Fall 2021

T-511-TGRA, Tölvugrafík

3D Maze game

(Halldór Andri Atlason), (Halldoraa20@ru.is)
(Tómas Elí Jafetsson), (Tomasj20@ru.is)

Október 11, 2021

Introduction

In this report we will explain the design steps, implementation and how we solved certain problems we faced in the implementation of our game.
We had the grading rubric in mind when we designed the game, but certain requirements, where something we did not want to include in our game, f.x top down view. Which would basically ruin the game, since the player would always know where the monsters were.
Our focus for this assignment was to make a fun game, with good and realistic graphics and smooth gameplay. So the biggest question before starting this assignment was, what was the best way to achieve this goal.

Collision check

Logic

In designing the collision checks for our walls, we had to do a lot of calculations. We had to account for the walls x-minimum, x-maximum,
z- minimum and z-maximum positions for each and every wall. We did this by taking the x and z coordinates of the wall and subtracting/adding the x and z scale divided by two. Then all we had to was check which side of the wall the player was hitting and returning True if he was hitting any of the walls sides f.x. If the player was hitting the left side of the wall, the program would return:

left_side_collision = True.

When we knew which side of the wall the player was hitting we had to make him slide accordingly in the right direction. We did this by checking which direction the player was going and what wall he was hitting, then we simply just made him slide by pushing him either positively up the x-axis, or negatively down the x-axis and of course not make it possible to move any further down the z-axis (1.5*delta_time, 0, 0 or -1.5*delta_time, 0, 0), depending on which direction he was trying to walk to.

Problems we faced

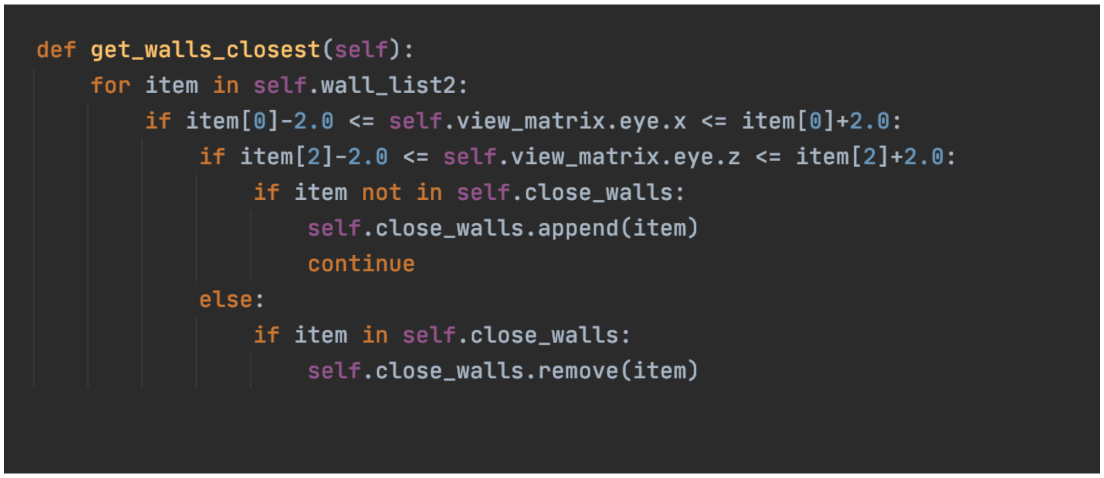
When the player was colliding with a wall, he would often get stuck to the wall when he tried to walk away or got in a corner. We solved this by checking if the player was looking away from the wall, and if he was he got to walk freely.
(0, 0, -1.5*delta_time).

