LIÈGE université
Sciences Appliquées

Microelectronics and IC design
(ELEN0037-1)

Project : Raycasting

Julien Boniver s212020
Quentin Rulmont s194355
Tom Weber s203806

1 Game Design

1.1 Game Presentation

Our first idea was to implement a retro-like FPV (First Person View) game with 2-D dynamics but with a 3-D rendering using ray casting engine like in the video game Wolfenstein3D[1]. The player was supposed to navigates on a grid and shoot fixed or moving targets. The game ends when the player has hit all the targets. A Timer is displayed to have a time reference of the performance.
The main challenge of this project was to implement a ray-casting algorithm in the VHDL language. It turned out to be more difficult than expected. The reasons for this are explained in more detail in a later section of the report.
The implementation of raycasting having taken longer than expected, the game is simpler than initially thought.
The game consists in finding the way out of a maze in the shortest time possible. To add a little difficulty to this rather simple game, the player has no access to the maze map.

1.2 Controls

We developed a full communication protocol using all 12 buttons between the SNES controllers and the FPGA. For our application we only required the use of 6 buttons, the use of each button is described in the figure below.

Figure 1: SNES controller

- **Arrows :** Only the up and down arrows are used. They allow the player to move to the front or to the back.
- **A and Y :** Allow the player to rotate to the left or to the right. This choice has been made to allow the player to move to the front or to the back and to rotate at the same time.
- **R :** Allows the player to shoot.
- **START :** Allows to start the game when in the menu.

2 Raycast

In this section, the raycasting will be explained and the implementation in vhdl. As mentioned earlier, the raycasting was the biggest challenge of this project.

[1] https://fr.wikipedia.org/wiki/Wolfenstein_3D

First of all, the main idea behind this rendering method is to display a 2D map as seen from the eye of the player instead of the usual top down view. In order to have the 3D effect, things that are further apart are displayed smaller than thing that are close. This gives the effect of perspective. The algorithm can be simply explained in a few steps. For each frame:

1. initiate i to 0
2. compute the distance and the first wall detected along the direction given by the ray number i.
3. store the height of the wall to be displayed equals to $\frac{\texttt{Display height in pixel}}{\texttt{distance * cos(ray angle)}}$
4. increase i go back to step 2 until all rays are processed

At the end of this simple loop, the result will be an array that tells us for each ray the height of the wall that has to be displayed. To render this, one can simply divide the horizontal space of the rendering window into an equal number of vertical strips equal to the number of ray. Then in each strip, the vertical space is divided into 3 part, the sky, the wall whose height as been computed earlier and the ground.

The last missing piece of this algorithm is to define a field of view as the player does not see at 360° and a way to identify the ray and their direction. A naive solution is to use angles. The player will have an angle from an absolute referential and the field of view will be a range around this angle for example if the angle of the player is θ, the field of view can be $[\theta - 30, \theta + 30]$ for a field of view of 60°. Then for identifying the ray, one can use the angle of the ray which can also be used along with sines and cosines for computing the direction. However, having the sine and cosine in vhdl is tricky as it requires either a large lookup table as we need lot of value to have a good result or use numeric method to compute the sine and cosine which will had a lot of complexity as well as increase the computation time for each ray of the FOV.

The idea we decided to use was to define along the position of the player in the 2D space, 2 additional vector. A direction vector and a camera plane. Those 2 vectors need to be perpendicular at all time and allows us to define rays and get their position without using the angle and the cosine and sine. The idea is that the camera plane represent the Field of view of the player and the ray direction are identify by summing the direction vector of the player and an offset along that plane as can be seen on the following figure. Using this vectorial approach gives us directly the direction of a ray along the x-axis and y-axis and the ray can be identify by the offset along the camera plane. The last part to simplify is to get rid of the cosine in the formula of the height of the wall given above. This cosine correct the fisheye effect. It comes from the fact that if the player is looking at a wall, for each of the ray, the distance will not be the same. Indeed, at the extremities of the field of view, the distance between the player and the wall along that direction will be larger that the distance along the direction that look straight at the wall. This will give the effect that the wall is distorted. However, one can see that this effect will be neglected if instead of taking the distance between the player and the wall along the ray (the euclidean distance), we take as the distance, the distance between the wall and the camera plane. This will give us the same distance for all rays if we look straight at a wall.

