frames are skipped for an arbitrary amount of time, if the column data came in only partially in between frames. Exceedingly unlikely, but making the design **correct** is important nonetheless.

4.1.5 Scaling Factors - Several Approaches

Initially, it was thought that there could only be 480 (number of screen rows) possible scaling factors since the main changing variable in the formula was the projected wall height. We came up with a dictionary lookup table to find the preset scaling factors. However it was later realized that there were far many more possible values, and it was determined that it would be better to transmit the scaling factor precalculated as columnar data.

4.1.6 Array Storage and Retrieval - Double Barrel Design



The Perils of Memory Inference

```
module twoport3(
  input logic clk,
  input logic [8:0] aa, ab,
  input logic [19:0] da, db,
  input logic wa, wb,
  output logic [19:0] qa, qb);

logic [19:0] mem [511:0];

always_ff @(posedge clk) begin
  if (wa) begin
     mem[aa] <= da;
     qa <= da;
  end else qa <= mem[aa];
end

always_ff @(posedge clk) begin
  if (wb) begin
     mem[ab] <= db;
     qb <= db;
  end else qb <= mem[ab];
end

endmodule
```

Finally!

Took this structure from a template:
Edit→Insert Template→Verilog HDL→Full Designs→RAMs and ROMs→True Dual-Port RAM (single clock)

3

We utilized the blueprint from Professor Edwards' memory lecture (above) to create a double barrel memory design. Anecdotally, with the incorrect design initially attempted, our compilation failed because we used up all the SRAM blocks. This is because the compiler incorrectly blew up the design just like the lecture said it would.

[3] http://www.cs.columbia.edu/~sedwards/classes/2022/4840-spring/memory.pdf

For the texture data, we made use of the **readmemh** function in verilog to read in thousands of hex values for the registers, creating essentially a lookup table. Because the data was fixed, it made sense to do it this way.

4.2 TILE-BASED CHARACTER RENDERER

4.2.1 General Principles

The character renderer piece of the hardware will render a white monospace character overlayed on top of the rest of the output. It operates by accepting the first 128 ASCII characters and specifying which position they are on the 80x30 tile grid on the screen. It can specify whether the character should be white on a transparent background, or vice versa.

4.2.2 Two Stage Pixel Pipeline

Similar, yet simpler than the column decoding functions, we accomplish our computation in all but 2 clock cycles.

We need four pieces of data, and we aim to find this out for the **next** pixel: character column, column offset, character row, and row offset. Offset is the exact pixel within a row or column.

The offsets are calculated by low bit count registers that simply wrap around and reset to zero after being incremented so many times. The characters are 8x16 thus are sized perfectly for such a technique.

We then ask the character module to fetch the yes/no value at that exact location, which takes only 1 cycle due to the simpler design of the module vs the columns module.

When we display the data with the vga pins, we give preference to the character data, if it is in a “yes” position, we write a white rgb value to the wires. We did not implement different text colors, though this would be trivial.

4.2.3 Font Data Storage and Retrieval

Likewise we employ the use of the **readmemh** function in verilog to load the font data initially. The font data was obtained from the C array in lab two[4]. There is also the requirement of storing 30x80 different character values at the different row/col positions. This can be specified by the driver speaking over the Avalon bus.

Between the column, row, character, and background color: writing any character requires 20 total bits. Once again we write in two separate stages, though in this case we do create discrete

[4] http://www.cs.columbia.edu/~sedwards/classes/2022/4840-spring/lab2.tar.gz

write locations for the two. Any time the second write stage is written to, the value is automatically updated in the character module.

We use an assign statement in the character module to save one cycle for the retrieval step. Why this works with the simpler design and not the more complex and bit-heavy columns module design is an open question.

An aside: to set the character to a blank value, we chose the natural state of “space” to represent this since no special codes were necessary. Indeed, the array is set to all spaces in the verilog file for the initial startup of the hardware.

5 SOFTWARE

5.1 Game Loop

After the lookup tables for trigonometry are initially filled in; hardware, driver, thread elements are set up, the main function starts running a continuous “game loop”. The loop simply examines the player’s current position in the selected map, checks if the position needs to change, and ensures that the player is not running into a wall. Then it calls the render function.

The main loop can also decide, based on the current state of the game, to call the initial menu select, pause menu (a variation of menu select), or finish the game with a special message.

5.2 Renderer

The render routine, when called, will go through each column of the screen, marching rays out from the viewer's field of view, and calculate all relevant column data.

We separate the calculation of column data from the sending of column data for timing purposes. We want the sending of data to happen extremely quickly. Therefore, the column data is stored in a struct, **columns_t**.

The renderer then polls the hardware about its frame status, waiting for when it is in the VBLANK in between frame status, and then sends a pointer to the columns struct to the driver to be sent off to hardware land.

Because of our semi-modern ARM cpu, the rendering loop happens extremely quickly, allowing 60 frames per second to easily be achieved.

5.3 Pause Menu, Finish Game

The pause menu takes advantage of our tile character renderer, showing the available maze choices. The USB controller or keyboard navigates up or down the menu, and the currently selected maze is indicated by the white background text. The start button on the controller or the enter key on the keyboard will exit the menu and start the selected level.

The pause menu is a variation of this, with the addition of saving the current grid position and angle of the player should they decide to return to the currently selected maze being played.

5.4 Timing With VBLANK

As we previously alluded to, the render function makes a series of **ioctl** calls to the driver, checking for vblank status. It sleeps for 50 microseconds then checks again. The polling design is simple, but less efficient than the more complex interrupt-based method.

Only after the renderer gets the go-ahead that the display output is in between frames - whether that be the front or back porch, it sends out the column data to the driver. The driver handles the rest.

5.5 Maps / Maze

The **maze_t** struct includes width, height, area (for bypassing the constant multiplication), name, and the maze itself as described by an array of shorts. In retrospect, the maze could have been type uint_8* for more space savings.

Each maze is simply an array of numbers 0-8. 0 indicates the absence of a wall (walkable floor area) and 1-8 are the different wall textures. For extra readability, the numbers are changed to defines of single capital letters corresponding to abbreviations:

```
#define B 1 // bluestone
#define C 2 // colorstone
#define E 3 // eagle (end level)
#define G 4 // greystone
#define M 5 // mossy
#define P 6 // purplestone
#define R 7 // redbrick
#define W 8 // wood

#define O 0 // opening (floor)
```

```
typedef struct {
      int width;
      int height;
      int area;
      char name[20];
      short map[1296]; //make this largest area of any map
} maze_t;
```

5.6 USB Interface & Threads

The keyboard functionality was largely modeled after the skeleton code in lab 2. The code operates in a separate thread, checking for changes in the key state array. Global variables are modified which ultimately are read by the game loop.

Likewise, the controller exists on its own thread as well. Some trial and error was required to find out the precise values of different buttons at-rest and depressed.

The game loop examines both inputs and decides what move, if any, is happening, at any given time.

6 HARDWARE-SOFTWARE INTERFACE

6.1 General Principles

The driver to communicate with the hardware is modeled from the skeleton files provided with lab three[5]. The drivers provides several overarching functions:

- Reset columns
- Send columns data
- Blackout screen
- Un-blackout screen
- Set char at position
- Get vblank state

All but the last are write operations, while the last is a read operation, returning simply a true or false (1 or 0) value.

6.2 Send Columns & Timing

[5] http://www.cs.columbia.edu/~sedwards/classes/2022/4840-spring/lab3-sw.tar.gz

The driver is responsible for copying over the columns data from the pointer in userland. Once it has the data, it runs a loop for each column, splitting the data into its five separate parts based on our specification. It combines certain data through the use of bit shifting and bitwise OR operations.

For each of the five prepared parts of the data, it sends it sequentially. The order greatly matters, as they are all going to the same memory location. The hardware is set to accept the data in this particular order and keeps track of where it is in the data receiving process.

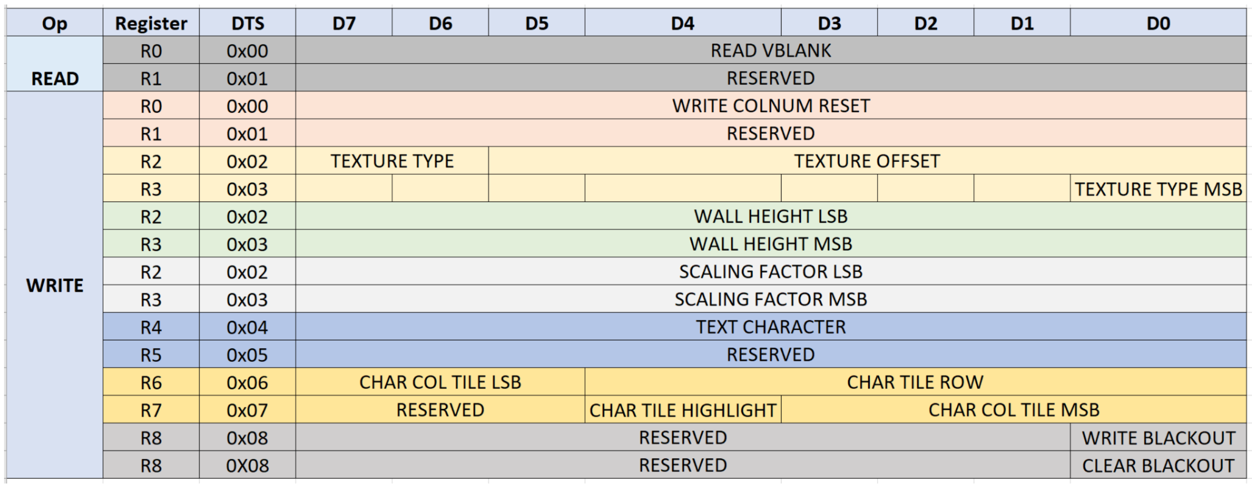
There is no handshake negotiation process going on, so the transmission is not robust. In the future, it would be wise to have a kind of confirmation sent back.

The scaling factor is calculated in driver land, and its design is sub optimal. This should be moved over to userland in a future iteration for good practices. Drivers should do as little calculations as possible.

6.3 Tile Based Character Rendering

A paired-down version of the above method is employed for character writing inside the driver. The driver shifts bit values as appropriate to match the expected format inside the hardware. Only two write steps are necessary due to the smaller data size. Each of these steps has its own convenient memory address to write to based on the hardware design spec, making the process even simpler.

6.4 Register Map



Op	Register	DTS	D7	D6	D5	D4	D3	D2	D1	D0
READ	R0	0x00	READ VBLANK							
	R1	0x01	RESERVED							
WRITE	R0	0x00	WRITE COLNUM RESET							
	R1	0x01	RESERVED							
	R2	0x02	TEXTURE TYPE		TEXTURE OFFSET					
	R3	0x03								TEXTURE TYPE MSB
	R2	0x02	WALL HEIGHT LSB							
	R3	0x03	WALL HEIGHT MSB							
	R2	0x02	SCALING FACTOR LSB							
	R3	0x03	SCALING FACTOR MSB							
	R4	0x04	TEXT CHARACTER							
	R5	0x05	RESERVED							
	R6	0x06	CHAR COL TILE LSB			CHAR TILE ROW				
	R7	0x07	RESERVED			CHAR TILE HIGHLIGHT	CHAR COL TILE MSB			
	R8	0x08	RESERVED							WRITE BLACKOUT
	R8	0X08	RESERVED							CLEAR BLACKOUT

7 RESOURCE BUDGET

7.1 RESOURCE TABLE

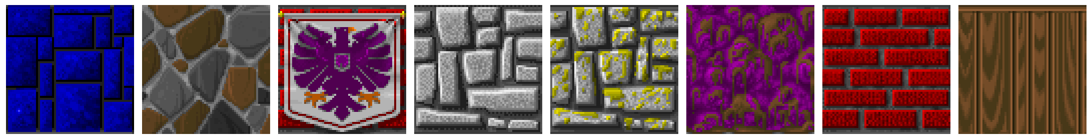
In memory objects that we keep inside the very fast SRAM and avoid using DRAM.