Result

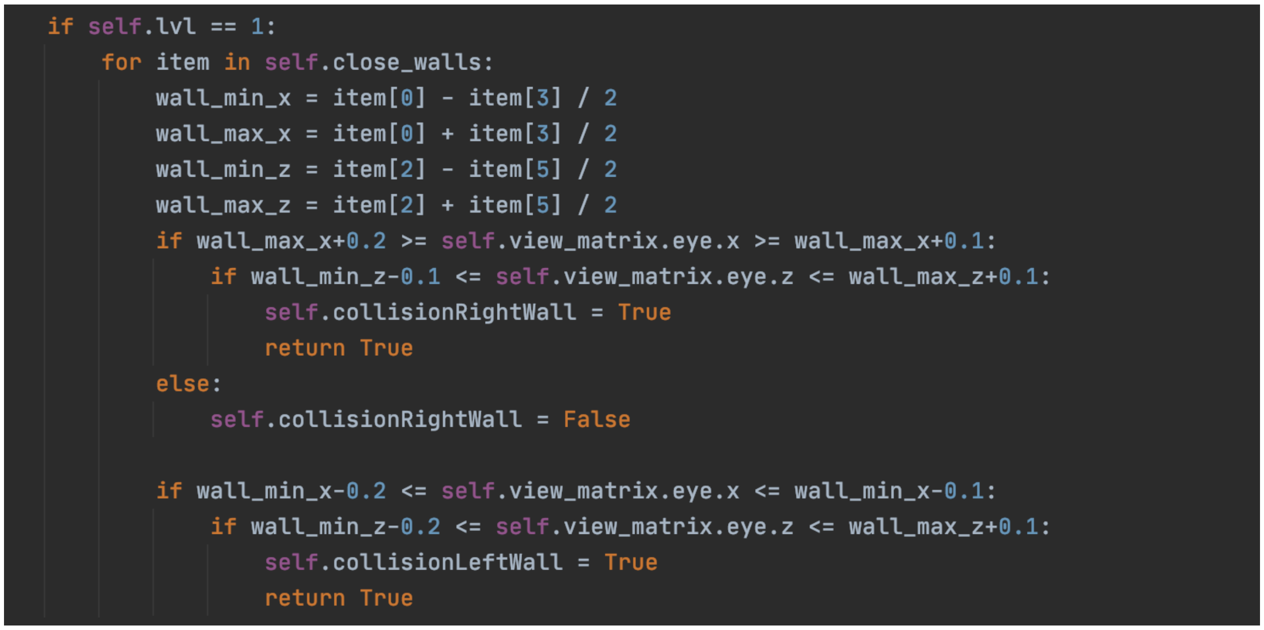
Our collision worked perfectly, the only problems we might face in the future would be because of proper drawing of the maze. The walls must connect.

Aftermath

We actually managed to make our code better for the collisions the day after we had to hand in our assignment. We made a new list called “close_walls” where we steadily added and removed walls to check for collision based on the distance from the player. Now our code only checks for collisions with the walls that are close to him.



```
def get_walls_closest(self):
    for item in self.wall_list2:
        if item[0]-2.0 <= self.view_matrix.eye.x <= item[0]+2.0:
            if item[2]-2.0 <= self.view_matrix.eye.z <= item[2]+2.0:
                if item not in self.close_walls:
                    self.close_walls.append(item)
                    continue
            else:
                if item in self.close_walls:
                    self.close_walls.remove(item)
```



```
if self.lvl == 1:
    for item in self.close_walls:
        wall_min_x = item[0] - item[3] / 2
        wall_max_x = item[0] + item[3] / 2
        wall_min_z = item[2] - item[5] / 2
        wall_max_z = item[2] + item[5] / 2
        if wall_max_x+0.2 >= self.view_matrix.eye.x >= wall_max_x+0.1:
            if wall_min_z-0.1 <= self.view_matrix.eye.z <= wall_max_z+0.1:
                self.collisionRightWall = True
                return True
        else:
            self.collisionRightWall = False

        if wall_min_x-0.2 <= self.view_matrix.eye.x <= wall_min_x-0.1:
            if wall_min_z-0.2 <= self.view_matrix.eye.z <= wall_max_z+0.1:
                self.collisionLeftWall = True
                return True
```

Texture

Logic

We wanted to apply textures to the items in our maze, even though we had no prior knowledge of how to do that, and that would not be relevant nor required in this assignment. We did not care. We are very interested in video games, and the development of them, and we did not like to have our walls or objects colored with one basic color(boring). So we dove into Kári's videos from 2019 on textures, looked at a lot of other videos, and checked out code examples on the internet. We spent way too much time doing something that would not be graded for this assignment, but we were very happy with the texture that we got, even though it was not that perfect.