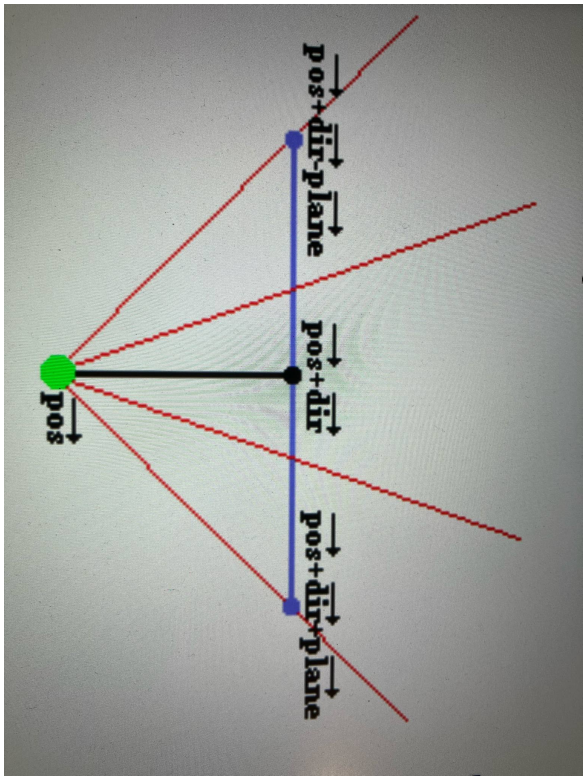

Figure 2: vectorial representation of the ray

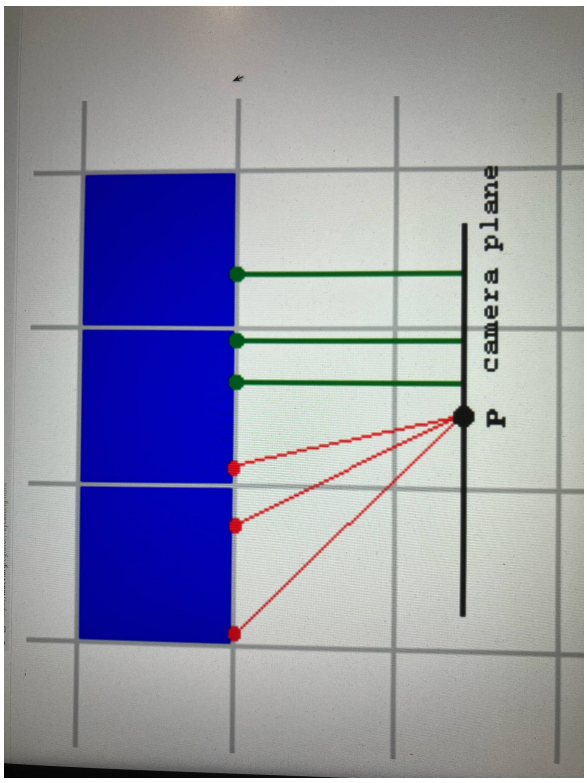

Figure 3: representation of the distance euclidean in red and the distance we use in green

The last step is to calculate the intersection point between the player and the wall along a given ray. To do that one can simply define a fixed step and check after each step in the direction of the ray if a wall is hit or not. Once a wall is it, the number of step times the size of the step gives us the distance. However, doing that is not very optimized as the result will be precised only if the step is small meaning that the amount of computing will increase. Instead, we use DDA which compute the same thing with a variable step size. At each step, we will look at the next intersection between the main grid and the ray as a collision can only appears there.

To improve the 3D effect, wall oriented in the north south direction are displayed in another color than wall in the west east direction. Thanks to the DDA, it is easy to compute wich orientation is intersected.

As a result, the problem is reduced to "simple" vector manipulation for each ray. For each ray, compute the direction of this ray by adding the direction vector and a fraction of the plane vector. Then for each ray, we perform the DDA to compute the distance between the player and the intersection. Once this is computed we divide the height of the screen by the distance and repeat the process for the next ray. This algorithm and the particular implementation we used can be found in more detail in this web page [2].