Name	Description	Size	Num Elements
Column Tuple	top_of_wall (16 signed), projected_height (16), wall_side (1), texture_offset (6), texture_type (3), scaling_factor(32)	74 bits	640 *3 (3 buffers)
Texture	64x64 array of RGB values (alpha assumed to be 1)	64 * 64 * 3 bytes (12.288kb) each	8
Character Font	16x8 array of single bits representing white or transparent for each ASCII character	128 bits	128
Character Tile Data	Array of characters at each row / column	7 bits	30*80

Based on the above, our SRAM memory needs are modest, coming in at 98.3kb for textures + 17.3kb for column tuples, and 2kb for all character font data, and another 2kb for character tile data.

7.2 GRAPHICS

For texturing we are using the freely provided wolfenstein .png files available at https://lodev.org/cgtutor/files/wolftex.zip. Here they are in all their 64x64 glory:



The popular ImageMagick library was used to output the files in a series of RGB values. With some light massaging using grep, awk, etc - we were able to get the texture values as one long file of hex values representing the memory state of the texture module in the hardware.

The floor was rendered as a static shade of gray, taking inspiration from a past project and the original Wolfenstein game.

The “ceiling” was rendered as a blue tone which had a nice gradient effect where it faded to white as the rows went down the screen. It was a simple effect to implement but added a liveliness to the final result.

8 MILESTONES

All milestones set forth in the design document were achieved. They are left for posterity.

8.1 Milestone 1

Software rendering prototype which writes data to framebuffer. Key design consideration to split up the raycasting loop and column renderer. The column renderer will be what becomes our hardware.

8.2 Milestone 2

Initial hardware implementation in Verilog. Loading of textures into FPGA as initial contents of ROM block using **MIF** (memory initialization files) in Quartus.[6]

Column decoding and display only at this milestone (untextured).

8.3 Milestone 3

Texturing in hardware. Basic gamification of maze’s. Multiple levels. Game completion screen.

[6] https://edstem.org/us/courses/17891/discussion/1332039

9 PLAYER INPUT

FPGA Raycasting uses libusb to receive and decode button presses from a USB HID NES Controller.

We also allow the control from the keyboard simultaneously to allow for maximum flexibility.

If neither are plugged in, the program swiftly exits with an error message.

10 RESULTS

Largely, we were able to achieve all the goals set forth in the design document. We have a smooth, 60fps, perfectly timed, textured output running through wolfenstein-like levels. We have a tile based text rendering system that is used to communicate game state to the player. For the time budget of this project the goals achieved were reasonable and substantial.

11 LESSONS LEARNED

The greatest lesson learned was that unlike software, hardware can be doing many things at once. It is a completely different paradigm. Initially, this seems like a weakness, but it can be exploited as an enormous strength.

A great limitation however, is how much can be done in one clock cycle. Similarly, one cannot share the results of multiple parallel operations until the next cycle.

This leads to the obvious necessity of **pipelining**. It can be challenging to get a grip on this concept but once understood, the sky is truly the limit.

That being said, our project was very amenable to pipelining. There may be other algorithms that do not admit a parallel-like solution so easily.

This is a classic case of, “the more you know, the more you realize how much you do not know”. Therefore we will refrain from extending our lessons too far; there is much to learn in this field, and this is just the initial exposure.

Another lesson we can derive is the importance of heeding the Professor’s advice and creating a simulation setup in Verilator for timing issues. Our approach was always trial and error when it came to timing issues, and in the end, it slowed us down. Some time invested in the beginning would have saved us much troubleshooting time later on.

12 GROUP MEMBER CONTRIBUTIONS

12.1 Adam

- Port initial Java demo code to create software mockup
- Create column decoder hardware
- Implement tile based text renderer
- Driver for software hardware interface
- Testing and bug fixing

12.2 Souryadeep

- USB controller work
- Develop parallel unused implementation of text tile renderer
- Testing and bug fixing
- Timing and mutex considerations
- Software model for game start and level select logic

13 FUTURE WORK

13.1 Floor / Ceiling Texturing

There exists a technique to cast additional rays towards the floor and ceiling in order to perform a similar effect in terms of texturing these surfaces. This would require an entire reworking of the architecture and is unlikely to be accomplished within the timeframe necessary.

13.2 Jumping

Of all the stretch goals, this seems like the most feasible, as it only requires the hardcoded camera height to be adjusted **in software** - as far as the hardware is concerned, it is still decoding the same data. Making jumping have a raison d'être is another matter entirely.

Update: This was implemented for demonstration purposes, but it has no value in terms of gameplay. A simple way to make this meaningful would be to have “pits” where the floor needs to be jumped over. This would require **12.1** to put into action, however.

13.3 Music / Sound Effects

Incorporation of an FM synthesis chip (off-the-shelf design) and hardcode some primitive sound effects.

13.4 On Screen Elements and Enemies

This could be accomplished with some kind of sprite scaling technique. Though the original Wolfenstein ran on computers that lacked any kind of sprite hardware.

14 CODE LISTING

14.1 Hardware

(not shown textures.mem and font.mem)

