

Spider Slayer: A FPGA Dungeon Crawler Report

Ash Hong

EECS Department

Massachusetts Institute of Technology

Cambridge, MA, USA

ashhong@mit.edu

Frank Kelley

EECS Department

Massachusetts Institute of Technology

Cambridge, MA, USA

fey@mit.edu

Zhiron Wu

EECS Department

Massachusetts Institute of Technology

Cambridge, MA, USA

zwuy@mit.edu

Abstract—Spider Slayer: A FPGA Dungeon Crawler aims to create a maneuverable, pseudo-3D “dungeon” where the player is able to explore rooms and defeat simple enemies through a first-person POV. The player will control the movement and actions of the character through a Wii remote-esque controller, with an accelerometer installed to simulate which direction/action to perform. This is then translated into a visual reference via HDMI where it displays the result. The main mechanics of this game would involve moving through the dungeon by changing rooms on-screen, and attacking simple enemies, such as spiders to align with the theme, by using a slashing or similar motion. The pseudo-3D dungeon would be created using raycasting, with priority granted towards creating low-poly rooms and reducing any latency [1], [2].

Index Terms—Field programmable gate arrays, Digital systems, Raycasting

I. INTRODUCTION

Within this paper, we will describe the design and implementation of Spider Slayer: A FPGA Dungeon Crawler. The inspiration of this game is derived from classic “dungeon crawler” RPGs, Fruit Ninja, and Wii games [1], [4]. Our primary mechanics focus on player movement, controller and FPGA communication for interactions, and rendering the environment around the player. The “slashing” motion from the controller should allow the FPGA to despawn any rendered spiders in front of the player, following Newtonian physics. We will utilize an accelerometer on the controller to track this specific kind of motion and ensure any non-slashing motions are not registered incorrectly as an attack. The FPGA will be reliable for implementing parallelization processes, such as ray-casting for rendering through processing parallel rays due to its hardware capabilities [1], [5].

The system will thus be demonstrated by the player utilizing an HDMI monitor and a physical controller to provide input and feedback to the game. The physical controller will contain buttons and an IMU to receive player inputs. The monitor will reflect any changes based on said input, as the FPGA constructs the visual output by calculating movement, rotation, and interactions for the player.

II. GAME SETUP

A. Representations

1) *Room*: Each room is represented as a 5x5 grid, with the walkable area (the interior of the room) seen as the inner 3x3 grid. Since we create two rooms at once to ease the transition



Fig. 1. An example of a dungeon crawler room from Hades

between rooms, we are working with a 5x9 grid total that includes two rooms (Fig 3).

To represent this on our board, we create a 5 by 9 grid register (i.e. `grid[i][j]` corresponds to the block at row `i` and column `j`). Each `grid[i][j]` then has a 4-bit value that describes its block type (Table I). A flat, unfolded version of the map can also be easily created by flattening the array into a list. This allows us to access two representations of the map as needed.

MSB	LSB	Description
00	-	Empty
01	00	Door: Open
01	01	Door: Closed
01	11	Door: Locked
10	-	Spider
11	-	Wall

TABLE I
4-BIT TO BLOCK TYPE

The rooms will be generated through our Room Generator module, which incorporates an element of pseudo-randomness to ensure room layouts have variability, just like dungeon crawler games (Fig 2).

2) *Player*: For both player location and direction, we represent each `x` and `y` axis using fixed point Q4.4 notation, where each value involves a signed bit, 3 bits to represent integers, and 4 bits to represent the decimal part. Location (0.0, 0.0) corresponds to the top left corner (thus, we do not expect to have negative locations unless out of bounds). This is helpful in that it allows us to not only easily perform