Finally for the moving the player, for translation, one can simply increase the position along the direction vector with a fixed step when needed. Regarding rotation, it is trickier as both the direction and plane vector need to rotate from the same quantity to keep the result realistic. If those 2 vector are no longer perpendicular, the result will be skewed. To rotate those vector, a simple rotation matrix is used. the vector are multiplied by the matrix $\begin{pmatrix} cos(\alpha) & -sin(\alpha) \\ sin(\alpha) & cos(\alpha) \end{pmatrix}$. The α angle is set to a constant to avoid computing sine and cosine. We can thus just replace

[2] https://lodev.org/cgtutor/raycasting.html

$sin(\alpha)$ and $cos(\alpha)$ by precomputed constant.

3 Code structure

The architecture consists of a main entity that dictate and control the behavior of the game by instantiating several other entities.

- **Wolfenstein3D :** The central entity that regroups the processes of the other entities and handle the scheduling for computing the frame.
- **Menu :** Display the menu.
- **Vga :** Handles all the elements that need to be displayed during the game.
- **Player :** Updates player parameters according to controller inputs.
- **Raycast :** Computes the perceived height of the wall for one ray.
- **Clocks :** Generates all the different clock frequencies used.
- **SNES :** Communication protocol of the SNES controllers.
- **Timer :** Generates a timer to display a counter on the screen.
- **SharedTypes :** Defines types used in other entities.

3.1 Entities

3.1.1 Wolfenstein3D

The central entity in the architecture is referred to as Wolfenstein3D. This entity consists of driving the states of the game. This is handled with one main FSM (Finite-State-Machine) `"game_state"`. This state machine has only two possible states, MENU and GAME. The state GAME handles the game logic and implements another FSM `"state_playing"` that computes all the ray given the player parameters. Then the height of the walls are computed given another FSM `"stateRay"`. The states of the game are illustrated here below.

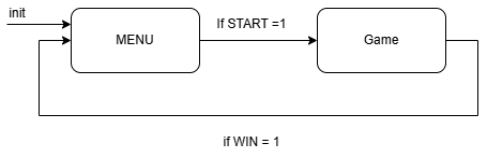

Figure 4: FSM game state

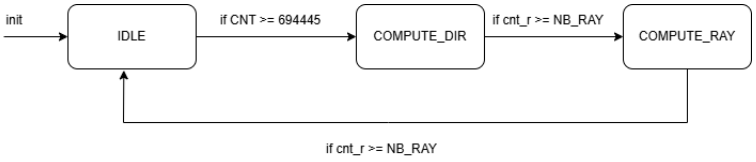

Figure 5: FSM playing state inside Game state

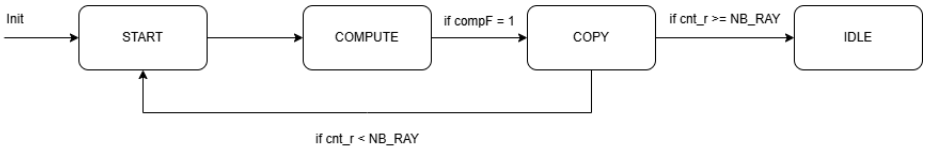

Figure 6: FSM raycasting state inside COMPUTE RAY state

3.1.2 Menu

The menu entity handles the display of the menu, which is pretty basic. The words appearing on the screen are made by assembling letters created by 8x8 bitmap. It was choosen to separe it from the Vga entity for more clarity. The entity is active when the flag `"menu_active"` is raised in the main entity "Wolfenstein3D".

3.1.3 Clocks

The Clocks entity is responsible for generating two clock signals, `clk_snes` and `clk_timer`, respectively used in the SNES entity and timer entity.

3.1.4 SNES

The SNES entity ensures communication with the SNES controller. The process is based on the `clk_snes` signal and is pretty straightforward.

