Algorithm and Programming (COMP6047001)

Final Project Documentation

Philipus Adriel Tandra: 2502031715

Ray-Casting Project

Table of Contents

Overview

The videogame DOOM released in 1993 was one of the most revolutionary games that would redefine the FPS genre for years to come. The specific technology that I aimed to be replicate was something called ray-casting.

Ray-casting is a technique that uses “light rays” which are emitted from the players point of view, hence the name. It can make 3D projections with the use of a 2D map.

With each “light-ray” falling onto a 2D wall, the computer will process height and depth and create a 3D projection based on these “light-rays” thus not only showing a 3D plane but also a first-person perspective.

During the production of Wolfenstein 3D, id Software aimed to develop a 3D game with the use of ray-casting. Soon enough, their future games would inherit this technique from Wolfenstein to games like DOOM and Quake.

It was revolutionary, we could actually see our videogame characters points of view. This was the birth of the First-Person Shooter genre.

Conception

Originally, I actually aimed to recreate DOOM in pygame considering that I have played the recent remastering of the game. However, with a few errors that I will go over later and a few time constraint issues, I thought that the implementation of guns, enemies and sprites for textures seemed too ambitious especially for my first semester.

Nevertheless, Pygame has several features that can help with the ray-casting, such as the use of shapes. By using that, I could create the player, the map, “light rays” and field of vision simply with the use of a combination of lines and rectangles. I could also use it for the 3D projection later on by making the height and depth variables and compiling them into different rectangles.

I started production on the 28th of November and used the PyCharm IDE due to my familiarity with the IDE.

Execution

Modules:

- PyGame
  - Used for the window and the different functions in the program
- Sys

- Used to exit the python program in the main loop
- Math
  - Math is involved for moving in a 3D plane and we need trigonometry for the projection of light rays from the player

Files:

- main.py
  - Draws the window, map and contains the main loop and ray-casting algorithm
- settings.py
  - Contains all of the variables including window, ray-casting, PyGame, player and colour variables.
- player.py
  - Contains the player class with player attribute and method for movement

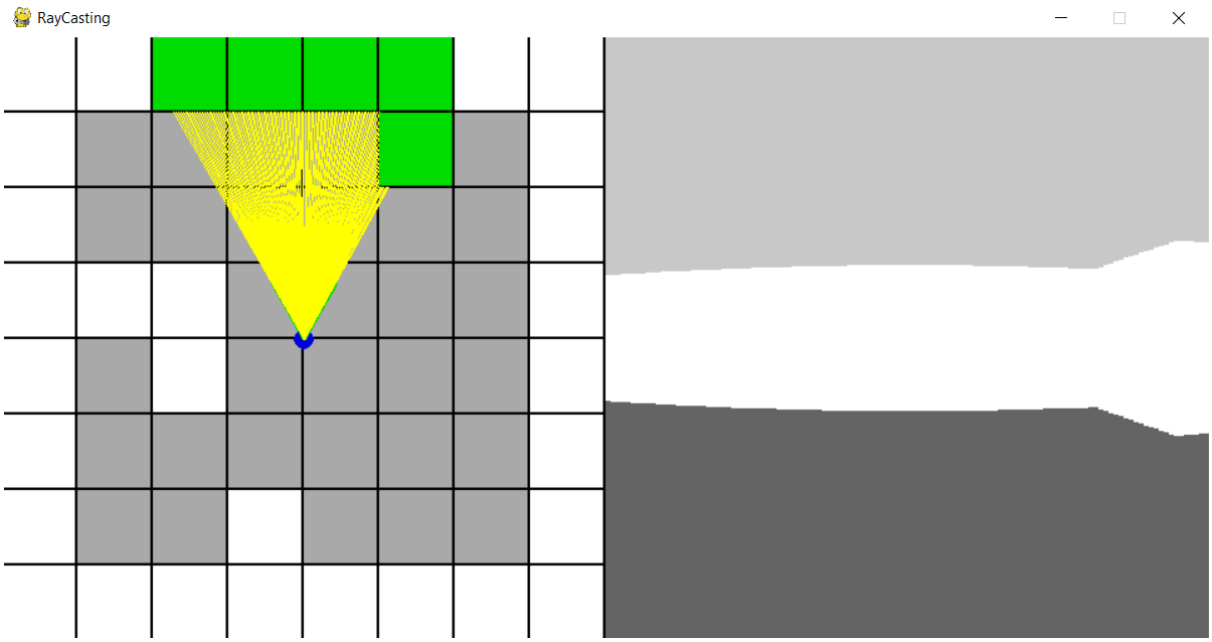


Figure I: What the player first sees

The game window is 480 by 960 with a half of the screen showing the 2D map with the player and its field of view. The other half shows the 3D projection that is shown by the left-hand side of the screen. The map is 8 by 8 with the white squares acting as walls and the grey square acting as empty space or perhaps the floor.

