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

FPGA Raycasting Spring 2022

Design Document

OVERVIEW

FPGA Raycasting is a project to implement raycasting techniques originally developed in the mid 1990's with games such as Wolfenstein and Doom, in FPGA hardware. The final project will be a maze type game. We will additionally demonstrate a maze auto-solver capability.

Our goal is to replicate the feel of https://js-dos.com/Wolf/.

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

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

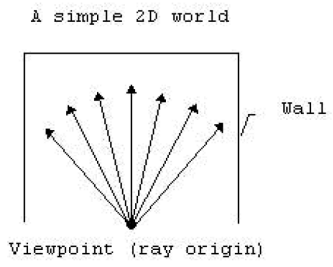
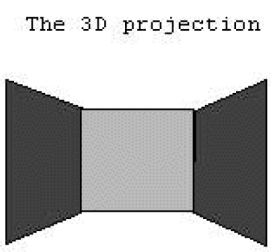
1. Walls are at 90 degree angle with the floor
2. Walls are made of cubes that have the same size
3. Floor is always flat

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.

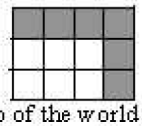
Some projection attributes defined:

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/ :

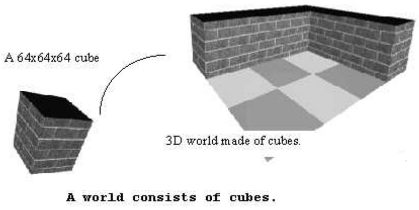
A simple 2D world:	**The 3D projection:**
	

2D grid:



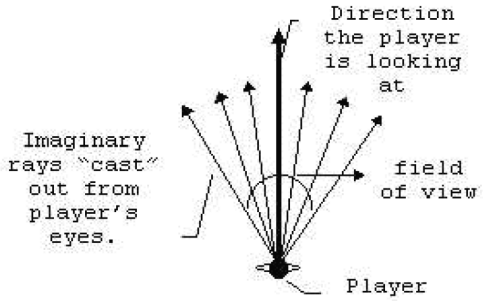
2D grid map of the world (each grid consists of 64x64 smaller units).

Corresponding 3D world:

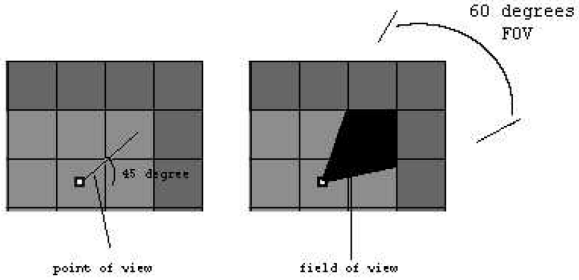


A world consists of cubes.

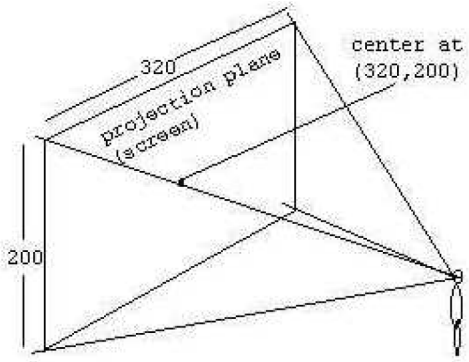
Field of View:



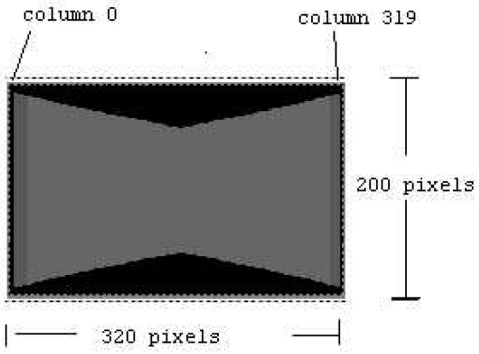
Field and point of view on the grid:



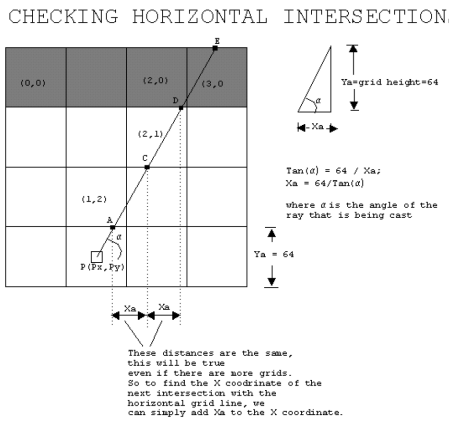
The projection plane:



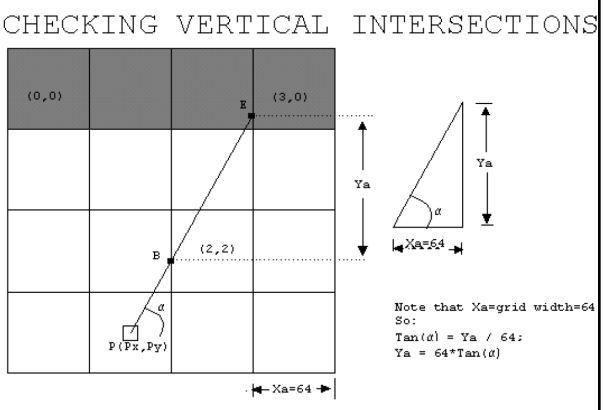
The ray cast on the screen (projection plane)



Horizontal intersection with walls on grid:



Vertical intersection with walls on grid:



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;

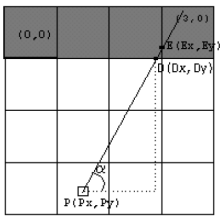
X intersection point:

- 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:



Two ways of finding distance:

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

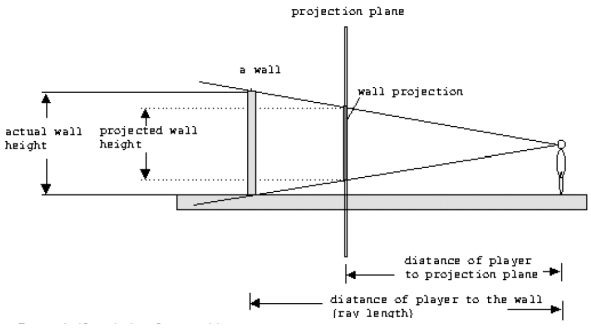
or

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

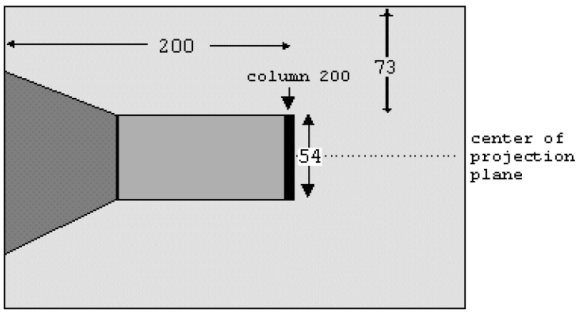
(where ABS=absolute value)

Height of the wall:

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



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



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.

COLUMN DECODER & DATA STRUCTURE

Design Assumptions

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

Specifics

For each pixel, we will grab the column tuple in SRAM corresponding to the row. Then we will calculate if the pixel is in the ceiling, wall, or floor. Floors and ceiling are hard coded with some RGB value and we may 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 30 bits:

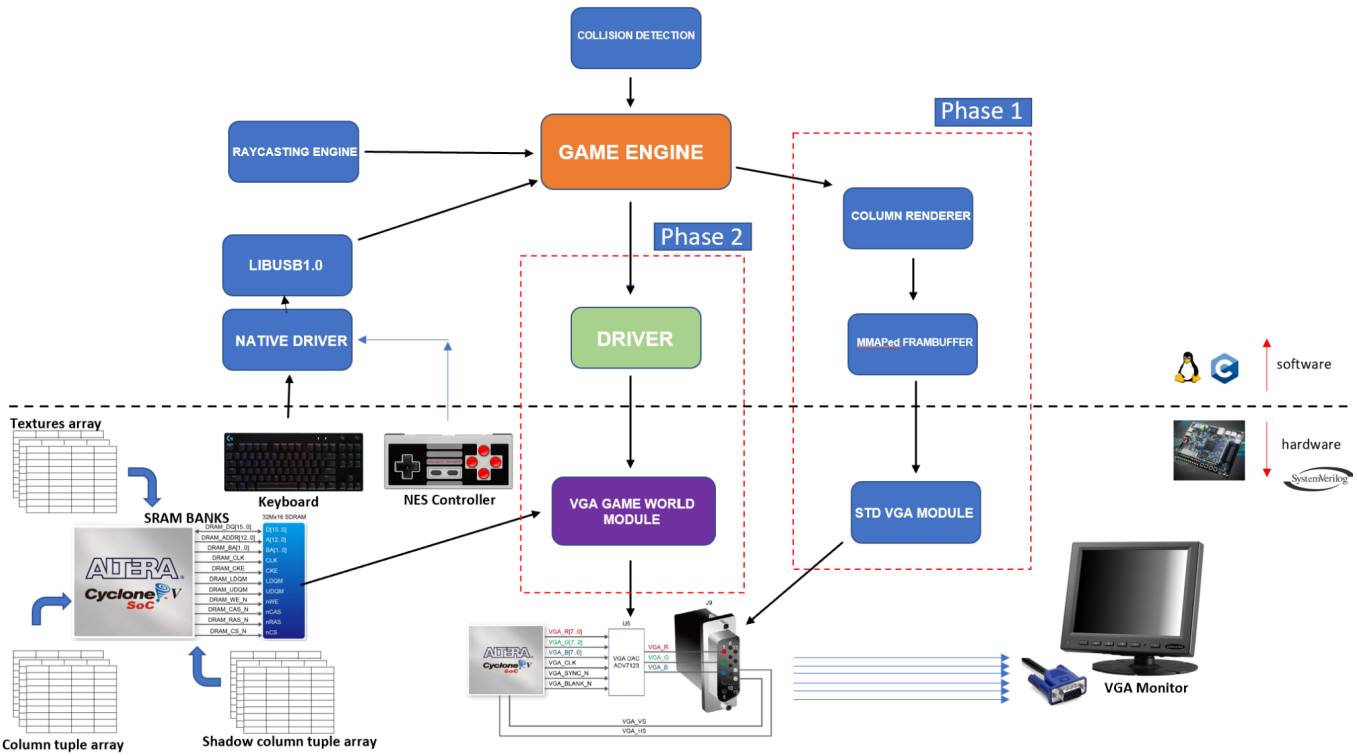
- Top of wall [10]: what row the ceiling ends and wall starts
- Projection wall height[10]: how many rows of pixels for the wall starting with the above value
- Wall side[1]: a single bit to say if the wall is along the x or y axis on the map. Using this data you can darken or lighten the wall to create 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.

Optimizations

The distinct challenge of designing an efficient rendering pipeline is that the vga signal is horizontal in nature, whereas our data structure is vertical in nature. Some possible ways around it are:

- Storing texture data with the x and y coordinates flipped
- Using a 90 degree rotated raycaster[1]
- Using bit shifting instead of other math where possible due to power-of-two sizing of elements

DESIGN BLOCK DIAGRAM



[1] https://lodev.org/cgtutor/raycasting.html, “Performance Considerations” Section

GRAPHICS

For texturing we are using the freely provided wolfenstein .png files available at https://lodev.org/cgtutor/files/wolftex.zip. Then we use the tool https://lvgl.io/tools/imageconverter to convert the file into an array of RGB values. They are 64x64 pixels which is the same dimensions of our chosen block size.

RESOURCE BUDGET

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

Name	Description	Size	Num Elements
Column Tuple	~~Column_num (9)~~, top_of_wall (10), projected_height (10), wall_side (1), texture_offset (6), texture_type (3)	30 bits	480
Texture	64x64 array of RGB values (alpha assumed to be 1)	64 * 64 * 3 bytes (12.288kb) each	8

Based on the above, our SRAM memory needs are modest, coming in at 98.304kb for textures + 1.8kb for column tuples.

We will not transmit the column number but instead auto-increment the column number in a sequential internal array since the column tuples will be pipelined, one after another.

HARDWARE SOFTWARE INTERFACE

We will employ a similar interface as Lab 3 with a linux driver (registering our VGA game world as a character device) responsible for updating all the tuples in a pipelined manner. There may be some details missing in this explanation, but there will be a protocol where we first request a write, then we wait until acknowledged. Then we pulse the values on the bus, one by one over each clock cycle. How this is done in software with the complexities of a scheduler does give some concern. It could be the case that the hardware can tolerate a missing value and wait for the next one, whenever that occurs.