3.1.5 Player

The Player entity is responsible for updating the player parameters ("posX", "posY", "dirX", "dirY", "planeX" and "planeY") and derive these signals for the following entities and game logic.
Player movements are allowed through the use of two variables "pX" and "pY". These variables are used to check if the player hits a wall or not. If the player does not hit a wall, the player position is incremented by the term $speed * playerDirection$.
The player direction as the camera plane are updated at any time given the following formulas.
Rotating to the right :

$$\begin{aligned} dirX &= dirX * cos + dirY * sin \\ dirY &= dirY * cos - dirX * sin \\ CamPlaneX &= CamPlaneX * cos + CamPlaneY * sin \\ CamPlaneY &= CamPlaneY * cos - CamPlaneX * sin \end{aligned}$$

Rotating to the left :

$$\begin{aligned} dirX &= dirX * cos - dirY * sin \\ dirY &= dirY * cos + dirX * sin \\ CamPlaneX &= CamPlaneX * cos - CamPlaneY * sin \\ CamPlaneY &= CamPlaneY * cos + CamPlaneX * sin \end{aligned}$$

cos and *sin* are defined as constant in this raycast algorithm.

3.1.6 Raycast

The raycasy entity is responsible to compute the color and the height of the wall to be displayed. It computes those 2 quantities for one ray. It is the Main entity that is responsible of giving this entity the direction of the ray. It implements the algorithm described in the raycast section. A simple state machine has been used to do that.