Implementation

First thing we did was to make a load_texture function in our main program, then we found the textures we wanted to put on our walls, and selected the materials we wanted to put the texture on. Then we needed to make a texture array, and a texture map for our .obj files. We sent the texture into the load_texture function by binding the texture to the texture id before drawing the object, and then disabled it after we drew the object. We had to send both diffuse and specular versions of the texture to our shader, with the corresponding id of the texture object. From there we sent it to our vert shader and frag shader so we could apply the right lighting.

Result

We managed to apply the textures and we were pretty happy about the result, since it transformed our maze to a much more realistic version of itself.

Lighting/shaders

Logic

We needed 2 different light sources, other than the global light. So we decided to give the player a flashlight, and a lantern light.

First we implemented the global light in our maze, the global light is made by defining a light direction vector instead of a position vector, we have to negate the global light direction vector to switch its direction, and normalize the vector since it is unwise to assume the input vector to be a unit vector. Because all of the light rays are parallel, it makes no difference how each object is positioned in relation to the light source because the light direction is the same for all of the objects in the scene. Because the light's direction vector is constant, the lighting computations for each object in the scene will be similar.

Then we implemented the flashlight. We did this by defining the spotlight's position vector (to calculate the fragment-to-light's direction vector),
the spotlight's direction vector, and the cutoff angle. The cut off angle was calculated by getting the cosine value based on an angle and then finally we sent those values to the shader and from there to the fragment shader.
To implement the attenuation we needed 3 extra values in the fragment shader: namely the constant, linear and quadratic terms of the equation. The constant is always zero and the others were something we defined based on our maze.

We put some great comments in our code in the fragment shader but I will also put them here:

//Theta is the angle between the Light_dir vector and the u_light_direction vector. The θ value should be smaller than Φ to be inside the spotlight.

//calculate the theta θ value and compare this with the cutoff ϕ value to determine if we're in or outside the spotlight:

//We can retrieve the distance term by calculating the difference vector between the fragment and the light source and take that resulting vector's length.

//Then we include this attenuation value in the lighting calculations by multiplying the attenuation value with the ambient, diffuse and specular colors.

We also made the flashlight only appear if the player was holding down the spacebar key, by setting the use_flashlight to one if they were otherwise it would be zero. Then we checked if use_flashlight was equal to one then we would call the flashlight function else it would only calculate the global light and the player lantern.

The player lantern was very easy to make. It was almost exactly like the flashlight except it's starting position was slightly above the player and it's direction vector was aiming exactly down on the player. Then we just copy-pasted the code for the flashlight.

Result

We were very happy with how our lighting worked in the program, and we think it is very realistic.

Importing .obj file

Logic

Again we had no prior knowledge on how to import an .obj file but after watching many youtube videos and reviewing a lot of code and not succeeding, we finally realized that Kári had a video on how to do this from an old lecture back in 2019 and from there it was a piece of cake. We made the loadobj class exactly like Kári did, and implemented the mesh model, and material in our Base3Dobjects. The only thing missing from Kári's code was getting the uv coordinates. We then altered the loadobj class, to read vt values and put them in an empty list. Then we simply added the values to the mesh_model vertex and got them from the right place ([1][1]-1 and so on.....).

We drew our mesh model by calling it in a simple draw function, and gave the mesh_model certain locations, rotation and scale.