14.1.1 column.decoder.sv

```
/*
 * Avalon memory-mapped peripheral that generates VGA signal from
 * column data in ray casting context.
 *
 * Adam Carpentieri AC4409
 * Columbia University

 */

module column_decoder(input logic clk,
          input logic       reset,
            input logic          write,
            input                    chipselect,
        input logic [3:0]   address,
        input logic [15:0]  writedata,

        output logic [15:0] readdata,

            output logic [7:0] VGA_R, VGA_G, VGA_B,
            output logic         VGA_CLK, VGA_HS, VGA_VS,
                               VGA_BLANK_n,
            output logic         VGA_SYNC_n);

    logic [10:0]  hcount;
    logic [9:0]     vcount, vcount_1_ahead;

    logic [2:0]      cur_col_write_stage = 3'h0; //which of 3 write stages
per column
    logic [1:0]      col_module_index_to_read = 2'b00; //which columns
module to read
    logic [1:0]      col_module_index_to_write = 2'b01;  //which columns
module to write to
    logic [2:0]      col_write = 3'b0;
    logic            new_columns_ready = 1'b0;

    logic [9:0]      cur_col_first_write_stage_data;
    logic [15:0]     cur_col_second_write_stage_data;
    logic [15:0]     cur_col_third_write_stage_data;
    logic [15:0]     cur_col_fourth_write_stage_data;

    logic [9:0]  colnum [2:0];
    logic [41:0] new_coldata [2:0];
    logic [41:0] col_data [2:0];
    logic [31:0] new_sfdata[2:0];
    logic [31:0] sf_data [2:0];

    logic [2:0] texture_type_select = 1'b0;
    logic [5:0] texture_row_select = 6'b0;
    logic [5:0] texture_col_select = 6'b0;
    logic [23:0] cur_texture_rgb_vals = {8'hff, 8'hff, 8'hff}; //output

    logic [2:0] pixel_type = 3'b0; //0: ceiling, 1: wall, 2: floor
    logic       pixel_wall_dir = 1'b0;

    logic [2:0]  next_pixel_type = 3'b0; //0: ceiling, 1: wall, 2: floor
    logic        next_pixel_wall_dir = 1'b0;
    logic [23:0] next_pixel;
    logic blackout_screen = 1'b0;

    logic freeze_pipeline = 1'b0; //freeze the pixel pipeline (during
vga_blank_n)

    //character generator state
    logic [4:0] char_write_row;
    logic [6:0] char_write_col;
    logic [7:0] char_write_char;
    logic       char_write_highlight;
    logic       char_write = 1'b0;

    logic [4:0] char_cur_row = 5'h0;
    logic [3:0] char_cur_row_offset = 4'h0;
    logic [6:0] char_cur_col = 7'h0;
    logic [2:0] char_cur_col_offset = 3'h0;

    logic       char_on = 1'b0;

    columns columns0 (clk, reset, col_write[0], colnum[0], new_sfdata[0],
new_coldata[0], sf_data[0], col_data[0]),
            columns1 (clk, reset, col_write[1], colnum[1], new_sfdata[1],
new_coldata[1], sf_data[1], col_data[1]),
            columns2 (clk, reset, col_write[2], colnum[2], new_sfdata[2],
new_coldata[2], sf_data[2], col_data[2]);

    textures textures0 (texture_type_select,
        texture_row_select,
        texture_col_select,
        cur_texture_rgb_vals
    );

    vga_counters counters (.clk50(clk), .*);

    chars chars0 (.*);

    always_ff @(posedge clk) begin

        if (reset) begin

            {colnum[0], colnum[1], colnum[2]} <= 30'h0;
            cur_col_write_stage <= 3'h0;
            col_module_index_to_read <= 2'b00;
            col_module_index_to_write <= 2'b01;
            col_write <= 3'b0;

        end else if (chipselect && write) begin

            //reset col num
            if(address == 4'h0) begin
                cur_col_write_stage <= 3'h0;
                col_write[col_module_index_to_write] <= 1'h0;
                colnum[col_module_index_to_write] <= 10'b0;
            end

            else if(address == 4'h1) begin

                //1st write stage
                if(!cur_col_write_stage) begin

                    cur_col_first_write_stage_data <= writedata[9:0];
                    col_write <= 3'b0;
                    cur_col_write_stage <= cur_col_write_stage + 3'h1;

                //second write stage
                end else if(cur_col_write_stage == 3'h1) begin
                    cur_col_second_write_stage_data <= writedata;
                    cur_col_write_stage <= cur_col_write_stage + 3'h1;

                //third write stage
                end else if(cur_col_write_stage == 3'h2) begin
                    cur_col_third_write_stage_data <= writedata;
                    cur_col_write_stage <= cur_col_write_stage + 3'h1;

                //fourth write stage
                end else if(cur_col_write_stage == 3'h3) begin
                    cur_col_fourth_write_stage_data <= writedata;
                    cur_col_write_stage <= cur_col_write_stage + 3'h1;

                //fourth write stage
                end else begin

                    if(colnum[col_module_index_to_write] == 10'h27F) begin
//639

                        col_module_index_to_write <= 2'b11 ^
col_module_index_to_write ^ col_module_index_to_read;
                        colnum[2'b11 ^ col_module_index_to_write ^
col_module_index_to_read] <= 10'b0;
                        new_columns_ready <= 1'b1;
                    end
                    else
                        colnum[col_module_index_to_write] <=
colnum[col_module_index_to_write] + 10'b1; //increment col num

                    new_coldata[col_module_index_to_write] <=
{cur_col_third_write_stage_data, cur_col_second_write_stage_data,
cur_col_first_write_stage_data};
                    new_sfdata[col_module_index_to_write] <=
{cur_col_fourth_write_stage_data, writedata};
                    col_write[col_module_index_to_write] <= 1'h1;
                    cur_col_write_stage <= 3'h0;

                end
            end

            else if(address == 4'h2) begin
                char_write_char <= writedata[7:0];
            end

            else if(address == 4'h3) begin
                char_write_row <= writedata[4:0];
                char_write_col <= writedata[11:5];
                char_write_highlight <= writedata[12];
                char_write <= 1'b1;
            end

            else if(address == 4'h4) begin
                blackout_screen <= writedata[0];
            end

            if(address != 4'h3)
                char_write <= 1'b0;
        end
        else begin
            col_write <= 3'b0;
            char_write <= 1'b0;
        end

    //pixel pipeline
   // always_ff @(posedge clk)
```

```
        if(hcount == 11'h4ff) //1279 (639)
            freeze_pipeline <= 1'b1;

        else if(hcount == 11'h638 && (vcount < 10'h1e0 || vcount ==
10'h20c)) //1591, 480, 524
            freeze_pipeline <= 1'b0;

    //pipeline stage 1 - retrieve column data
   // always_ff @(posedge clk)

        if(!freeze_pipeline)
            colnum[col_module_index_to_read] <= hcount < 11'h500 //1280
(4f8) and 797(31d)
                ? hcount[10:1] + 10'h2
                : hcount[10:1] - 10'h31d;

    //(pipeline stage 2 and 3 is just waiting for column data)

    //pipeline stage 4 - use column data to set texture registers
    //always_ff @(posedge clk)

        //ceil
        if(vcount_1_ahead < col_data[col_module_index_to_read][41:26] &&
!col_data[col_module_index_to_read][41]) //top of wall
            pixel_type <= 2'h0;

        //floor
        else if(vcount_1_ahead >
($signed(col_data[col_module_index_to_read][41:26]) +
col_data[col_module_index_to_read][25:10]))
            pixel_type <= 2'h2;

        else begin //wall

            pixel_type <= 2'h1;
            pixel_wall_dir <= col_data[col_module_index_to_read][9];

            texture_type_select <= col_data[col_module_index_to_read][8:6];
            texture_col_select <= col_data[col_module_index_to_read][5:0];
            texture_row_select <= ((vcount_1_ahead -
$signed(col_data[col_module_index_to_read][41:26])) *
sf_data[col_module_index_to_read]) >> 5'h19;
        end
    //pipeline stage 5 - pass the baton (introduce artificial delay for
timing - want on even cycles)
        next_pixel_type = pixel_type;

        if(pixel_type == 2'h1) begin

            next_pixel <= cur_texture_rgb_vals;
            next_pixel_wall_dir <= pixel_wall_dir;
        end

    //swap out column module to read from if new avail and in between
frames
    //always_ff @(posedge clk)

        if(vcount == 10'h20b && new_columns_ready) begin // 523

            new_columns_ready <= 1'b0;
            col_module_index_to_read <= 2'b11 ^ col_module_index_to_read ^
col_module_index_to_write;
        end

        //char row
        if(vcount >= 10'h1e0) begin //480
            char_cur_row <= 5'h0;
            char_cur_row_offset <= 4'h0;

        end else if(hcount[10:1] == 10'h280 && !hcount[0]) begin //640

            char_cur_row_offset <= char_cur_row_offset + 4'h1;

            if(char_cur_row_offset == 4'hf)
                char_cur_row <= char_cur_row + 5'h1;
        end

        //char col
        if(hcount[10:1] >= 10'h280) begin //640

            char_cur_col <= 7'h0;
            char_cur_col_offset <= 3'h0;

        end else if(hcount[10:1] < 10'h280 && !hcount[0]) begin //640

            char_cur_col_offset <= char_cur_col_offset + 3'h1;
            if(char_cur_col_offset == 3'h7)
                char_cur_col <= char_cur_col + 7'h1;
        end

    end
    always_comb begin

        //"pipeline" stage 6 (current clock)

        vcount_1_ahead = hcount >= 11'h63e //1598
            ? ( vcount > 10'h1df ? 10'h0 : vcount + 10'h1 )
            : vcount;

        {VGA_R, VGA_G, VGA_B} = {8'h0, 8'h0, 8'h0};

        if (VGA_BLANK_n) begin

            if(!blackout_screen) begin

                if(next_pixel_type == 2'h0) //ceil
                    {VGA_R, VGA_G, VGA_B} = {(vcount[8:1] + 8'h00),
(vcount[8:1] + 8'h00), 8'hff};

                else if(next_pixel_type == 2'h2) //floor
                    {VGA_R, VGA_G, VGA_B} = {8'h40, 8'h40, 8'h40};

                else if(!next_pixel_wall_dir) //wall faded
                    {VGA_R, VGA_G, VGA_B} = {(next_pixel[23:16]>>1),
(next_pixel[15:8]>>1), (next_pixel[7:0]>>1)};

                else //wall full brightness
                    {VGA_R, VGA_G, VGA_B} = next_pixel;
            end

            //text character
            if(char_on)
                {VGA_R, VGA_G, VGA_B} = {8'hff, 8'hff, 8'hff};
        end

        //for vblank detection
        if(vcount > 10'h1df)
            readdata = 16'h1;
        else
            readdata = 16'h0;
    end

endmodule

module columns(
    input logic clk, reset, write,
    input logic [9:0] col_num,
    input logic [31:0] new_sf_data,
    input logic [41:0] new_col_data,
    output logic [31:0] sf_data,
    output logic [41:0] col_data
);

    //declare array https://www.chipverify.com/verilog/verilog-arrays
    logic [41:0] columns [639:0];
    logic [31:0] sfs [639:0]; //scaling factors

    integer i;

    initial begin
        for (i=10'h0; i<10'h280; i=i+10'h1) begin
            columns[i] <= 42'b0;
            sfs[i] <= 32'b0;
        end
    end

    always_ff @(posedge clk) begin
        if(write) begin

            columns[col_num] <= new_col_data;
            col_data <= new_col_data;
        end else
            col_data <= columns[col_num];
    end

    always_ff @(posedge clk) begin
        if(write) begin

            sfs[col_num] <= new_sf_data;
            sf_data <= new_sf_data;
        end else
            sf_data <= sfs[col_num];
    end
```

```
endmodule

//types: 0: bluestone, 1: colorstone, 2: eagle, 3: greystone, 4: mossy, 5:
purplestone, 6: redbrick, 7: wood
module textures(
    input logic [2:0] texture_type,
    input logic [5:0] row,
    input logic [5:0] col,
    output logic [23:0] texture_data
);

    logic [23:0] textures [0:32767]; //texture type, row num, col num

    //https://projectf.io/posts/initialize-memory-in-verilog/
    initial begin
        //$display("Loading textures.");
        $readmemh("textures.mem", textures);
    end

    assign texture_data = textures[{texture_type, row, col}];

endmodule

module chars(
    input logic clk,

    input logic [4:0] char_write_row,
    input logic [6:0] char_write_col,
    input logic [7:0] char_write_char,
    input logic       char_write_highlight,
    input logic       char_write,

    input logic [4:0] char_cur_row,
    input logic [3:0] char_cur_row_offset,
    input logic [6:0] char_cur_col,
    input logic [2:0] char_cur_col_offset,

    output logic char_on
);

    logic [7:0] char_data [0:2047];
    logic [7:0] chars [2399:0];
    logic       chars_highlight [2399:0];
    logic [21:0] char_data_index;
    logic [21:0] char_index_to_write = 22'b0;
    logic        char_write_now = 1'b0;

    logic char_highlight_to_write;
    logic [7:0] char_to_write;

    logic char_highlight;

    integer i;

    initial begin
        //$display("Loading font.");
        $readmemh("font.mem", char_data);
    end

    initial begin
        for (i=12'h0; i<12'h960; i=i+12'h1) begin
            chars[i] <= 8'h20; //space
            chars_highlight[i] <= 1'b0;
        end
    end

    always_ff @(posedge clk) begin

        if(char_write) begin
            char_index_to_write <= (char_write_row * 11'h50) +
char_write_col;
            char_to_write <= char_write_char;
            char_highlight_to_write <= char_write_highlight;
            char_write_now <= 1'b1;
        end

        if(char_write_now) begin
            chars[char_index_to_write] <= char_to_write;
            chars_highlight[char_index_to_write] <= char_write_highlight;
            char_write_now <= 1'b0;
        end

        //pipeline step 1
        char_data_index <=
            ((chars[(char_cur_row * 12'h50) + char_cur_col]) * 12'h10)
            + char_cur_row_offset;
        char_highlight <= chars_highlight[(char_cur_row * 12'h50) +
char_cur_col];
    end

    assign char_on = char_data[char_data_index[11:0]][char_cur_col_offset]
- char_highlight;

endmodule


module vga_counters(
 input logic           clk50, reset,
 output logic [10:0] hcount,  // hcount[10:1] is pixel column
 output logic [9:0]  vcount,  // vcount[9:0] is pixel row
 output logic          VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n);

/*
 * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other cycle
 *
 * HCOUNT 1599 0             1279       1599 0
 *
 *            _______________            ________
 * ___________|    Video      |____________|  Video
 *
 *
 * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
 *
 *      _______________________     _____________
 * |____|       VGA_HS          |____|
 */
   // Parameters for hcount
   parameter HACTIVE      = 11'd 1280,
             HFRONT_PORCH = 11'd 32,
             HSYNC        = 11'd 192,
             HBACK_PORCH  = 11'd 96,
             HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC +
                            HBACK_PORCH; // 1600

   // Parameters for vcount
   parameter VACTIVE      = 10'd 480,
             VFRONT_PORCH = 10'd 10,
             VSYNC        = 10'd 2,
             VBACK_PORCH  = 10'd 33,
             VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC +
                            VBACK_PORCH; // 525
   logic endOfLine;

   always_ff @(posedge clk50 or posedge reset)
     if (reset)          hcount <= 0;
     else if (endOfLine) hcount <= 0;
     else            hcount <= hcount + 11'd 1;

   assign endOfLine = hcount == HTOTAL - 1;

   logic endOfField;

   always_ff @(posedge clk50 or posedge reset)
     if (reset)          vcount <= 0;
     else if (endOfLine)
       if (endOfField)   vcount <= 0;
       else              vcount <= vcount + 10'd 1;

   assign endOfField = vcount == VTOTAL - 1;

   // Horizontal sync: from 0x520 to 0x5DF (0x57F)
   // 101 0010 0000 to 101 1101 1111
   assign VGA_HS = !( (hcount[10:8] == 3'b101) &
                  !(hcount[7:5] == 3'b111));
   assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);

   assign VGA_SYNC_n = 1'b0; // For putting sync on the green signal;
unused

   // Horizontal active: 0 to 1279     Vertical active: 0 to 479
   // 101 0000 0000  1280            01 1110 0000  480
   // 110 0011 1111  1599            10 0000 1100  524
   assign VGA_BLANK_n = !( hcount[10] & (hcount[9] | hcount[8]) ) &
                  !( vcount[9] | (vcount[8:5] == 4'b1111) );

   /* VGA_CLK is 25 MHz
    *             __    __    __
    * clk50    __|  |__|  |__|
    *
    *             _____       __
    * hcount[0]__|     |_____|
    */
   assign VGA_CLK = hcount[0]; // 25 MHz clock: rising edge sensitive

endmodule
```