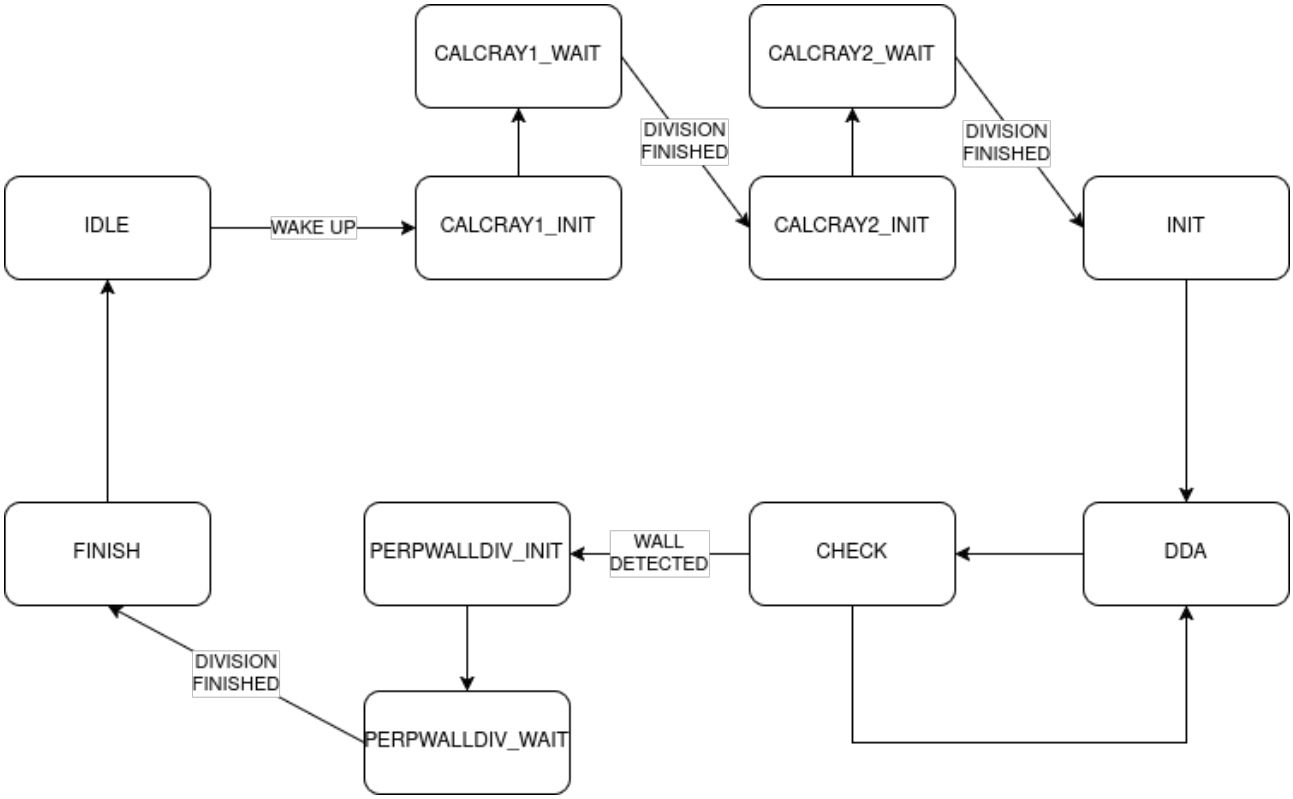

Figure 7: FSM raycast

As one can see above, we performed the division one after the other. It allows us to instantiate only one divider to reduce the number of logical element of the project. Once this entity is woken up, it will perform the division and initialize all the variable int the first states. Then it will loop between DDA and check until a wall is it. Once the wall detected flag is raised we will perform the last division and enter the finish state that will communicate with the main entity to give the height and color of the wall then the sate will get back to IDLE until it is woken up again.

3.1.7 vga and timer

The vga entity handles the display of the game. A rectangle of 512x780 pixels is dedicated to the player view. A timer and the player's live (just for the show) is displayed below this rectangle. The timer is handled thanks to the timer entity.

3.2 Game States

There are only two possible game states : Menu and Game. When the FPGA is powered, the user faces the menu. If the users presses the START buttons, he begins to play the game. When the player find the way out of the maze, it goes back to the menu state.

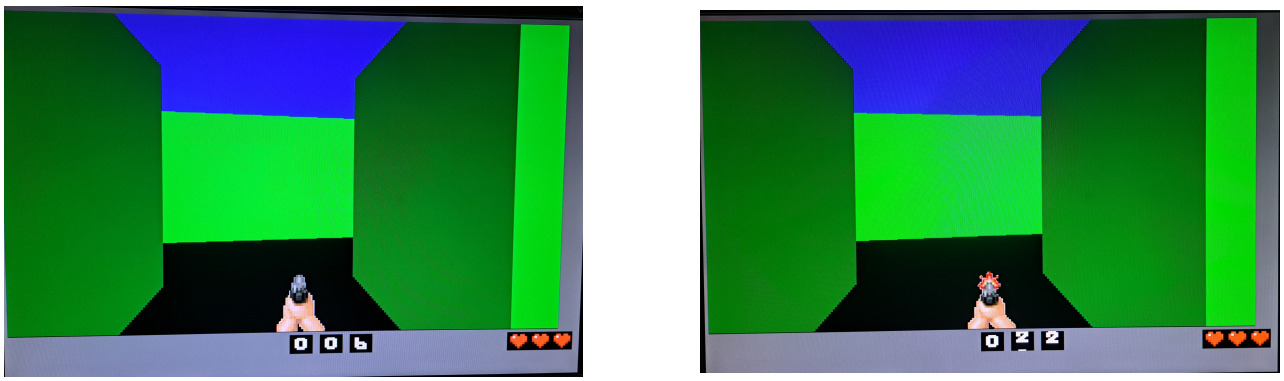

Figure 8: Images taken from the gameplay. On the left, the player is still. On the right, the player is shooting.

4 Problem solving strategies

4.1 Raycast

The main problem encounters in this section where to do math on real number. The final solution implemented was to use a Q16.16 representation of the number it means that each number is represented with 32 bits. The 16 most significant bit are natural power of 2 as for integer representation. The 16 least significant bit are negative power of 2 they represent the non integer part of the number. To represent positive and negative number, the two's complement is used. This number representation allows us to perform addition and subtraction without changing anything. However for the multiplication and division, a scaling factor need to be applied as the q16.16 representation is the same as representing the number time $2^{16} = 65536$. Once this scalling factor applied, the multiplication and division can be applied on the number and the result will be the result in q16.16 representation. We use this representation as it allows us to have a good precision which is needed in the rotation to preserve the direction and plane vector perpendicular.