The player can move with the W, A, S and D keys and use the left and right arrow keys to look left and right around the room. There is also relaxing background music that loops as the player explores the wonders of ray-casting.

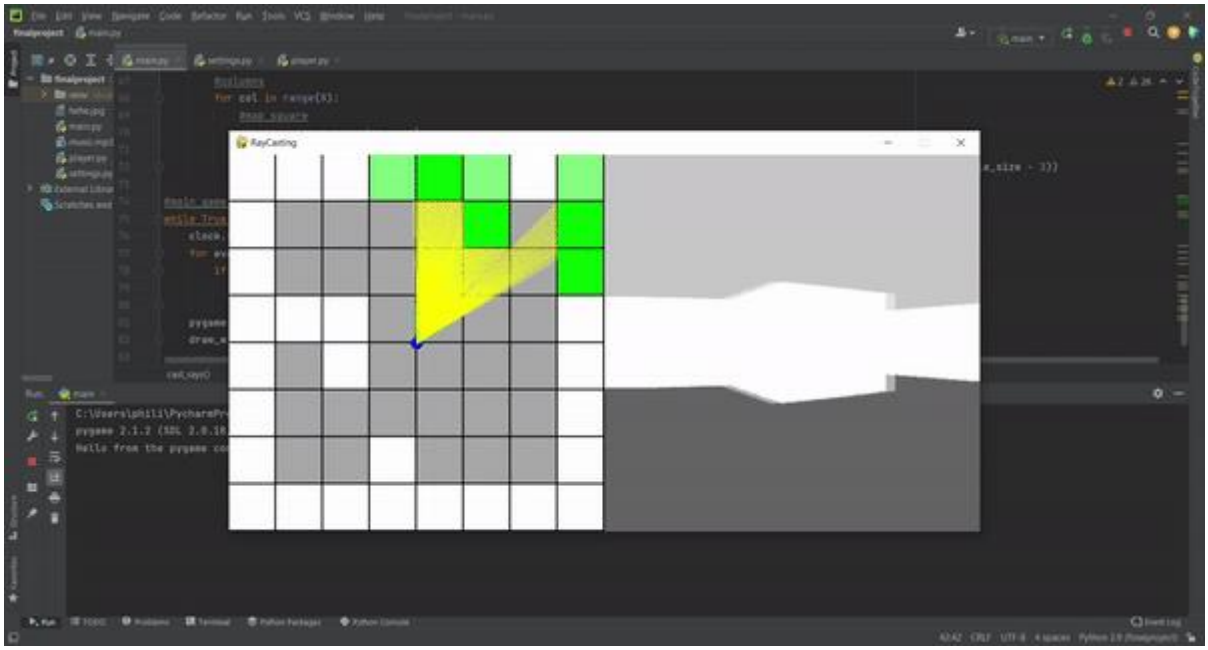
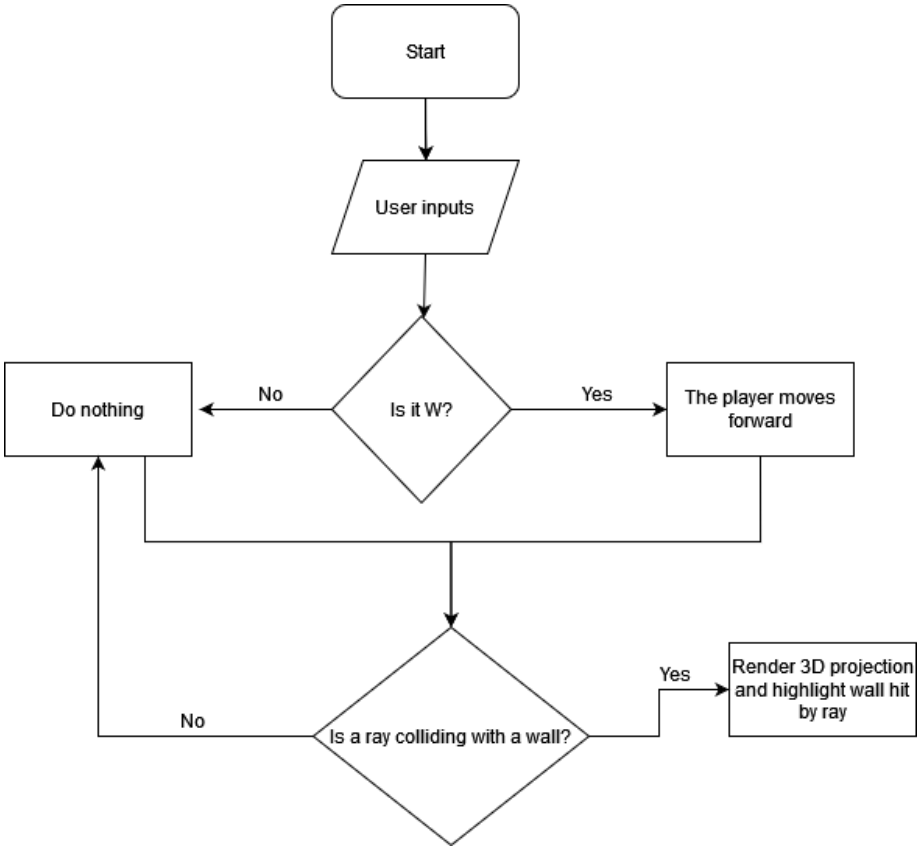