14.2 Software

14.2.1 column_decoder.c

```
/* * Device driver for the Raycasting Column Decoder
 *
 * A Platform device implemented using the misc subsystem
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * Modified by Adam Carpentieri AC4409
 * Columbia University Spring 2022
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *               drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod column_decoder.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree column_decoder.c
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/delay.h>
#include <asm/types.h>
```

```
#include "column_decoder.h"

#define DRIVER_NAME "column_decoder"

/* Device registers */
#define WRITE_COLNUM_RESET(x) (x)
#define WRITE_COLS(x) ((x)+2)
#define WRITE_CHAR_S1(x) ((x)+4)
#define WRITE_CHAR_S2(x) ((x)+6)
#define WRITE_BLACKOUT(x) ((x)+8)
#define READ_VBLANK(x) (x)

/*
 * Information about our device
 */
struct column_decoder_dev {
      struct resource res; /* Resource: our register */
      void __iomem *virtbase; /* Where register can be accessed in memory
*/
      columns_t *columns;
} dev;


//ensure the next write will be for the first column
static void reset_col_num(void) {

      iowrite8(0x00, WRITE_COLNUM_RESET(dev.virtbase)); //value does not
matter
}

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
static void write_columns(columns_t *columns)
{
      __u16 bits_to_send;
      __u16 i;
      __u32 scaling_factor;
      column_arg_t column_arg;

      for(i=0; i<640; i++) {

            column_arg = columns->column_args[i];
            bits_to_send = 0x0000 | (column_arg.wall_side << 9);
            bits_to_send |= (column_arg.texture_type << 6);
            bits_to_send |= column_arg.texture_offset;
            iowrite16(bits_to_send, WRITE_COLS(dev.virtbase));

            iowrite16(column_arg.wall_height, WRITE_COLS(dev.virtbase));

            iowrite16(column_arg.top_of_wall, WRITE_COLS(dev.virtbase));

            //TODO figure out why it needs to be multiplied by -1:
otherwise texture in hardware is upside down
            scaling_factor = column_arg.wall_height ? (0x00000000 | ((64 <<
25) / column_arg.wall_height)) * -1 : 0x00000000;

            bits_to_send = (__u16) ((scaling_factor & 0xFFFF0000) >> 16);
            iowrite16(bits_to_send, WRITE_COLS(dev.virtbase));

            bits_to_send = (__u16) (scaling_factor & 0x0000FFFF);
            iowrite16(bits_to_send, WRITE_COLS(dev.virtbase));
      }

      //TODO copy column data manually
      //dev.columns = columns;
}


static void write_char(char_tile_t *char_tile) {

      __u16 s2_val;

      iowrite8(char_tile->char_val, WRITE_CHAR_S1(dev.virtbase));

      s2_val = (0x0000 | char_tile->highlight) << 12;
      s2_val |= ((0x0000 | char_tile->col) << 5);
      s2_val |= char_tile->row;

      iowrite16(s2_val, WRITE_CHAR_S2(dev.virtbase));
}

static void blackout_screen(void) {
      iowrite16(0x0001, WRITE_BLACKOUT(dev.virtbase));
}
```

```
static void remove_blackout_screen(void) {
      iowrite16(0x0000, WRITE_BLACKOUT(dev.virtbase));
}

static __u8 read_vblank(void) {

      return (__u8) ioread16(READ_VBLANK(dev.virtbase));
}


/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long column_decoder_ioctl(struct file *f, unsigned int cmd, unsigned
long arg)
{
      columns_t   columns;
      char_tile_t char_tile;
      __u8        vblank_val;

      switch (cmd) {

      case COLUMN_DECODER_RESET_COL_NUM:
            reset_col_num();
            break;

      case COLUMN_DECODER_WRITE_COLUMNS:

            if (copy_from_user(&columns, (columns_t *) arg,
sizeof(columns_t)))
                  return -EACCES;

            write_columns(&columns);

            break;

      case COLUMN_DECODER_WRITE_CHAR:

            if (copy_from_user(&char_tile, (char_tile_t *) arg,
sizeof(char_tile_t)))
                  return -EACCES;
            write_char(&char_tile);

            break;

      case COLUMN_DECODER_BLACKOUT_SCREEN:

            blackout_screen();

            break;

      case COLUMN_DECODER_REMOVE_BLACKOUT_SCREEN:
            remove_blackout_screen();

            break;

      case COLUMN_DECODER_READ_VBLANK:

            vblank_val = read_vblank();

            if (copy_to_user((__u8 *) arg, &vblank_val, sizeof(__u8)))
                  return -EACCES;

            break;

      default:
            return -EINVAL;
      }

      return 0;
}

/* The operations our device knows how to do */
static const struct file_operations column_decoder_fops = {
      .owner            = THIS_MODULE,
      .unlocked_ioctl = column_decoder_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev
*/
static struct miscdevice column_decoder_misc_device = {
      .minor            = MISC_DYNAMIC_MINOR,
      .name       = DRIVER_NAME,
      .fops       = &column_decoder_fops,
};
```

```
/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init column_decoder_probe(struct platform_device *pdev)
{

      int ret;

      /* Register ourselves as a misc device: creates /dev/column_decoder
*/
      ret = misc_register(&column_decoder_misc_device);

      /* Get the address of our registers from the device tree */
      ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
      if (ret) {
            ret = -ENOENT;
            goto out_deregister;
      }

      /* Make sure we can use these registers */
      if (request_mem_region(dev.res.start, resource_size(&dev.res),
                         DRIVER_NAME) == NULL) {
            ret = -EBUSY;
            goto out_deregister;
      }

      /* Arrange access to our registers */
      dev.virtbase = of_iomap(pdev->dev.of_node, 0);
      if (dev.virtbase == NULL) {
            ret = -ENOMEM;
            goto out_release_mem_region;
      }

      return 0;

out_release_mem_region:
      release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
      misc_deregister(&column_decoder_misc_device);
      return ret;
}
```

```
/* Clean-up code: release resources */
static int column_decoder_remove(struct platform_device *pdev)
{
      iounmap(dev.virtbase);
      release_mem_region(dev.res.start, resource_size(&dev.res));
      misc_deregister(&column_decoder_misc_device);
      return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id column_decoder_of_match[] = {
      { .compatible = "csee4840,column_decoder-1.0" },
      {},
};
MODULE_DEVICE_TABLE(of, column_decoder_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver column_decoder_driver = {
      .driver     = {
            .name = DRIVER_NAME,
            .owner      = THIS_MODULE,
            .of_match_table = of_match_ptr(column_decoder_of_match),
      },
      .remove     = __exit_p(column_decoder_remove),
};

/* Called when the module is loaded: set things up */
static int __init column_decoder_init(void)
{
      pr_info(DRIVER_NAME ": init\n");
      return platform_driver_probe(&column_decoder_driver,
column_decoder_probe);
}

/* Called when the module is unloaded: release resources */
static void __exit column_decoder_exit(void)
{
      platform_driver_unregister(&column_decoder_driver);
      pr_info(DRIVER_NAME ": exit\n");
}

module_init(column_decoder_init);
```

```
module_exit(column_decoder_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Adam Carpentieri, Columbia University");
MODULE_DESCRIPTION("Raycasting Column Decoder driver");
```

14.2.2 column_decoder.h

```
#ifndef _COLUMN_DECODER_H
#define _COLUMN_DECODER_H

#include <linux/ioctl.h>
#include <asm/types.h>

typedef struct {

  short top_of_wall; //signed
  __u8 wall_side;
  __u8 texture_type;

  __u16 wall_height;
  __u8 texture_offset;

} column_arg_t;

typedef struct {
      column_arg_t column_args[640];
} columns_t;

typedef struct {

  char char_val;

  __u8 row;
  __u8 col;

  __u8 highlight;

} char_tile_t;


#define COLUMN_DECODER_MAGIC 'q'
```

```
/* ioctls and their arguments */
#define COLUMN_DECODER_RESET_COL_NUM _IOW(COLUMN_DECODER_MAGIC, 1, __u8)
#define COLUMN_DECODER_WRITE_COLUMNS _IOW(COLUMN_DECODER_MAGIC, 2,
columns_t *)
#define COLUMN_DECODER_WRITE_CHAR _IOW(COLUMN_DECODER_MAGIC, 3, char_tile_t
*)
#define COLUMN_DECODER_BLACKOUT_SCREEN _IOW(COLUMN_DECODER_MAGIC, 4, __u8)
#define COLUMN_DECODER_REMOVE_BLACKOUT_SCREEN _IOW(COLUMN_DECODER_MAGIC, 5,
__u8)

#define COLUMN_DECODER_READ_VBLANK _IOR(COLUMN_DECODER_MAGIC, 6, __u8)


#endif
```