Considering the division, the LPM_DIVIDE IP as been used. It allows us to perform the division in 20 clock cycle. This was needed because a division on two 64 bits(64 and not 32 due to the scalling factor needed) takes quite a lot of time. Our clock was to fast at 50 MHz to perform this operation on one clock cycle we thus spread the computation onto 20 clock cycle thanks to this IP. Another solution to simplify the division and allow quartus to synthetise it better was to always compute the inverse of a positive number. This work in our case. Indeed the first two division where to compute the absolute value of the inverse of the ray direction along the x and y direction. Computing the inverse of the absolute value yield the same result. For the final division, it was computing the height of the wall. If instead of dividing the screen height by the distance, we multiply the screen height by the inverse of the distance (always positive) then all division were replaced by inverse calculation. Regarding the height of the wall computation, we chose 512 pixel height to display the game thanks to that, we can shift left the result of the inverse by 9 instead of performing a full multiplication.

4.2 Display/ROM

To integrate some images in the game some methods have been tested. The first one, maybe a straightforward solution, consist in storing repeated elements only once to save a considerable amount of space. Secondly, two tricks have been implemented to reduce the numbers of bits needed to save RGB image original on 12 bits :

1. The black and white image like the numbers were encoded on 1 bit so a image of `32x32` goes from 12 288 bits for RGB to 1 024 bits, then the pixel value is rescaled on 12 bit for VGA communication.

2. For the RGB images, we decide to compress them by looking for the number of unique color in each image which gave us the minimal number of bits needed to encode this image, then we associate a number to each of this unique color and encode this number on less than 12 bits (the number of bits depend on how much the colors are repeated, the more repetitive the image's color, the better the compression would be. The image is then decoded thanks to a LUT (look-up table) which associates the number of the unique color to its RGB code on 12 bits. Since adding a LUT, even of hundreds entries, is negligible compared to the space saved, it seems a good solution.

Image's name	pixels	unique color bits	12 bits size	encoded sized	compression
`heart.png`	1,024	5	12,288	5,120	2.4
`frame1.png`	6,365	7	76,380	44,555	1.714
`frame2.png`	6,365	8	76,380	50,9205	1.5
`frame10.png`	6,365	6	76,380	38,190	2.0
`frame11.png`	6,365	7	76,380	44,555	1.714
`frame12.png`	6,365	7	76,380	44,555	1.714

Those 2 strategies saving respectively 112,640 and 166,293 bits on what those images would have take in the ROM normally.

5 Future improvements

In terms of gameplay, we could have added more interactions with the environment such as fixed and moving targets to shoot at, instead of only walking through a maze. We could also have implemented a time limit to find the way out of the maze. In terms of graphics, we could have added wall textures to make the game more immersive. With these improvements, we truly believe that the game could have been really fun and satisfying to play with. The biggest part was already tackled in the raycast. Adding stuff will be just manipulating flag. As an example, we can very simply add a zoom button by dividing the plane vector by Two once it is pressed and multiplying the wall height by 2. This reduce the FOV from 66 to 33 degrees and makes the detail twice as big. Another part easy to implement will be target on the wall and the ability to shoot them as the raycast is already there. The only thing to add is another identifier in the map grid for the target and a flag that tell us if the ray whose offset is 0 (the ray wich represent also the aim of the gun) intersect the target. If this flag is set to one and the player shoot then the target is touched. This gives us a bullet of infinite speed but could have been a nice and easy improvement if time would have allows.

For the debugging of the raycast, we could have been more efficient if we really divided our problems into the smallest blocks possible instead of testing a bit of every part at the same time. This could have saved us precious time to better develop the gameplay. However, depsite the fact that the game is rather simple for now, the main challenge was to implement the raycasting algorithm. The gameplay options proposed here above would not have been a too difficult task but it feels unfinished. We are still very proud of what we acheived.

6 Members contribution

The group collaboration was going well. We spent many hours together in the R100 lab working on different tasks. Tom took over the ROM and graphics parts while Quentin and Julien were in charge of implementing the raycast and game logic part. In the end, everyone was working together to put all the parts together to create a playable game. Overall, the project was a perfect collaboration and we had a lot of relief and enthusiasm when our project was finally working !