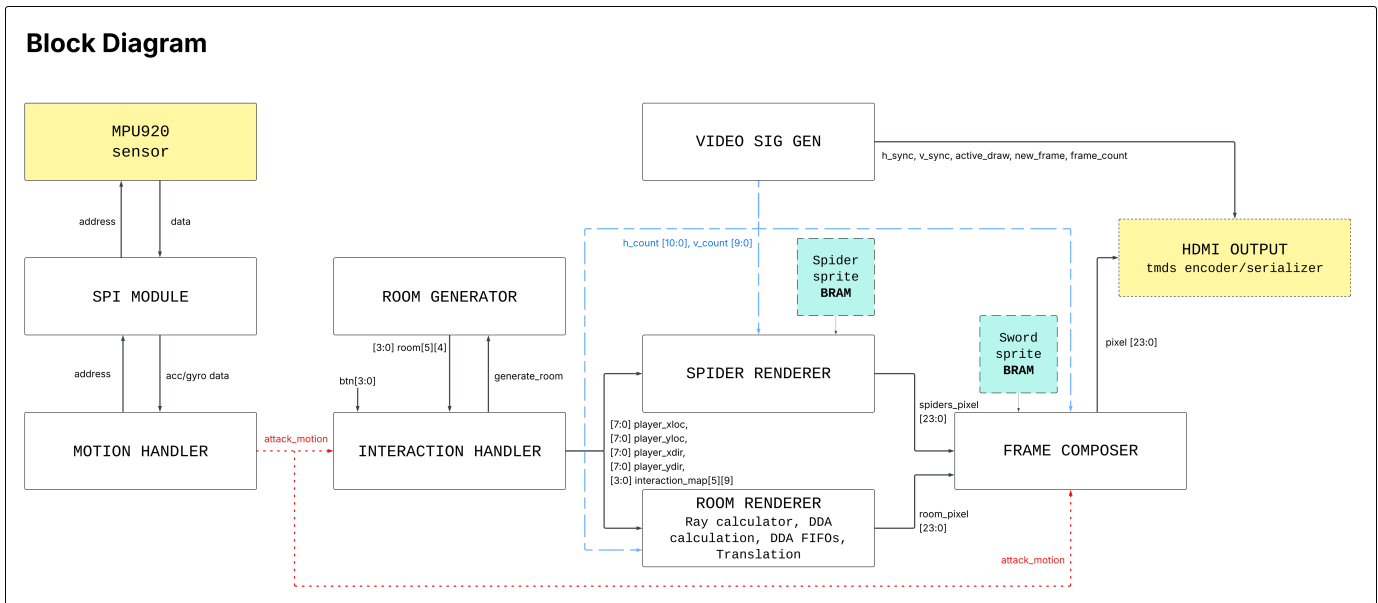


Fig. 2. Block diagram of all modules (excluding raycasting)

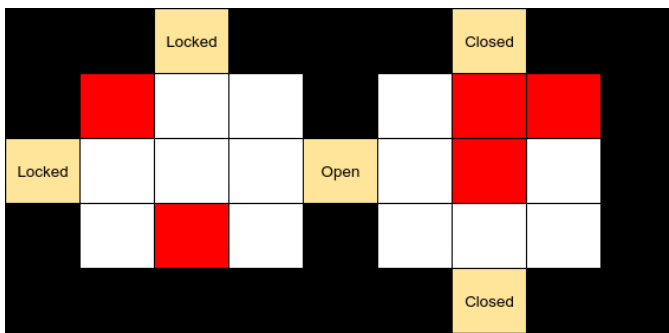


Fig. 3. Room grid representation

arithmetic but also to directly use the integer part to correspond to the block's location in the map grid, which helps with any boundary or interaction checking (e.g. defeating a spider, opening a door, etc).

3) *Spiders*: The enemies in our Spider Slayer are the titular spiders themselves, rendered as 2D sprites currently. Building upon work from prior labs, such as when we had to render popcat through our camera, the spiders will spawn in based on two primary factors: the player's current perspective and the randomly-generated number of spiders calculated per room.

Based on the direction the player is facing and their current coordinates within the grid, the field of view (FOV) generated from that point can be calculated by finding the nearby neighbors the player would find visible. From these neighbors, the spider spawn locations have already been determined based on the room generation module. If one of the visible neighbors has a spider assigned to it, we can render the sprite in and scale it according to how close the player is to the neighbor in terms of coordinates. The number of neighbors is determined by the player's current position, as there are two scenarios utilized:

the player is in the middle of the room, or they are by a wall.

If the player is in the middle of the room, their FOV will allow them to see up to 3 tiles of the 3x3 occupiable space in the room. In the latter scenario, the player's FOV will allow them to see 6 tiles if they are not in a corner, otherwise they can view 8 tiles. Thus, a minimum and maximum x/y range based on the player's position can be calculated and used to check specific parts of the `interaction_map` input.

The width and height of the `image_sprite` for the spider begins as 256x256 and is then scaled appropriately based on the distance from the player, scaling down by a factor of either 1/2, 1/4, or 1/8 as needed. A generator is used for the number of spiders (1-3) generated by the pseudorandom room generator to create the correct number of sprites. To support this, three modules based off of the `image_sprite` module have been created to support a 256x256, 127x126, and 64x64 version sprite of the spider.