14.2.3 mazes.c

```
#include "mazes.h"

//types: B: bluestone, C: colorstone, E: eagle, G: greystone, M: mossy, P:
purplestone, R: redbrick, W: wood

maze_t mazes[3] = {
      { 12, 12, 144, "CESPR",
            {
                  G,G,G,G,G,G,G,G,G,G,G,G,
                  G,O,O,O,O,O,O,O,O,O,O,G,
                  G,O,O,O,O,O,O,O,O,O,O,G,
                  G,O,O,O,O,O,O,O,C,O,O,G,
                  G,O,O,C,O,C,O,O,C,O,O,G,
                  G,O,O,C,O,C,C,O,C,O,O,G,
                  G,O,O,C,O,O,C,O,C,O,O,G,
                  G,O,O,O,C,O,C,O,C,C,C,G,
                  G,O,O,O,C,O,C,O,O,O,O,G,
                  G,O,O,O,C,C,C,C,C,O,O,G,
                  G,O,O,O,O,O,O,O,O,O,O,E,
                  G,G,G,G,G,G,G,G,G,G,G,G
            }
      },
      { 24, 24, 576, "MUDD",
            {
```

```
         R,R,R,R,R,R,R,R,R,R,R,R,R,R,R,R,R,R,R,R,R,R,R,R,
         R,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,R,
         R,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,R,
         R,O,O,O,O,O,O,O,O,O,O,O,W,O,W,W,W,W,W,W,O,O,O,R,
         R,O,O,O,O,W,O,O,O,O,W,O,W,O,O,O,O,O,O,W,O,O,O,R,
         R,O,O,W,O,W,W,O,W,O,W,O,W,O,O,O,O,O,O,W,O,O,O,R,
         R,O,O,W,O,O,W,O,W,O,W,O,W,O,W,W,W,W,W,W,O,O,O,R,
         R,O,O,O,W,O,W,O,W,O,W,O,W,O,W,O,O,O,O,O,O,O,O,R,
         R,O,O,O,W,O,W,O,W,O,W,O,W,O,W,O,W,W,W,W,W,W,W,R,
         R,O,O,O,W,O,W,O,W,O,W,O,W,O,W,O,O,O,O,O,O,O,O,R,
         R,O,O,O,W,O,W,O,W,O,W,O,W,O,O,O,O,O,O,O,O,O,O,R,
         R,O,O,O,W,O,W,O,W,O,W,O,W,O,W,W,W,W,W,W,O,W,W,R,
         R,O,W,O,W,O,W,O,W,O,O,O,W,O,O,O,O,O,W,O,O,O,O,R,
         R,O,W,O,W,O,W,O,W,W,W,W,W,W,W,W,O,O,W,O,O,O,O,R,
         R,O,W,O,W,O,W,O,O,O,O,O,O,W,O,W,O,O,W,O,O,O,O,R,
         R,O,W,O,W,O,O,W,W,W,W,W,W,W,O,W,O,O,W,O,O,O,O,R,
         R,O,W,O,W,O,O,W,O,O,O,O,O,O,O,W,O,O,W,W,W,W,W,R,
         R,O,O,O,W,O,O,W,O,O,O,O,O,O,O,W,W,O,O,O,O,O,O,R,
         R,O,O,O,W,W,W,W,O,O,O,W,O,O,O,O,O,R,O,O,O,O,O,R,
         R,O,W,O,O,O,O,O,O,O,O,W,O,O,O,W,O,W,O,O,O,O,O,E,
         R,O,W,O,O,O,O,O,O,O,O,W,O,O,O,W,O,W,O,O,O,O,O,R,
         R,R,R,R,R,R,R,R,R,R,R,R,R,R,R,R,R,R,R,R,R,R,R,R
      }
   },
   { 36, 36, 1296, "PUPIN",

      {

B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,

B,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,B,

B,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,B,

B,O,O,O,O,O,O,O,O,O,O,O,M,O,M,M,M,M,M,M,O,O,O,O,O,O,M,M,M,M,M,M,M,M,O,B,

B,O,O,O,O,M,O,O,O,O,M,O,M,O,O,O,O,O,O,M,O,O,O,O,O,O,M,O,O,O,O,O,O,O,O,B,

B,O,O,M,O,M,M,O,M,O,M,O,M,O,O,O,O,O,O,M,O,O,O,O,O,O,M,O,M,M,M,M,M,M,M,B,

B,O,O,M,O,O,M,O,M,O,M,O,M,O,M,M,M,M,M,M,O,O,O,O,O,O,M,O,O,O,O,O,O,O,O,B,

B,O,O,O,M,O,M,O,M,O,M,O,M,O,M,O,O,O,O,O,O,O,O,O,O,O,M,O,O,O,O,O,O,O,O,B,
B,O,O,O,M,O,M,O,M,O,M,O,M,O,M,O,M,M,M,M,M,M,M,O,O,O,M,O,O,M,O,M,M,O,O,B,

B,O,O,O,M,O,M,O,M,O,M,O,M,O,M,O,O,O,O,O,O,O,O,O,O,O,M,O,O,M,O,O,M,O,O,B,

B,O,O,O,M,O,M,O,M,O,M,O,M,O,O,O,O,O,O,O,O,O,O,O,O,O,M,O,O,M,O,O,M,O,O,B,

B,O,O,O,M,O,M,O,M,O,M,O,M,O,M,M,M,M,M,M,O,M,M,M,M,M,M,O,O,M,O,O,M,O,O,B,

B,O,M,O,M,O,M,O,M,O,O,O,M,O,O,O,O,O,M,O,O,O,O,O,O,O,M,O,O,M,O,O,M,O,O,B,

B,O,M,O,M,O,M,O,M,M,M,M,M,M,M,M,O,O,M,O,O,O,O,O,O,O,M,O,O,M,O,O,M,O,O,B,

B,O,M,O,M,O,M,O,O,O,O,O,O,M,O,M,O,O,M,O,O,O,O,O,O,O,M,O,O,M,O,O,M,O,O,B,

B,O,M,O,M,O,O,M,M,M,M,M,M,M,O,M,O,O,M,O,O,O,O,O,O,O,M,O,O,M,O,O,M,O,O,B,

B,O,M,O,M,O,O,M,O,O,O,O,O,O,O,M,O,O,M,M,M,M,M,O,O,O,M,O,O,M,O,O,M,O,O,B,

B,O,O,O,M,O,O,M,O,O,O,O,O,O,O,M,M,O,O,O,O,O,O,O,O,O,M,O,O,O,M,O,M,O,O,B,

B,O,O,O,M,M,M,M,O,O,O,M,O,O,O,O,O,M,O,O,O,O,O,O,O,O,M,O,O,O,M,O,M,O,O,B,

B,O,M,O,O,O,O,O,O,O,O,M,O,O,O,M,O,M,O,O,O,O,O,O,O,O,M,O,O,O,M,O,M,M,M,B,

B,O,M,O,O,O,O,O,O,O,O,M,O,O,O,M,O,M,O,O,O,O,M,M,M,M,M,O,O,M,O,O,O,O,O,B,

B,M,M,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,O,M,O,O,O,O,O,O,M,O,O,O,O,O,B,

B,O,O,O,M,O,M,O,M,O,M,O,M,O,O,O,O,O,O,O,O,O,O,O,O,M,M,M,M,M,O,O,M,O,O,B,

B,O,O,O,M,O,M,O,M,O,M,O,M,O,M,M,M,M,M,M,O,M,M,O,O,M,O,O,O,O,O,O,M,O,O,B,

B,O,M,O,M,O,M,O,M,O,O,O,M,O,O,O,O,O,M,O,O,O,O,O,O,O,O,O,O,O,O,O,M,O,O,B,

B,O,M,O,M,O,M,O,M,M,M,M,M,M,M,M,O,O,M,O,O,O,O,O,O,O,O,O,O,O,O,O,M,O,O,B,

B,O,M,O,M,O,M,O,O,O,O,O,O,M,O,M,O,O,M,O,O,O,O,O,O,M,M,M,M,M,M,M,M,M,M,B,

B,O,M,O,M,O,O,M,M,M,M,M,M,M,O,M,O,O,M,O,O,O,O,O,O,M,O,O,O,O,O,O,O,O,O,B,

B,O,M,O,M,O,O,M,O,O,O,O,O,O,O,M,O,O,O,M,M,M,M,M,M,M,O,O,O,O,O,O,O,O,O,B,

B,O,O,O,M,O,O,M,O,O,O,O,O,O,O,M,M,O,O,O,O,O,O,O,O,O,M,O,O,O,O,O,O,O,O,B,
B,O,O,O,M,M,M,M,O,O,O,M,O,O,O,O,O,M,O,O,O,O,O,O,O,O,O,O,M,M,M,O,O,O,O,B,

B,O,M,O,O,O,O,O,O,O,O,M,O,O,O,M,O,M,O,O,O,O,O,O,M,O,O,O,O,O,M,O,O,O,O,B,

B,O,M,O,O,O,O,O,O,O,O,M,O,O,O,M,O,M,O,O,O,O,O,O,M,O,O,O,O,O,M,O,O,O,O,E,

B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B
      }
   }

};
```

14.2.4 mazes.h

```
#ifndef _MAZES_H

#define NUM_MAZES 3

//textures
#define B 1 // bluestone
#define C 2 // colorstone
#define E 3 // eagle (end level)
#define G 4 // greystone
#define M 5 // mossy
#define P 6 // purplestone
#define R 7 // redbrick
#define W 8 // wood

#define O 0 // opening (floor)

typedef struct {
   int width;
   int height;
   int area;
   char name[20];
   short map[1296]; //make this largest area of any map
} maze_t;

#endif
```

14.2.1 usb_devices.c

```
#include "usbdevices.h"

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define IDVENDOR 0x0079

/* References on libusb 1.0 and the USB HID/keyboard protocol
 *
 * http://libusb.org
 *
http://www.dreamincode.net/forums/topic/148707-introduction-to-using-libusb
-10/
 * http://www.usb.org/developers/devclass_docs/HID1_11.pdf
 * http://www.usb.org/developers/devclass_docs/Hut1_11.pdf
 */

/*
 * Find and return a USB keyboard device or NULL if not found
 * The argument con
 *
 */
struct libusb_device_handle *openkeyboard(uint8_t *endpoint_address) {

  libusb_device **devs;
  struct libusb_device_handle *keyboard = NULL;
  struct libusb_device_descriptor desc;
  ssize_t num_devs, d;
  uint8_t i, k;

  /* Start the library */
  if ( libusb_init(NULL) < 0 ) {
    fprintf(stderr, "Error: libusb_init failed\n");
    exit(1);
  }

  /* Enumerate all the attached USB devices */
  if ( (num_devs = libusb_get_device_list(NULL, &devs)) < 0 ) {
    fprintf(stderr, "Error: libusb_get_device_list failed\n");
    exit(1);
  }

  /* Look at each device, remembering the first HID device that speaks
     the keyboard protocol */

  for (d = 0 ; d < num_devs ; d++) {

    libusb_device *dev = devs[d];

      if ( libusb_get_device_descriptor(dev, &desc) < 0 ) {
      fprintf(stderr, "Error: libusb_get_device_descriptor failed\n");
      exit(1);
    }

    if (desc.bDeviceClass == LIBUSB_CLASS_PER_INTERFACE) {

        struct libusb_config_descriptor *config;
      libusb_get_config_descriptor(dev, 0, &config);

        for (i = 0 ; i < config->bNumInterfaces ; i++)


      for ( k = 0 ; k < config->interface[i].num_altsetting ; k++ ) {

        const struct libusb_interface_descriptor *inter =
config->interface[i].altsetting + k ;

        if ( inter->bInterfaceClass == LIBUSB_CLASS_HID &&
inter->bInterfaceProtocol == USB_HID_KEYBOARD_PROTOCOL) {

          int r;

          if ((r = libusb_open(dev, &keyboard)) != 0) {
            fprintf(stderr, "Error: libusb_open failed: %d\n", r);
            exit(1);
          }

          if (libusb_kernel_driver_active(keyboard,i))
            libusb_detach_kernel_driver(keyboard, i);

          libusb_set_auto_detach_kernel_driver(keyboard, i);

          if ((r = libusb_claim_interface(keyboard, i)) != 0) {
            fprintf(stderr, "Error: libusb_claim_interface failed: %d\n",
r);
            exit(1);
          }

          *endpoint_address = inter->endpoint[0].bEndpointAddress;
          goto found;

        }
      }
    }
  }

 found:
  libusb_free_device_list(devs, 1);

  return keyboard;
}

struct libusb_device_handle *opencontroller(uint8_t *endpoint_address) {

  libusb_device **devs;
  //struct libusb_device_handle *keyboard = NULL;
  struct libusb_device_handle *controller = NULL;
  struct libusb_device_descriptor desc;
  ssize_t num_devs, d;
  uint8_t i, k;

  /* Start the library */
  if ( libusb_init(NULL) < 0 ) {
    fprintf(stderr, "Error: libusb_init failed\n");
    exit(1);
  }

  /* Enumerate all the attached USB devices */
  if ( (num_devs = libusb_get_device_list(NULL, &devs)) < 0 ) {
    fprintf(stderr, "Error: libusb_get_device_list failed\n");
    exit(1);
  }

  /* Look at each device, remembering the first HID device that speaks
     the keyboard protocol */

  for (d = 0 ; d < num_devs ; d++) {
    libusb_device *dev = devs[d];
    if ( libusb_get_device_descriptor(dev, &desc) < 0 ) {
      fprintf(stderr, "Error: libusb_get_device_descriptor failed\n");
      exit(1);
    }

    /*
     * libusb_class_code enum
     * LIBUSB_CLASS_PER_INTERFACE = 0x00
     * others.. CLASS_AUDIO, PRINTER, SECURITY, VIDEO...
     * keyboard and controller: bDeviceClass            0 (Defined at
Interface level)
    */

    /*temporary WA using the idVendor of device descriptor since keyboard
has an overlapping interface with controller*/
    if (desc.bDeviceClass == LIBUSB_CLASS_PER_INTERFACE && desc.idVendor ==
IDVENDOR) {

      struct libusb_config_descriptor *config;
      libusb_get_config_descriptor(dev, 0, &config);

      for (i = 0 ; i < config->bNumInterfaces ; i++)

      for ( k = 0 ; k < config->interface[i].num_altsetting ; k++ ) {

        /* libusb_interface pointer array is part of the config_descriptor
struct
         * there are bNumInterfaces # of these libusb_interface pointers
         * altsetting is a pointer to libusb_interface_descriptor. there is
an array of them in libusb_interface struct
         * */
        const struct libusb_interface_descriptor *inter =
          config->interface[i].altsetting + k ;

        /*
         * keyboard:
         * bInterfaceClass         3 Human Interface Device
         * bInterfaceSubClass      1 Boot Interface Subclass
         * bInterfaceProtocol      1 Keyboard
         *
         *
         * controller:
         * bInterfaceClass         3 Human Interface Device
         * bInterfaceSubClass      0 No Subclass
         * bInterfaceProtocol      0 None
         *
         * LIBUSB_CLAS_HID = 3
         * USB_HID_KEYBOARD_PROTOCOL = 1
         * USB_HID_CONTROLLER_PROTOCOL = 0
         * */

        if ( inter->bInterfaceClass == LIBUSB_CLASS_HID &&
             inter->bInterfaceSubClass ==  0 &&
             inter->bInterfaceProtocol == USB_HID_CONTROLLER_PROTOCOL) {

                int r;

                if ((r = libusb_open(dev, &controller)) != 0) {
                  fprintf(stderr, "Error: libusb_open failed: %d\n", r);
                  exit(1);
                }

                if (libusb_kernel_driver_active(controller,i))
                  libusb_detach_kernel_driver(controller, i);

                libusb_set_auto_detach_kernel_driver(controller, i);

                if ((r = libusb_claim_interface(controller, i)) != 0) {
                  fprintf(stderr, "Error: libusb_claim_interface failed:
%d\n", r);

                  exit(1);
                }

                *endpoint_address = inter->endpoint[0].bEndpointAddress;
                goto found;
        }
      }
    }
  }

 //this is required when you are done with the devices. but make sure not
to unreference the device you are about to open before actually opening it
 found:
  libusb_free_device_list(devs, 1);

  //return keyboard;
  return controller;
```

```
}

char get_char_from_keystate(struct usb_keyboard_packet *packet) {

      uint8_t keycode = get_last_keycode(packet->keycode);
      uint8_t modifiers = packet->modifiers;

      char ascii_char;

      bool shift_pressed = modifiers & USB_LSHIFT || modifiers &
USB_RSHIFT;

      char no_shift_chars[] = {
            '\0', '\0', '\0', '\0', 'a', 'b', 'c', 'd', 'e', 'f',
            'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
            'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z',
            '1', '2', '3', '4', '5', '6', '7', '8', '9', '0',
            '\0', '\0', '\0', '\0', ' ', '-', '=', '[', ']', '\\',
            '\0', ';', '\'', '`', ',', '.', '/', '\0', '\0', '\0',
            '\0', '\0', '\0', '\0', '\0', '\0', '\0', '\0', '\0', '\0',
            '\0', '\0', '\0', '\0', '\0', '\0', '\0', '\0', '\0', '\0',
            '\0', '\0', '\0', '\0', '/', '*', '-', '+', '\0', '1',
            '2', '3', '4', '5', '6', '7', '8', '9', '0', '.',
            '\0', '\0', '\0', '='
      };

      char shift_chars[] = {
            '\0', '\0', '\0', '\0', 'A', 'B', 'C', 'D', 'E', 'F',
            'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P',
            'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z',
            '!', '@', '#', '$', '%', '^', '&', '*', '(', ')',
            '\0', '\0', '\0', '\0', ' ', '_', '+', '{', '}', '|',
            '\0', ':', '"', '~', '<', '>', '?', '\0', '\0', '\0',
            '\0', '\0', '\0', '\0', '\0', '\0', '\0', '\0', '\0', '\0',
            '\0', '\0', '\0', '\0', '\0', '\0', '\0', '\0', '\0', '\0',
            '\0', '\0', '\0', '\0', '/', '*', '-', '+', '\0', '1',
            '\0', '\0', '\0', '5', '\0', '\0', '\0', '\0', '\0', '\0',
            '\0', '\0', '\0', '='
      };

      ascii_char = keycode >= 0x67 ? '\0' :
            shift_pressed ? shift_chars[keycode] : no_shift_chars[keycode];

      return ascii_char;
```

```
}

bool is_controller_key_pressed(int x, uint8_t key, uint8_t keycode[6]) {

      if(keycode[x] == key)
            return true;

      return false;
}

bool is_key_pressed(uint8_t key, uint8_t keycode[6]) {

      if(!*keycode)
            return 0;

      for(int i=0; i<6; i++) {

            if(keycode[i] == key)
                  return true;
      }

      return false;
}

char get_gameplay_key(uint8_t keycode) {

    switch(keycode) {

        case 0x4F:
            return 'R';

        case 0x50:
            return 'L';

        case 0x51:
            return 'D';

        case 0x52:
            return 'U';
    }

      return '\0';
}
```

```
int get_last_keycode_pos(uint8_t keycode[6]) {

      if(!*keycode)
            return 0;

      for(int i=1; i<6; i++) {

            if(!keycode[i])
                  return i-1;
      }

      return 5;
}

uint8_t get_last_keycode(uint8_t keycode[6]) {

      if(!*keycode)
            return 0;

      uint8_t last_keycode = keycode[0];

      for(int i=1; i<6; i++) {

            if(!keycode[i])
                  return last_keycode;

            last_keycode = keycode[i];
      }

      return last_keycode;
}
```