Figure II: Player looking around

Class Diagram

Player
-player_x: int -player_y: int -player_angle: int
+movement(self)

Flow Chart



Problems Faced

Before we analyse the code, I want to look back at all the errors I ran into very often.

ImportError: Cannot import name X

ZeroDivisionError: float division by zero

These two errors were a real pain to deal with. At one point I actually I redid the entire project due to the import error. However, I finally know how to solve these problems. The import error came from a cyclic import error. That means that I would import two files at the same time.

For example: I have my main.py file and I am looking to import variables from main.py to my settings.py file and vice versa. This would actually cause an error and instead I can only import from settings to main or main to settings. Not at the same time. I had never heard of this error and rarely try to make projects that involve multiple files.

Because of this, I decided to redo the entire project altogether and became more careful with my imports. Originally, I actually wanted to make the map and ray casting in their own files and keep the main file mostly empty to make it look cleaner. In this attempt, I ended up destroying my entire project and making things really complicated.

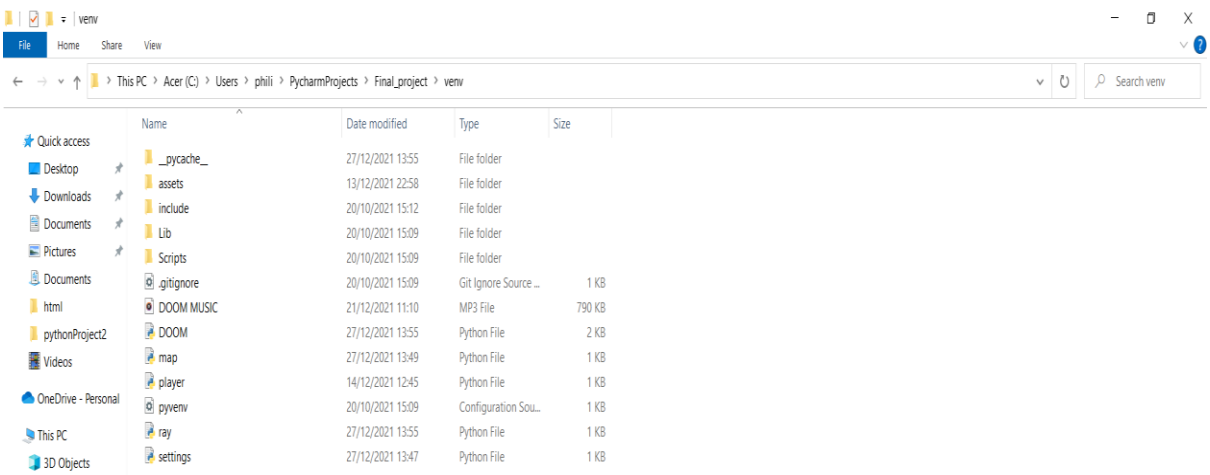


Figure III: Original final project

Another problem was that ZeroDivsionError. Because of the abundance of math involved in the code, there were a lot of problems that arose from it including this error in particular. This is how I fixed it.

```
wall_height = 30000 / (depth + 0.0001)
```