The current implementation of the Spider Renderer module sends out a ray into the room until a collision is detected. Using the `interaction_map`, the module determines if the collision is a spider based on the current grid. Since the distance from the collision to the player can be determined via this ray, the spider sprite will be sized accordingly to the player. Currently, the center ray has been implemented fully into the dungeon and any number of rays from the player can be generated at different angles based on this format. Since the angles can be constant and new rays can be derived from sine and cosine, they can be instantiated as parameters to allow the player to rotate and still continue to see the spider in their FOV.

It will then feed into the Top Level file `spiders_pixel`, outputting the spider sprite location and referencing the appropriately scaled module.

B. Room generation

To avoid repetitive gameplay, room generation is done through sampling pseudorandom bits using LFSR. This way, we can randomize which doors appear and what locations in our inner 3x3 room contain spiders. This will prevent the rooms from feeling stagnant and repetitive throughout the game, and add the element of randomness and surprise often found in other dungeon crawlers and their method of dungeon/room generation.

III. MOTION AND INTERACTION

A. Motion Detection

To attack or interact with a door, the player must make a slashing motion with a breadboard containing an IMU (MPU-920 in this case). The FPGA board communicates with it through SPI by sending a read bit and the address of the desired data; for instance, sending 0XBF and 0xC0 would return the high and low bits of the z-axis accelerometer reading. Currently, the z-axis accelerometer is read every millisecond and compared against its previous reading. If the absolute difference is above a threshold (which was empirically tested to be around 0x047F), it is determined that an `attack_motion` was performed, and a cooldown of 1,000,000 clock cycles is initialized.

B. Player Motion

To move around the room, the player will utilize the buttons on the FPGA board to control their current position and direction.

Forward and backward movements are straightforward, requiring only the addition or subtraction of the player directions from locations, checking whether the new location's block is valid (e.g., not a wall), and updating the player location.

Each left or right button press constitutes a change in angle by 10°. Thus, the new direction vectors can be determined by multiplying a rotation matrix with the old directions; to do this, we precalculate and store $\sin(10^\circ)$ and $\cos(10^\circ)$ in Q2.6 format (i.e. one sign, one integer, and six fractional bits).

C. Room Interaction

Through the attack/slashing motion, two kinds of interactions can be carried out: opening doors and attacking spiders. Spider defeat can be easily done by updating the `interaction_map` spider block into an empty one. When opening doors, more complex changes are needed. Since `room_grid` is 5x9 (i.e. horizontal) we must make sure that the rooms are connected correctly. Thankfully, door location possibilities are limited to the center of each wall (except for the left wall, as that is reserved as the door players come in by), so only $\pm 90^\circ$ room rotations are needed. Room changes needed when opening a door are:

- 1) Set opened door as open (0100), and all other doors as locked (0111).
- 2) Rotate left room grid (5x5) and player location/direction.
 - If right door, no rotation needed.

- If top door, rotate by 90° clockwise.
- If bottom door, rotate by 90° counterclockwise.

Another way the room grid can change is by the player moving on to the next room. Once a player's location passes beyond the open door block and into the right room, the grid changes in the following ways:

- 1) Set passed through door as locked (0111).
- 2) Slide right room over to the left side of the grid.
- 3) Generate a new side room to put on the right.

IV. RAYCASTING AND RENDERING

To render the player's view of the dungeon, our rendering module casts rays from the player's location for each horizontal point on the screen, and uses a digital differential analysis algorithm to calculate the intersection point and distance between the ray and the wall. This distance is used to calculate the size the vertical slice of the wall should appear on the screen.

The raycasting script for the project was first developed in Python, before being separated into the necessary modules and translated into Verilog for the final project. The modules comprising the ray caster prototype are the AF and B4 DDA FIFOs, the DDA, and the Ray Calculator.

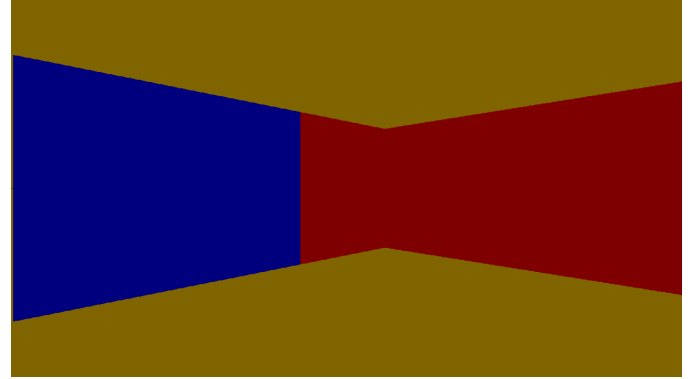


Fig. 4. An example image of the raycasting prototype in Python

A. Ray Generation

Each ray's direction is calculated with the following equation:

$$\vec{ray_dir} = \vec{player_dir} + \left(\frac{2(h_count)}{SCREEN_WIDTH} - 1 \right) \vec{plane} \quad (1)$$

Depending on the relative length of the camera's plane vector and the player's direction vector, this can generate a field of view of various widths. Currently, `camera_plane` and `player_dir` are orthonormal vectors, generating a 90 degree cone of vision.