14.2.1 usb_devices.h

```
#ifndef _USBDEVICES_H
#define _USBDEVICES_H

#include <libusb-1.0/libusb.h>
#include <stdbool.h>

#define USB_HID_KEYBOARD_PROTOCOL 1
#define USB_HID_CONTROLLER_PROTOCOL 0
```

```
/* Modifier bits */
#define USB_LCTRL  (1 << 0)
#define USB_LSHIFT (1 << 1)
#define USB_LALT   (1 << 2)
#define USB_LGUI   (1 << 3)
#define USB_RCTRL  (1 << 4)
#define USB_RSHIFT (1 << 5)
#define USB_RALT   (1 << 6)
#define USB_RGUI   (1 << 7)

struct usb_keyboard_packet {
  uint8_t modifiers;
  uint8_t reserved;
  uint8_t keycode[6];
};

/* Find and open a USB keyboard device.  Argument should point to
   space to store an endpoint address.  Returns NULL if no keyboard
   device was found. */
extern struct libusb_device_handle *openkeyboard(uint8_t *);
extern struct libusb_device_handle *opencontroller(uint8_t *);

extern char get_char_from_keystate(struct usb_keyboard_packet *);
extern char get_gameplay_key(uint8_t);
extern bool is_key_pressed(uint8_t, uint8_t[6]);
extern bool is_controller_key_pressed(int, uint8_t, uint8_t[6]);

extern uint8_t get_last_keycode(uint8_t[6]);
extern int get_last_keycode_pos(uint8_t[6]);

#endif
```