I added a number that was so small that it would barely make a difference but also close to zero

Map Code

```
map = (
    "########"
    "#     # #"
    "#       #"
    "###     #"
    "# #     #"
    "#       #"
    "#   #   #"
    "########"
)
#map variables
map_size = 8
tile_size = int((width / 2) / map_size)
#map function
def draw_map():
    #rows
    for row in range(8):
        #columns
        for col in range(8):
            #map square index
            square = row * map_size + col
            #drawing the map, if there is no #, the color changes to grey
instead of white
            pygame.draw.rect(win, white if map[square] == "#" else
dark_grey, (col * tile_size, row * tile_size, tile_size - 2, tile_size -
2))
```

First let's take a look at the code used to make the map. There are for loops for both the row and columns. Then we calculate the square index that we will use to call each row or column from the tuple.

We also have an if function in the pygame.draw.rect function to show grey when there is no "#". Then in the next parameter there is the map length and height with their respective sizes. The tile size is equal is half of the width which means that it will take up the horizontal half of the screen.

After that, each "#" would be walls and the empty spaces being actual empty space. With this method, I would be able to manipulate the map however I liked by adding and removing walls.

Now let's take a look at how I made the player.

Player Code

```
import pygame
from settings import*
import math
#player class
class Player:
    #attributes
    def __init__(self):
        self.x = player_x
        self.y = player_y
        self.angle = player_angle
    #player movement method
    def movement(self):
        sin_a = math.sin(self.angle)
        cos_a = math.cos(self.angle)
        keys = pygame.key.get_pressed()
        if keys[pygame.K_a]:
            self.x += player_speed * cos_a
            self.y += player_speed * sin_a
        if keys[pygame.K_d]:
            self.x += -player_speed * cos_a
            self.y += -player_speed * sin_a
        if keys[pygame.K_s]:
            self.x += player_speed * sin_a
            self.y += -player_speed * cos_a
        if keys[pygame.K_w]:
            self.x += -player_speed * sin_a
            self.y += player_speed * cos_a
        if keys[pygame.K_LEFT]:
            self.angle -= 0.03
        if keys[pygame.K_RIGHT]:
            self.angle += 0.03
pygame.draw.circle(win, blue, (int(player.x), int(player.y)), 8)
```

The movement method in the Player class uses if functions on the keys pressed for the user to control the player. One thing I learned is that if used elif instead of if, the game would not allow the player to input several inputs at the same time, so just if functions would be the best in this case. You can see that we use cos and sin for the player movement. That is because all FPS games make use of tank controls as opposed to free movement found in 2D games. This means that left and right would be completely dependant on where the player is looking at. The

player model is a simple blue circle that contains the player class for it's parameters.

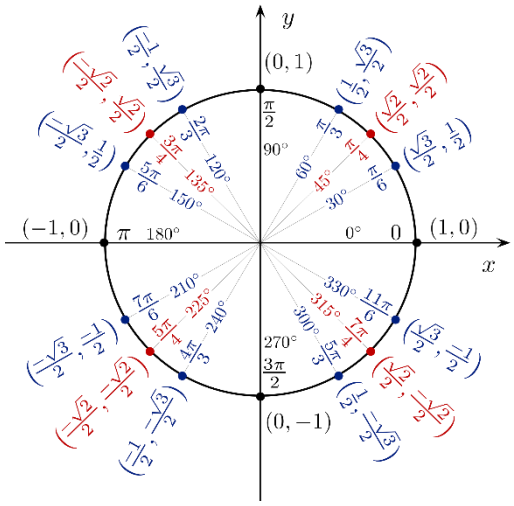


Figure IV: Unit Circle

Player FOV Code

```
FOV = math.pi / 3
half_fov = FOV / 2

pygame.draw.line(win, green, (player.x, player.y), (player.x -
math.sin(player.angle) * 50, player.y + math.cos(player.angle) * 50), 3)

pygame.draw.line(win, green, (player.x, player.y), (player.x -
math.sin(player.angle - half_fov) * 50, player.y + math.cos(player.angle -
half_fov) * 50), 3)

pygame.draw.line(win, green, (player.x, player.y), (player.x -
math.sin(player.angle + half_fov) * 50, player.y + math.cos(player.angle +
half_fov) * 50), 3)
```