We added in the monsters and a flashlight for the player. We had to draw two monsters in level 1 and three monsters in level 2. The player always had a flashlight in front of him that rotated with him. So we had to define an angle and rotate that with the yaw, and then send it back to the draw function:

```
self.model_matrix.add_rotate_z(self.flashlight_angle)
```

Worked like a charm!

Result

We now have our beautifully, ugly, scary monsters and they are positioned throughout the maze. Be careful to not look at them!

Jump scare

Logic

We had to know when the player was watching the monster, where he was positioned on the map and how long he had watched the monster. So we had to find the area where the player could see the monster, we did this by trial and error, and then hardcoded the coordinates in and then we had to check if the players z vector was facing the same way as the monster was. If the above requirements are met, a counter starts, if the counter reaches a certain number, the monster “jumps” on the player and the game ends. If the player looks at the monster and then looks away the counter does not go back to zero, so be careful to not take a second look. The counter only restarts if the player dies, or if he clears the level.

Result

The jumpscare was tricky to have exactly right, and we were very close to the end, so it did not function perfectly regrettably. But we were still happy that the monster jumped at the player most of the time

Levels

Logic

We made two levels for the maze. The first one is rather easy to solve, it has only two monsters that can kill you and is much smaller than the second level. The second level however is rather big and very complex, it holds 3 monsters and they are much bigger and scarier. If the player clears the first level he is sent straight to the second level. To implement the levels we needed to add a lot of code and we basically set extra if statements for most of the code, f.x if self.lvl == 1: do this, if self.lvl==2: do this other thing.

Result

We did not experience any glitches between levels, no duplicate drawings, or such. But it added a lot of extra work and code.

Drawing the maze

Logic

We made two lists of lists in our __init__ function, where we hardcoded the coordinates of our walls, and their scale. Then we simply drew the walls by making a for loop and iterating through all the lists inside the list and assigned their individual coordinates and scale to their cube. Finally we added a little bit of texture to make them look realistic. Then we drew the floor and ceiling individually so they would not be collision checked.

Result

Our method of drawing the maze worked very well, but it took a lot of time. We could have made a more efficient way of drawing our maze, f.x. only drawing walls that were close to the player, but it was hard changing it when we realized that we could probably make it in a better way. We did not experience any lagging issues, so we were overall pretty happy with the final outcome

Gameplay

The game starts off with a tutorial screen that tells you how the controls work and how to start the game. Now that the game has started the player has to solve two dark and hard, scary mazes with nothing but a flashlight. However there's a twist, throughout the map we have a couple horrifying and blood hungry monsters waiting for you if you take the wrong turn, but due to their blindness you might have a couple seconds to run after you see them and escape death's grip, but death's grip can have many ways of getting to you in our maze for example what your foot and make sure you don't fall to your death. Get to the other side of both mazes alive and you win. If and only if the player wins he will be offered to either take their winnings and quit the game or double down and go for round 2 in our terrifying maze.

Why some features are missing according to the rubric

We have fulfilled most of the rubric and are very happy with what we have made, however we deliberately skipped a few things that were on that rubric because it simply did not fit in with our game. We were asked to make a overhead camera or a minimap of somesorts in the rubric but our game is meant to be scary and we did not want to show how the map layout is designed and were the monsters are hiding and waiting so in our case we thought it would be best do completely ignore the overhead camera. We also kind of disregarded the chase camera because it in a way does not work in the map that we created due to the fact that the maze is very narrow and has tight corners, we did however make it so the player can increase his FOV to the extent that it looks like a chase camera of some sort. The last thing that we dismissed that was on the rubric was the collision with a weird object which was simply unnecessary , we instead invested our time in getting our monsters right and texturing the whole map and all of its contents.

Final words

We have spent a ridiculous amount of time on this assignment(ca. 100 hours), but we didn’t mind. This assignment was very challenging but very rewarding. We are very happy with our final product, even though it would have been nice to set up different camera angles, to get those points in the rubric. This is by far the most fun assignment that we have had to do in RU and has really caught our interest in making video games.