14.2.1 raycaster.c

```
/*
Adapted from https://permadi.com/activity/ray-casting-game-engine-demo/

2022 Adam Carpentieri (ac4409@columbia.edu) and Souryadeep Sen
(ss6400@columbia.edu)
*/
```

```c
#include "column_decoder.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "usbdevices.h"
#include "mazes.h"
#include <pthread.h>
#include <stdbool.h>
#include <math.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdint.h>

// size of tile (wall height) - best to make some power of 2
#define TILE_SIZE 64
#define WALL_HEIGHT 64

//do not change - hardware precisely calibrated for these values - will
crash system otherwise
#define PROJECTIONPLANEWIDTH 640
#define PROJECTIONPLANEHEIGHT 480

#define ANGLE60 PROJECTIONPLANEWIDTH
#define ANGLE30 (ANGLE60/2)
#define ANGLE15 (ANGLE30/2)
#define ANGLE90 (ANGLE30*3)
#define ANGLE180 (ANGLE90*2)
#define ANGLE270 (ANGLE90*3)
#define ANGLE360 (ANGLE60*6)
#define ANGLE0 0
#define ANGLE5 (ANGLE30/6)
#define ANGLE10 (ANGLE5*2)

//best to make this some power of 2 (with column decoder MUST be 1)
#define COLUMN_WIDTH 1

#define CONTROLLER_DPAD_DEFAULT 0x7F
#define CONTROLLER_BTN_DEFAULT 0xF

#define CHAR_NUM_ROWS 30
#define CHAR_NUM_COLS 80
```

```
//where the game will drop you on the map (FIXME make part of maze_t spec
for per-map customization)
#define STARTING_POINT_X 100
#define STARTING_POINT_Y 160

columns_t columns;

// precomputed trigonometric tables
float fSinTable[ANGLE360+1];
float fISinTable[ANGLE360+1];
float fCosTable[ANGLE360+1];
float fICosTable[ANGLE360+1];
float fTanTable[ANGLE360+1];
float fITanTable[ANGLE360+1];
float fFishTable[ANGLE360+1];
float fXStepTable[ANGLE360+1];
float fYStepTable[ANGLE360+1];

// player's attributes
int fPlayerX = STARTING_POINT_X; int tmpPlayerX; int storedPlayerX;
int fPlayerY = STARTING_POINT_Y; int tmpPlayerY; int storedPlayerY;
int fPlayerArc = ANGLE0; int storedPlayerArc;
int fPlayerDistanceToTheProjectionPlane = 677;
int fPlayerSpeed = 6;
int fProjectionPlaneYCenter = PROJECTIONPLANEHEIGHT / 2;

// movement flag
bool fKeyUp = false;
bool fKeyDown = false;
bool fKeyLeft = false;
bool fKeyRight = false;

struct libusb_device_handle *keyboard;
struct libusb_device_handle *controller;
uint8_t endpoint_address_kb;
uint8_t endpoint_address_ctr;

//pthread_mutex_t kp_mutex = PTHREAD_MUTEX_INITIALIZER;
//pthread_cond_t kp_cond = PTHREAD_COND_INITIALIZER;

//function signatures
void render();
void create_tables();
```

```
float arc_to_rad(float);
void handle_key_press(struct usb_keyboard_packet *, bool);
void put_char(char, uint8_t, uint8_t, uint8_t);
void put_string(const char *, uint8_t, uint8_t, uint8_t);
void clear_chars();
void set_blackout(bool);

void menu_select();
void finish_level();

bool up_pressed, down_pressed, left_pressed, right_pressed, enter_pressed;
bool up_ctr_pressed, down_ctr_pressed, left_ctr_pressed, right_ctr_pressed,
start_ctr_pressed;

bool jump_pressed, jump_pressed_ctr, is_jumping;
bool sched_jump_start;
int  jump_frame;

pthread_t keyboard_thread;
void *keyboard_thread_f(void *);

pthread_t controller_thread;
void *controller_thread_f(void *);

int column_decoder_fd;

//state for level completion / menu
int selected_maze = 0; int cur_selected_maze;
bool level_select_show = true;
bool level_finished = false;
bool menu_has_scrolled = false;
bool is_paused = false;
bool just_paused = false;
bool level_selected = false;

extern maze_t mazes[];

int main() {

    //TODO way to clear screen

    static const char filename[] = "/dev/column_decoder";

      printf("Raycaster Userspace program started\n");
      if ( (column_decoder_fd = open(filename, O_RDWR)) == -1) {

            fprintf(stderr, "could not open %s\n", filename);
            return -1;
      }

    create_tables();

    /* Open the keyboard */
    if ((keyboard = openkeyboard(&endpoint_address_kb)) == NULL)
        fprintf(stderr, "Did not find a keyboard\n");

      /* Open the controller */
    if ((controller = opencontroller(&endpoint_address_ctr)) == NULL)
        fprintf(stderr, "Did not find a controller\n");

      if(controller == NULL && keyboard == NULL)
            exit(1);

      //start keyboard and controller threads
      pthread_create(&keyboard_thread, NULL, keyboard_thread_f, NULL);
      pthread_create(&controller_thread, NULL, controller_thread_f, NULL);

      render();

      //reset column number in hardware for good measure
      if (ioctl(column_decoder_fd, COLUMN_DECODER_RESET_COL_NUM, 0x00)) {
            perror("ioctl(COLUMN_DECODER_RESET_COL_NUM) failed");
            exit(1);
      }

      set_blackout(false);
      clear_chars();

    while(true) {

            /*
            pthread_mutex_lock(&kp_mutex);
            while(!up_pressed && !down_pressed && !left_pressed &&
!right_pressed
                  && !up_ctr_pressed && !down_ctr_pressed &&
!left_ctr_pressed && !right_ctr_pressed) {
                  pthread_cond_wait(&kp_cond,&kp_mutex);
            }
            */

            if(!start_ctr_pressed && !enter_pressed) {

                  just_paused = false;
                  level_selected = false;
            }

            menu_select();

            if(!level_select_show && !level_finished) {

                  //pause menu
                  if((start_ctr_pressed || enter_pressed) && !is_paused &&
!level_selected) {

                        clear_chars();
                        is_paused = true;
                        level_select_show = true;
                        storedPlayerX = fPlayerX;
                        storedPlayerY = fPlayerY;
                        storedPlayerArc = fPlayerArc;
                        cur_selected_maze = selected_maze;
                        just_paused = true;
                  }

                  //jump
                  if(sched_jump_start) {

                        is_jumping = true;
                        jump_frame = 0;
                        sched_jump_start = false;
                  }

                  if(is_jumping) {

                        jump_frame++;

                        fProjectionPlaneYCenter = (int)
(PROJECTIONPLANEHEIGHT / 2) + (

                              (.1 * (jump_frame - 32) * (jump_frame - 32))
- 100
```

```
                        );

                        if(fProjectionPlaneYCenter >=
(PROJECTIONPLANEHEIGHT / 2)) {

                              fProjectionPlaneYCenter =
PROJECTIONPLANEHEIGHT / 2;
                              is_jumping = false;
                        }
                  }

                  // rotate left
                  if(left_pressed || left_ctr_pressed) {
                        if((fPlayerArc -= ANGLE5) < ANGLE0)
                              fPlayerArc += ANGLE360;
                  }

                  // rotate right
                  else if(right_pressed || right_ctr_pressed) {
                        if((fPlayerArc += ANGLE5) >= ANGLE360)
                              fPlayerArc -= ANGLE360;
                  }


                        //   _____     _
                        // |\ arc      |
                        // |  \        y
                        // |    \      |
                  //            -
                        // |--x--|
                        //
                        //  sin(arc)=y/diagonal
                        //  cos(arc)=x/diagonal   where diagonal=speed
                  float playerXDir = fCosTable[fPlayerArc];
                  float playerYDir = fSinTable[fPlayerArc];

                  // move forward
                  if(up_pressed || up_ctr_pressed) {

                        tmpPlayerX = fPlayerX + (int)(playerXDir *
fPlayerSpeed);
                        tmpPlayerY = fPlayerY + (int)(playerYDir *
fPlayerSpeed);

                        int map_index = (tmpPlayerX / TILE_SIZE) +
((tmpPlayerY / TILE_SIZE) * mazes[selected_maze].height);

                        if(map_index < mazes[selected_maze].area &&
!mazes[selected_maze].map[map_index]) {

                              fPlayerX = tmpPlayerX;
                              fPlayerY = tmpPlayerY;
                        }
                        else if(mazes[selected_maze].map[map_index] == E)
                              finish_level();
                  }

                  // move backward
                  else if(down_pressed || down_ctr_pressed) {

                        tmpPlayerX = fPlayerX - (int)(playerXDir *
fPlayerSpeed);
                        tmpPlayerY = fPlayerY - (int)(playerYDir *
fPlayerSpeed);

                        int map_index = (tmpPlayerX / TILE_SIZE) +
((tmpPlayerY / TILE_SIZE) * mazes[selected_maze].height);

                        if(map_index < mazes[selected_maze].area &&
!mazes[selected_maze].map[map_index]) {

                              fPlayerX = tmpPlayerX;
                              fPlayerY = tmpPlayerY;
                        }
                        else if(mazes[selected_maze].map[map_index] == E)
                              finish_level();
                  }
            }

            //pthread_mutex_unlock(&kp_mutex);

            //check current texture is eagle, then finish_level();


            render();
    }

      pthread_cancel(keyboard_thread);
    pthread_join(keyboard_thread, NULL);
```

```
      pthread_cancel(controller_thread);
    pthread_join(controller_thread, NULL);

    return 0;
}

void menu_select() { //will pick up delay inside render because being
called inside main loop

       if (level_select_show) {

            if(up_pressed || up_ctr_pressed) {

                  if(!menu_has_scrolled && --selected_maze < 0)
                        selected_maze = (NUM_MAZES - 1);

                  menu_has_scrolled = true;

                  if(is_paused) {
                        fPlayerX = selected_maze == cur_selected_maze ?
storedPlayerX : STARTING_POINT_X;
                        fPlayerY = selected_maze == cur_selected_maze ?
storedPlayerY : STARTING_POINT_Y;
                        fPlayerArc = selected_maze == cur_selected_maze ?
storedPlayerArc : ANGLE0;
                  }
            }
            else if(down_pressed || down_ctr_pressed) {

                  if(!menu_has_scrolled && ++selected_maze == NUM_MAZES)
                        selected_maze = false;

                  menu_has_scrolled = true;

                  if(is_paused) {
                        fPlayerX = selected_maze == cur_selected_maze ?
storedPlayerX : STARTING_POINT_X;
                        fPlayerY = selected_maze == cur_selected_maze ?
storedPlayerY : STARTING_POINT_Y;
                        fPlayerArc = selected_maze == cur_selected_maze ?
storedPlayerArc : ANGLE0;
                  }
            }
            else if ((enter_pressed || start_ctr_pressed) && !just_paused)
{

                  level_select_show = false;
                  menu_has_scrolled = false;
                  is_paused = false;
                  level_selected = true;
                  clear_chars();

                  put_string("Find the Eagle", 0, 33, 0);

                  return;
            }
            else
                  menu_has_scrolled = false;

            int start_row = 10;
            int col = 25;
            char selection_title[40];

            put_string((is_paused ? "Giving Up So Soon?" : "Choose Your
Nightmare..."), (start_row-2), col, 0);

            for(int i=0; i<NUM_MAZES; i++) {

                  if(is_paused && i == cur_selected_maze) {

                        strcpy(selection_title, "Back to ");

                        strcat(selection_title, mazes[i].name);
                  }
                  else
                        strcpy(selection_title, mazes[i].name);

                  put_string(selection_title, (start_row+i), col,
selected_maze==i);
            }
       }
}

void finish_level() {

      clear_chars();
      set_blackout(true);
      put_string("Great job, definitely getting an A", 14, 23, 0);

      sleep(5);

      fPlayerX = STARTING_POINT_X;
      fPlayerY = STARTING_POINT_Y;
      fPlayerArc = ANGLE0;

      clear_chars();
      set_blackout(false);

      level_select_show = true;
}

void *keyboard_thread_f(void *ignored) {

      int transferred;

      bool prev_jump_state;

      struct usb_keyboard_packet packet;

      while(true) {

            libusb_interrupt_transfer(keyboard, endpoint_address_kb,
                        (unsigned char *) &packet, sizeof(packet),
                        &transferred, 0);

        if (transferred == sizeof(packet)) {

                  prev_jump_state = jump_pressed;

                  //pthread_mutex_lock(&kp_mutex);

                  up_pressed =    is_key_pressed(0x52, packet.keycode);
                  down_pressed =  is_key_pressed(0x51, packet.keycode);
                  left_pressed =  is_key_pressed(0x50, packet.keycode);
                  right_pressed = is_key_pressed(0x4f, packet.keycode);
                  jump_pressed =  is_key_pressed(0x2c, packet.keycode) &&
!level_select_show; //spacebar
                  enter_pressed =  is_key_pressed(0x28, packet.keycode);

                  if(jump_pressed && !is_jumping && !prev_jump_state)
```

```
                        sched_jump_start = true;

                  //pthread_cond_signal(&kp_cond);
                  //pthread_mutex_unlock(&kp_mutex);
            }
      }

      return NULL;
}

void *controller_thread_f(void *ignored) {

      int transferred;

      bool prev_jump_state_ctr;

      struct usb_keyboard_packet packet;

      while(true) {

            libusb_interrupt_transfer(controller, endpoint_address_ctr,
                        (unsigned char *) &packet, sizeof(packet),
                        &transferred, 0);

        if (transferred == sizeof(packet)) {

                  prev_jump_state_ctr = jump_pressed_ctr;

                  //pthread_mutex_lock(&kp_mutex);

                  if (packet.keycode[1] != CONTROLLER_DPAD_DEFAULT ||
packet.keycode[2] != CONTROLLER_DPAD_DEFAULT || packet.keycode[2] !=
CONTROLLER_BTN_DEFAULT || packet.keycode[4] != 0) {

                        up_ctr_pressed          =
is_controller_key_pressed(2, 0x00, packet.keycode);