B. Ray Calculator

The Ray Calculator module is designed to output the correct pixel value depending on the state of the game. If `new_room` is valid, then the module can output the loading/transition

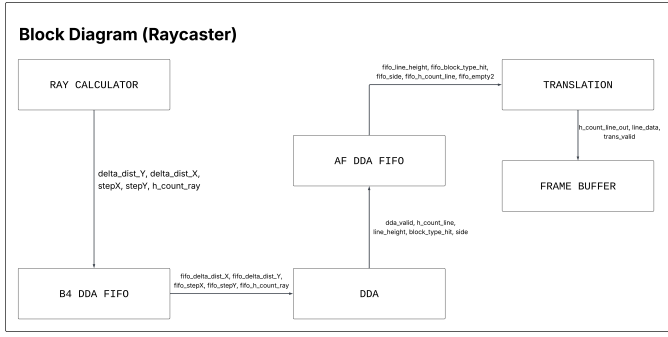


Fig. 5. Block diagram of the raycasting modules

image pixel. If `attack_motion` is valid, then we add the attack sprite animation on top of the right side of the screen. Otherwise, the Ray Calculator will simply output the rendered room pixel.

The Ray Calculator utilizes a simple finite state machine (FSM) to ensure the smoothest transition of data, using four states shown in 6. The module takes and calculates the delta distances in the x and y direction, the step size, and the h_count_ray using the divider module from prior labs and the player's current x and y directions. Furthermore, the Ray Calculator will output a `valid_out` variable, an indicator for when the last ray has been calculated, and a `busy` variable as well. These are all for assisting modules downstream within their FSMs and other calculations as needed.

By expanding upon the Ray Calculator, further stretch goals can be achieved in the future, such as making open doors transparent for the player.

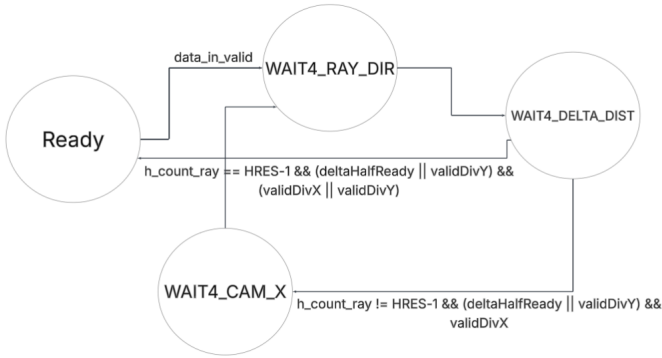


Fig. 6. A diagram of the FSM utilized within the Ray Calculator

C. DDA FIFOs

Due to the variable latency of the Digital Differential Analyzer (DDA) submodule, FIFOs will regulate the temporary build up of data between the ray calculator and the DDA, and between the DDA and the translator.

There are two DDA FIFOs within our project, a "before" DDA FIFO, the `B4_DDA_FIFO` module, to handle the data between the Ray Calculator and the DDA module, and an "after" DDA FIFO, the `AF_DDA_FIFO` module, to handle

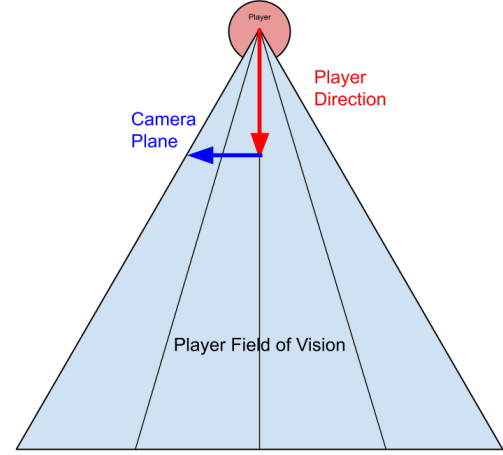


Fig. 7. Diagram of the player's Field of Vision (FOV)

the data between the DDA and the translator. Their logic is closely identical to one another, altered based on the incoming and expected outgoing variables between the two modules it communicates between.