From the hardware side, it will not send the acknowledgement bit back until it is in a VBLANK state. At that point we are safe to overwrite the tuple values. If we allowed the tuple values to be overwritten mid-frame it would cause screen tearing and / or other visual artifacts.

We may also simply have a shadow tuples array which can be written to at any time, and then it is simply copied to the real tuple array (if some flag is triggered) during VBLANK.

Below is a best guess interface, and we fully expect it will evolve over time with experimentation and feedback from our wise elders.

reg0	reg1	Reg2
1 bit write intent	1 bit write flag enabled / disabled	30 (00 padded 32 bit) column tuple to copy when populated

More research needs to be completed to figure out how to coordinate the pipelining of the data in a robust, reliable, and repeatable manner.

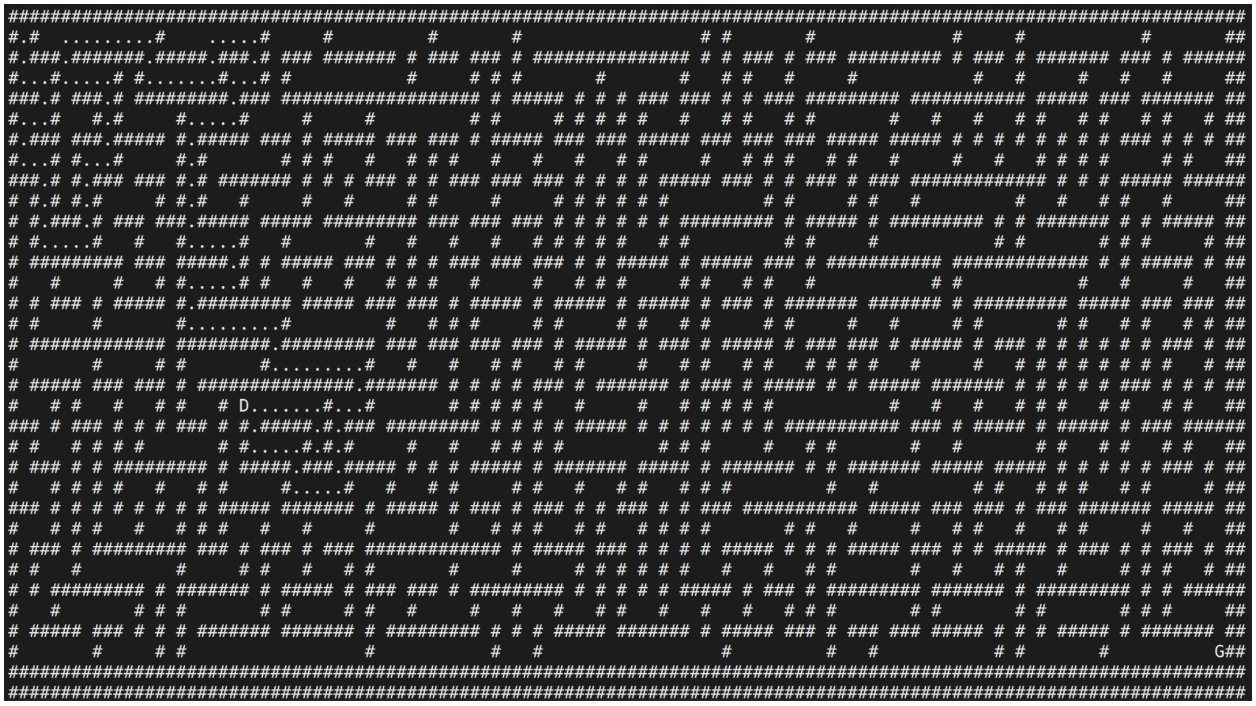
PLAYER INPUT

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

Initially we intend to use the keyboard to control the game since in fact this is what was originally used for the games.

MAZE AUTO-SOLVER

We intend to also show in the terminal a 2D representation of the maze that will be cast to the 3D world. **'D'** is our maze player, looking for gold **'G'.** Below is a pure software implementation.



MILESTONES

Milestone 1

(Completed) 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.

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.[2]

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

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

Milestone 3

Texturing in hardware. Maze auto solver. Basic gamification of maze's. Multiple levels. Game completion screen.

STRETCH GOALS

Let's be honest - these rarely get accomplished due to the competing time requirements of other classes. But you never know, maybe we'll be the first.

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.

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.

Music / Sound Effects

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

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.