                        down_ctr_pressed  = is_controller_key_pressed(2,
0xff, packet.keycode);
                        left_ctr_pressed  = is_controller_key_pressed(1,
0x00, packet.keycode);
                        right_ctr_pressed       =
is_controller_key_pressed(1, 0xff, packet.keycode);
                        jump_pressed_ctr  = is_controller_key_pressed(3,
0x2f, packet.keycode) && !level_select_show; //a btn
```

```
                        start_ctr_pressed   = is_controller_key_pressed(4,
0x20, packet.keycode);

                        if(jump_pressed_ctr && !is_jumping &&
!prev_jump_state_ctr)
                              sched_jump_start = true;
                  }
                  else {
                        up_ctr_pressed        = false;
                        down_ctr_pressed  = false;
                        left_ctr_pressed  = false;
                        right_ctr_pressed       = false;
                        jump_pressed_ctr  = false;
                        start_ctr_pressed       = false;
                  }

                  //pthread_cond_signal(&kp_cond);
                  //pthread_mutex_unlock(&kp_mutex);
            }
      }

      return NULL;
}

void render() {

    int verticalGrid;        // horizotal or vertical coordinate of
intersection
    int horizontalGrid;      // theoritically, this will be multiple of
TILE_SIZE
                               // , but some trick did here might cause
                               // the values off by 1
    int distToNextVerticalGrid; // how far to the next bound (this is
multiple of
    int distToNextHorizontalGrid; // tile size)
    float xIntersection;  // x and y intersections
    float yIntersection;
    float distToNextXIntersection;
    float distToNextYIntersection;

      uint8_t textureH, textureV, texture;
      uint8_t vblank = 0;

    int xGridIndex;        // the current cell that the ray is in
    int yGridIndex;

    float distToVerticalGridBeingHit;      // the distance of the x and y
ray intersections from
    float distToHorizontalGridBeingHit;      // the viewpoint

    int castArc, castColumn;

    castArc = fPlayerArc;
        // field of view is 60 degree with the point of view (player's
direction in the middle)
        // 30  30
        //    ^
        //  \ | /
        //   \|/
        //    v
        // we will trace the rays starting from the leftmost ray
    castArc -= ANGLE30;
        // wrap around if necessary
    if(castArc < 0)
        castArc = ANGLE360 + castArc;

    for(castColumn=0; castColumn < PROJECTIONPLANEWIDTH; castColumn +=
COLUMN_WIDTH) {

        // ray is between 0 to 180 degree (1st and 2nd quadrant)
        // ray is facing down
        if(castArc > ANGLE0 && castArc < ANGLE180) {
                // truncate then add to get the coordinate of the FIRST
grid (horizontal
                // wall) that is in front of the player (this is in pixel
unit)
                // ROUND DOWN
            horizontalGrid = (fPlayerY / TILE_SIZE) * TILE_SIZE +
TILE_SIZE;

            // compute distance to the next horizontal wall
            distToNextHorizontalGrid = TILE_SIZE;

            float xtemp = fITanTable[castArc] * (horizontalGrid -
fPlayerY);
                    // we can get the vertical distance to that wall by
                    // (horizontalGrid-GLplayerY)
                    // we can get the horizontal distance to that wall by
                    // 1/tan(arc)*verticalDistance
                    // find the x interception to that wall
            xIntersection = xtemp + fPlayerX;
        }

        // else, the ray is facing up
        else {

            horizontalGrid = (fPlayerY / TILE_SIZE) * TILE_SIZE;
            distToNextHorizontalGrid = -TILE_SIZE;

            float xtemp = fITanTable[castArc] * (horizontalGrid -
fPlayerY);
            xIntersection = xtemp + fPlayerX;

            horizontalGrid--;
        }

        // LOOK FOR HORIZONTAL WALL
        if(castArc == ANGLE0 || castArc == ANGLE180)
            distToHorizontalGridBeingHit=__FLT_MAX__;//Float.MAX_VALUE;

        // else, move the ray until it hits a horizontal wall
        else {
            distToNextXIntersection = fXStepTable[castArc];

            while(true) {

                xGridIndex = (int)(xIntersection / TILE_SIZE);
                // in the picture, yGridIndex will be 1
                yGridIndex = (horizontalGrid / TILE_SIZE);

                if((xGridIndex >= mazes[selected_maze].width) ||
                    (yGridIndex >= mazes[selected_maze].height) ||
                    xGridIndex < 0 || yGridIndex < 0) {

                    distToHorizontalGridBeingHit = __FLT_MAX__;
                    break;
                }
                else if (mazes[selected_maze].map[yGridIndex *
mazes[selected_maze].width + xGridIndex]) {

                              textureH =
mazes[selected_maze].map[yGridIndex * mazes[selected_maze].width +
xGridIndex] - 1;
                    distToHorizontalGridBeingHit  =
(xIntersection-fPlayerX)*fICosTable[castArc];
                    break;
                }
                // else, the ray is not blocked, extend to the next block
                else {
                    xIntersection += distToNextXIntersection;
                    horizontalGrid += distToNextHorizontalGrid;
                }
            }
        }

        // FOLLOW X RAY
        if(castArc < ANGLE90 || castArc > ANGLE270) {

            verticalGrid = TILE_SIZE + (fPlayerX / TILE_SIZE) * TILE_SIZE;
            distToNextVerticalGrid = TILE_SIZE;

            float ytemp = fTanTable[castArc] * (verticalGrid - fPlayerX);
            yIntersection = ytemp + fPlayerY;
        }
        // RAY FACING LEFT
        else {

            verticalGrid = (fPlayerX/TILE_SIZE)*TILE_SIZE;
            distToNextVerticalGrid = -TILE_SIZE;

            float ytemp = fTanTable[castArc] * (verticalGrid - fPlayerX);
            yIntersection = ytemp + fPlayerY;

            verticalGrid--;
        }

        // LOOK FOR VERTICAL WALL
        if(castArc == ANGLE90 || castArc == ANGLE270)
            distToVerticalGridBeingHit = __FLT_MAX__;

        else {

            distToNextYIntersection = fYStepTable[castArc];

            while(true) {
                // compute current map position to inspect
                xGridIndex = (verticalGrid / TILE_SIZE);
                yGridIndex = (int)(yIntersection / TILE_SIZE);

                if ((xGridIndex >= mazes[selected_maze].width) ||
                    (yGridIndex >= mazes[selected_maze].height) ||
                    xGridIndex < 0 || yGridIndex < 0) {
                    distToVerticalGridBeingHit = __FLT_MAX__;
                    break;
                }
                else if (mazes[selected_maze].map[yGridIndex *
mazes[selected_maze].width + xGridIndex]) {

                              textureV =
mazes[selected_maze].map[yGridIndex * mazes[selected_maze].width +
xGridIndex] - 1;
                    distToVerticalGridBeingHit = (yIntersection - fPlayerY)
* fISinTable[castArc];
                    break;
                }
                else  {
                    yIntersection += distToNextYIntersection;
                    verticalGrid += distToNextVerticalGrid;
                }
            }
        }

        // DRAW THE WALL SLICE
        float dist;
        uint16_t topOfWall;   // used to compute the top and bottom of the
sliver that
        uint16_t bottomOfWall;   // will be the staring point of floor and
ceiling
            uint8_t wall_side; //0=x, 1=y
            uint8_t offset;

            // determine which ray strikes a closer wall.
            // if yray distance to the wall is closer, the yDistance will
be shorter than
                // the xDistance
        if (distToHorizontalGridBeingHit < distToVerticalGridBeingHit) {

            // the next function call (drawRayOnMap()) is not a part of
raycating rendering part,
            // it just draws the ray on the overhead map to illustrate the
```

```
raycasting process
            dist = distToHorizontalGridBeingHit;
                  texture = textureH;
                  wall_side = 0;
                  offset = (int)xIntersection % TILE_SIZE;
        }
        // else, we use xray instead (meaning the vertical wall is closer
than
        //   the horizontal wall)
        else {

            // the next function call (drawRayOnMap()) is not a part of
raycasting rendering part,
            // it just draws the ray on the overhead map to illustrate the
raycasting process
            dist = distToVerticalGridBeingHit;
                  texture = textureV;
                  wall_side = 1;
                  offset = (int)yIntersection % TILE_SIZE;
        }

        // correct distance (compensate for the fishbown effect)
        dist /= fFishTable[castColumn];

            //0 distance makes no sense for below calcs
            if(dist == 0.0)
                  dist = .00001;

        // projected_wall_height/wall_height =
fPlayerDistToProjectionPlane/dist;
        int projectedWallHeight = (int)(WALL_HEIGHT *
(float)fPlayerDistanceToTheProjectionPlane / dist);
            projectedWallHeight = projectedWallHeight > 32767 ? 32767 :
projectedWallHeight;


        bottomOfWall = fProjectionPlaneYCenter + (int)(projectedWallHeight
* 0.5F);
            bottomOfWall = bottomOfWall > 32767 ? 32767 : bottomOfWall;

        topOfWall = PROJECTIONPLANEHEIGHT - bottomOfWall;

        columns.column_args[castColumn].top_of_wall = topOfWall;
        columns.column_args[castColumn].wall_side = wall_side;
        columns.column_args[castColumn].texture_type = texture;
        columns.column_args[castColumn].wall_height =
(short)projectedWallHeight;
        columns.column_args[castColumn].texture_offset = offset;

        // TRACE THE NEXT RAY
        castArc += COLUMN_WIDTH;
        if (castArc >= ANGLE360)
            castArc -= ANGLE360;
    }

      //wait for vblank to send columns
      while(true) {

            ioctl(column_decoder_fd, COLUMN_DECODER_READ_VBLANK, &vblank);

            if(vblank)
                  break;

            usleep(50);
      }

    //send the columns to the driver
    if (ioctl(column_decoder_fd, COLUMN_DECODER_WRITE_COLUMNS, &columns)) {
            perror("ioctl(COLUMN_DECODER_WRITE_COLUMNS) failed");
            return;
      }
}

void create_tables() {

    int i;
    float radian;

    for (i=0; i <= ANGLE360; i++) {
        // get the radian value (the last addition is to avoid division by
0, try removing
            // that and you'll see a hole in the wall when a ray is at 0,
90, 180, or 270 degree)
        radian = arc_to_rad(i) + (float)(0.0001);
        fSinTable[i] = (float)sin(radian);
        fISinTable[i] = (1.0F / (fSinTable[i]));
        fCosTable[i] = (float)cos(radian);
        fICosTable[i] = (1.0F / (fCosTable[i]));
        fTanTable[i] = (float)tan(radian);
        fITanTable[i] = (1.0F / fTanTable[i]);

        //  you can see that the distance between xi is the same
        //  if we know the angle
        //  _____|_/next xi______________
        //       |
        //  ____/|next xi_________    slope = tan = height / dist between
xi's
        //     / |
        //  __/__|_________  dist between xi = height/tan where height=tile
size
        // old xi|
        //                  distance between xi = x_step[view_angle];
        //
        //
        // facine left
        // facing left
        if (i >= ANGLE90 && i < ANGLE270) {

            fXStepTable[i] = (float)(TILE_SIZE / fTanTable[i]);
            if (fXStepTable[i] > 0)
                fXStepTable[i] = -1 * fXStepTable[i];
        }
        // facing right
        else {

            fXStepTable[i] = (float)(TILE_SIZE / fTanTable[i]);
            if(fXStepTable[i] < 0)
                fXStepTable[i] = -1 * fXStepTable[i];
        }

        // FACING DOWN
        if (i >= ANGLE0 && i < ANGLE180) {

            fYStepTable[i] = (float)(TILE_SIZE * fTanTable[i]);
            if (fYStepTable[i] < 0)
                fYStepTable[i] = -1 * fYStepTable[i];
        }
        // FACING UP
        else {
            fYStepTable[i] = (float)(TILE_SIZE * fTanTable[i]);
            if (fYStepTable[i] > 0)
                fYStepTable[i] = -1 * fYStepTable[i];
        }
    }

    for (i = -ANGLE30; i <= ANGLE30; i++) {

        radian = arc_to_rad(i);
        // we don't have negative angle, so make it start at 0
        // this will give range 0 to 320
        fFishTable[i + ANGLE30] = (float)(1.0F / cos(radian));
    }
}

  //*******************************************************************//
//* Convert arc to radian
//*******************************************************************//
float arc_to_rad(float arc_angle) {
    return ((float)(arc_angle*M_PI)/(float)ANGLE180);
}

/*supplemental functions dealing with screen color override and text*/

void put_char(char c, uint8_t row, uint8_t col, uint8_t highlight) {

      if(row >= CHAR_NUM_ROWS || col >= CHAR_NUM_COLS)
        return;

      char_tile_t char_tile = { c, row, col, highlight };

      if (ioctl(column_decoder_fd, COLUMN_DECODER_WRITE_CHAR, &char_tile))
{
            perror("ioctl(COLUMN_DECODER_WRITE_CHAR) failed");
            exit(1);
      }
}


void clear_chars() {

      for(uint8_t row=0; row<CHAR_NUM_ROWS; row++) {
            for(uint8_t col=0; col<CHAR_NUM_COLS; col++)
                  put_char(' ', row, col, 0);
      }
}
```

```
void put_string(const char *s, uint8_t row, uint8_t col, uint8_t highlight)
{
  char c;

  if(row >= CHAR_NUM_ROWS)
        return;

  while ((c = *s++) != 0 && col < CHAR_NUM_COLS)
        put_char(c, row, col++, highlight);
}

void set_blackout(bool blackout) {

      if(blackout) {
            if (ioctl(column_decoder_fd, COLUMN_DECODER_BLACKOUT_SCREEN,
0x00)) {
                  perror("ioctl(COLUMN_DECODER_BLACKOUT_SCREEN) failed");
                  exit(1);
            }
      }
      else {
            if (ioctl(column_decoder_fd,
COLUMN_DECODER_REMOVE_BLACKOUT_SCREEN, 0x00)) {
                  perror("ioctl(COLUMN_DECODER_REMOVE_BLACKOUT_SCREEN)
failed");
                  exit(1);
            }
      }

}
```