The first pygame.draw.line() function is to find the player's direction. It starts from the player's position and uses similar logic from our tank controls with the use of more trigonometry. We then get our player and direction but the FOV is still missing. Think of each of these next two lines as the visual limit for the left and right eye of the player. We just get the player direction code and add or subtract the player angle which will result in the three lines that act come out from the player that can act as their Field Of Vision.

Ray-casting Algorithm and Code

```
max_depth = 480
casted_rays = 120
step_angle = FOV / casted_rays

#ray casting algorithm
def cast_rays():
    start_angle = player.angle - half_fov
    #loop over casted rays
    for ray in range(casted_rays):
```

```
        for depth in range(max_depth):
            target_x = player.x - math.sin(start_angle) * depth
            target_y = player.y + math.cos(start_angle) * depth
            col = int(target_x / tile_size)
            row = int(target_y / tile_size)
            square = row * map_size + col
            if map[square] == "#":
                #highlight wall hit by ray
                pygame.draw.rect(win, green, (col * tile_size, row *
tile_size, tile_size - 2, tile_size - 2))
                #draw casted rays
                pygame.draw.line(win, yellow, (player.x, player.y),
(target_x, target_y))
                break
        #incremental rays
        start_angle += step_angle
```

First, let's take a look at the variables. casted_rays is equal to the amount of rays that we want coming from the player. Before, we only had three rays but we want more rays for better 3D projection. The start_angle variable has the same angle as the second pygame.draw.line() function that was used to get our visual limit for our "left eye". Now, we want to incrementally add rays by making a loop for each ray up until 120.

The maximum depth is equal to the height of our PyGame window because we want our depth of vision to catch and see everything in our half of the window. For target_x and target_y, we have another similarity to our pygame.draw.line functions used for our FOV, this is because we want it to multiply by depth instead of a constant, because we want that incremental increase. Then we have our col and row variables, these are to convert our target_x and target_y into column and rows by dividing it by our tile sizes.

The square, if function and even the parameters in the pygame.draw.rect() is taken directly from our draw_map() function from before and we are left with a highlighter on which "#" or walls are being highlighted and which rays are hitting and colliding with said walls. The parameters are the same because the highlight is about the same position, width and length as the actual map grid just with a different colour and it is responsive to the "light rays".

We end with the incremental increase of start angle by the step angle thus creating 120 rays in total. This is the heart of our program and will be instrumental in our 3D projection

3D Projection Code

```
scale = (width / 2) / casted_rays
#calculate wall height
wall_height = 30000 / (depth + 0.0001)
#3D floor
pygame.draw.rect(win, gray, (480, height / 2, height, height))
#3D ceiling
pygame.draw.rect(win, light_gray, (480, -height / 2, height, height))
```

```
#draw the 3d projection
pygame.draw.rect(win, white, (height + ray * scale, (height / 2) -
wall_height / 2, scale, wall_height))
```

For the 3D projection, it is actually pretty simple due to the fact that we have already done the casted rays. The scale is basically 480 which is half of the screen divided by our number of casted rays which is about 120. We do this because we want it so that when the player moves closer to a wall, it will increase in size and when they go further away, the wall will shrink. So then, we make the floor and ceiling which is just grey and a lighter shade of grey.

All that is left to do now is make the walls which is just the depth which is how far we want to look at something and a random constant I just picked. We then draw the 3D projection with the x coordinate being based of our rays from the casted rays multiplied by our scale for that shrink and grow illusion and the y coordinate being the vertical half of the screen reduced by the wall height divided by 2 so that it is in the middle of the screen and the player has a good height in his virtual environment. The width and height would be the scale and the wall height because we are only technically drawing the walls as the ceiling and floor have already been made.