D. Digital Differential Analyzer

The calculated rays are then used to find the distance from the rays' intersection points with the dungeon wall to the nearest point on the camera plane. The distance it takes for the ray to intersect with the next grid mark, $\delta dist$, is the reciprocal of ray_dir . The DDA algorithm steps along each intersection of the ray with a grid mark, keeping track of the distance the ray has traveled, until it reaches a wall block. Since this algorithm has a variable latency, there will be FIFOs between it and the translation module to deal with any temporary build up of data.

The design is inspired by the DDA calculation architecture laid out in the MazeCaster report; however, a key difference is that our module will not need to arbitrate between BRAM accesses, given that our room's smaller size does not require map storage in BRAM [1]. Instead, we will utilize partially-transparent blocks, such as the spiders and doors, to impact the output data. If the spider is set at a high height, then this and other data will be considered in the Frame Composer's determinations of pixel values, allowing us to render the spider over the wall.

E. Translation

The DDA will provide data on the type of block hit, its height, and what side of the block was hit, which will be used to determine what color each pixel of the frame should be. The translation module will build the room's image from this data and supply it to the Frame Composer/Top Level as needed. Currently, the module only needs to select from a few colors, but its job will become more complicated once wall textures are implemented.

When wall textures are implemented to the project, the Translation module will receive inputs from the DDA and communicate with the textures BRAM to compute the value of each pixel to be displayed. This will allow the dungeon to have wall, floor, and ceiling textures that adapt to the player's FOV. Ideally, these textures would resemble cobblestone or dungeon-related textures to provide an appropriate tone and atmosphere to the game.

F. Overlaying Sprites

The frame composer module will take in the pixel data from the room renderer and spider renderer, and overlay any present spider sprite pixels over the pixels from the room. In making this module and completing the system's design, it will be important to ensure that the inputs to the frame composer are properly synchronized.

Currently, the Top Level file serves as our "frame composer" for the time being, as it is capable of rendering the sprites on top of the generated room and relocating them to remain relevant to the player's FOV.

V. CONTRIBUTIONS

The workload was distributed between team members in the following manner. Frank created the raycasting module in Python before translating it over into Verilog and splitting it into smaller modules. Zhi was responsible for creating modules that deal with the IMU, motion, and interpreting player inputs/actions. Ash was tasked with focusing on the spider rendering-related modules and contributed heavily to the report. This was designed to ensure a fair workload amongst team members and to allow them to engage with their interests within the project; however, all members ended up contributing to the modules and designs of others, creating a cooperative and less rigorous environment.

VI. EVALUATION

The current scope of our project allowed us to:

A. Size

The Top Level utilized 9456 total LUTs and 4877 Logic LUTs. Furthermore, it used 60.67% of BRAM and 9.22% of LUTs.

B. Speed

- Due to the inability to fully implement the ray casting functionality, I am unsure if the frame is able to be calculated in the of clock cycles available between each frame.

C. Playability

- The gameplay loop is fully implemented, with spiders spawning, able to be attacked, and rooms being generated.

VII. REFLECTION

Given more time to minimize slack and size and integrate more complex modules, some more stretch goals can be achieved in the nearby future. The first stretch goals to be completed would be providing the walls and floors of the dungeon a texture via BRAMs, and providing a 3D animated sprite to the spiders. From there, we could expand upon providing further gameplay mechanics, including a health bar, interactable weapons and items, and more enemies in future iterations of this project.

VIII. SOURCE CODE

Our source code can be found at this GitHub repository.

IX. ACKNOWLEDGMENTS

The team would like to thank our instructor Joe Steinmeyer and TA mentor Hasan Yildiz for their continued support and guidance throughout this project, as well as the rest of the 6.2050 staff for their advice and assistance.

REFERENCES

- [1] T. Hagenlocker, C. Hu, H. Hussein. MazeCaster: Pseudo 3D World using Raycasting Preliminary Report, 2024.
- [2] A. Lee, S. Qiao, M. Slowikowski. Flap Through the Gap, 2024.
- [3] N. Mustafa, C. Price. FPGA Hole in the Wall, 2024.
- [4] N. Salahuddin, L. Sun. FPGA Fruit Ninja, 2018
- [5] Y. Chen, R. Yin, B. Gao, L. Peng and M. Gong, "Ray Tracing on Single FPGA," 2021 IEEE Asia-Pacific Conference on Image Processing, Electronics and Computers (IPEC), Dalian, China, 2021, pp. 1290-1294, doi: 10.1109/IPEC51340.2021.9421209.
- [6] L. Vandevenne, Computer Graphics Tutorial: Raycasting. [Online]. Available: <https://lodev.org/cgtutor/raycasting.html>. [Accessed: Nov. 26, 2025].