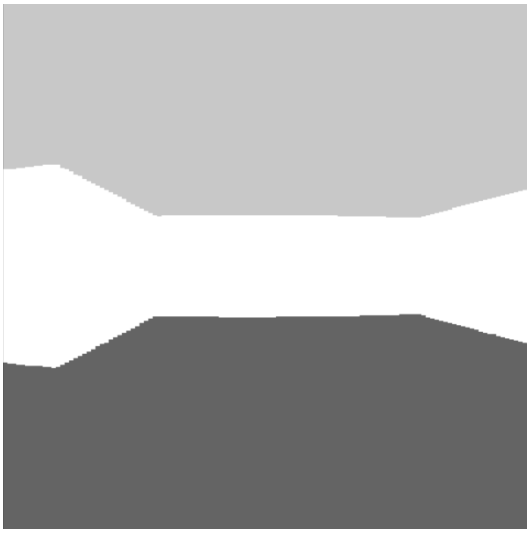


Figure V: Vision

Reflection

The key points I learnt in this project was the use of PyGame and how fun and simple it was to create something from scratch. Another is the abundance of math involved in programming, I was very surprised to see this amount of mathematics and trigonometry in something like programming. I also learnt that the Python programming languages uses radians instead of degrees, that took a while to get used to.

I also can't help but wonder how a game developed during the 1990s creating something as complicated as this, let alone with limited technology. I always thought that ray-casting was an awesome technique but I can't help but admire it even more now that I have felt the effort put into creating something like it. I think I should also be a little less ambitious for projects, the actual scope of this project was far bigger than I could have handled so I am quite glad that I had to cut corners and make for time because these things happen in actual practical work so I am thankful for the experience.

Overall, programming can be a lot of things; challenging, fun, interesting or just downright frustrating but overall, this project had a satisfying result and the feeling of using and incorporating all of the things I have learnt like classes, functions, variables and loops all culminated into something that is honestly quite beautiful.

Thank you, Sir Jude.

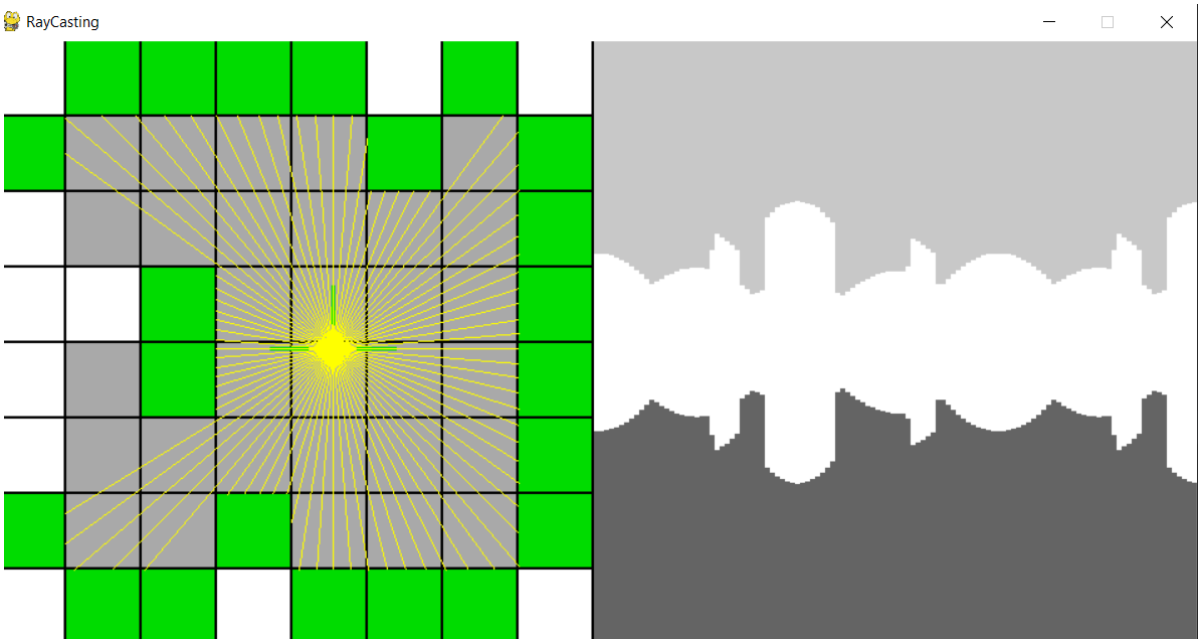


Figure VI: Fun in the sun