Documentation: Gabriel A. Santiago Plaza's Game

Indices

Introduction:

I would like this documentation to be more of an informal investigation paper on the process of making this videogame. In each of these sections I will discuss the challenges, and goals of each part on the game. I would also like to add lot of the work that I did but it is not present in the game itself, instead it is commented (and there is a lot more content that I removed) since I believe that the purpose of this class and this project should not be about making a perfect product, but to learn by mistakes and understand most of the concepts presented.

The Map:

The first part of the project was to make a map where the player will be in. This map can be represented as a 3x3 grid, each grid as "plane*", where * is their location and *x* and *z* are the center of each 'box' in respect of the scene. Like so

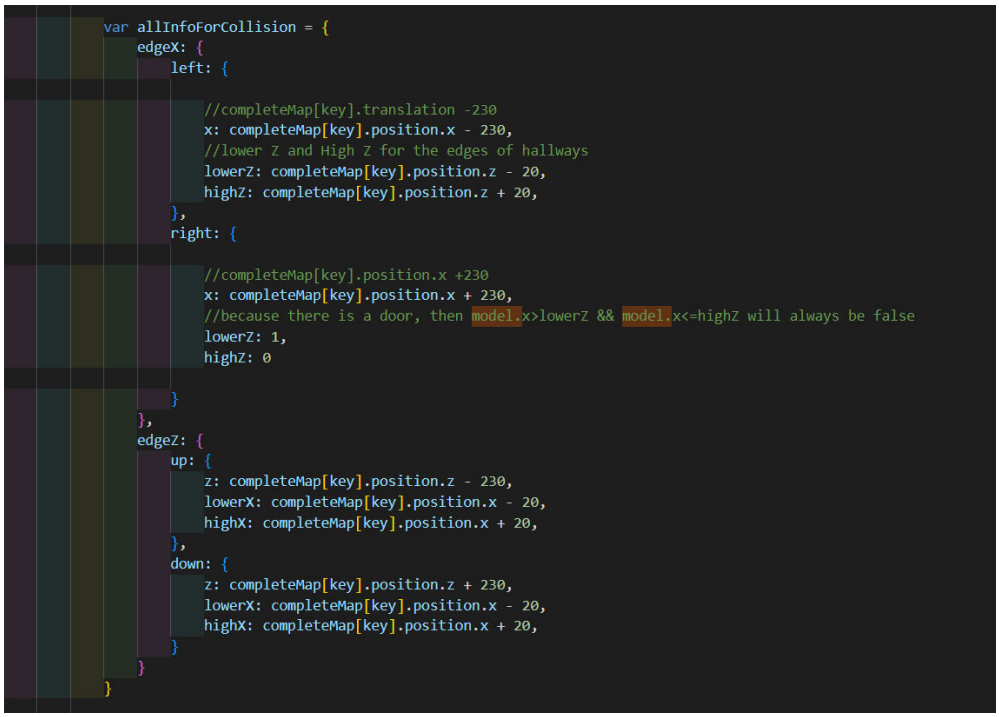
planeTopLeft x = -500 z = -500	planeTop x = 0 z = -500	planeTopRight x=500 z = -500
planeLeft x = -500 z = 0	planeCenter x = 0 z = 0	planeRight x=500 z =0

planeBottomLeft x = -500 z = 500	planeBottom x = 0 z = 500	planeBottomRight x=500 z =500

And each of them were a copy of planeCenter, except with a door on the edges so the player would not fall of the map. So planeTopLeft would have a door at the z = -750 and at x = -750, with the rotations necessary. Each room has a rectangular PointLight on the ceiling, sort of like an actual square ceiling light, they have the walls, and floors. Since most of the walls, lights and ceilings were the same, I made them all inside two forloops, see in createMap function at like 826. Inside this same function, I made a switch case for the doors, depending on what plane the loop was on. Each of these new planes after their creation were added to the scene, and positioned on the scene accordingly, like the figure suggests and added to other Data Structures. One array called *allGeometries* , that is going to be used for raycasting and wall collision, and a dictionary called *uuidMapping*, that takes the unique id of each plane, and uses it as a key to pair another dictionary called *allInfoForCollision* that creates all the boundaries of each plane. *allGeometries* and *uuidMapping* are going to be explained a bit later in Wall Collision, but I would like to discuss *allInfoForCollision* first.

For this Data Structure, I had to decide which were going to be the edges of X and the edges of Z in each plane. If we would be on the center of any plane, the walls are located at x= 250, x=-250, z=250, and z=-250 in child coordinates. In scene coodinates, any wall would be located on those same coordinates, but you would have to add the center coordinates of each plane. To make myself clear, planeBottom has walls on x = 250, and x=-250, but in scene coordinates, the walls are actually located on planeBottom.position.x -250 and on planeBottom.position.x +250, where position.x is x position of the center. Now, since the walls were thick, and for the raycasting I did not want the caracter to literally touch the walls, I set the edges instead of 250, to 230, so it would be 20 ‘points’ away from the actual wall center.

This was not enough, since I had small hallways for the map, I had to add more to my edges. Lets look at planeRight’s *allInfoForCollision*:



```
var allInfoForCollision = {
    edgeX: {
        left: {

            //completeMap[key].translation -230
            x: completeMap[key].position.x - 230,
            //lower Z and High Z for the edges of hallways
            lowerZ: completeMap[key].position.z - 20,
            highZ: completeMap[key].position.z + 20,
        },
        right: {

            //completeMap[key].position.x +230
            x: completeMap[key].position.x + 230,
            //because there is a door, then model.x>lowerZ && model.x<=highZ will always be false
            lowerZ: 1,
            highZ: 0

        }
    },
    edgeZ: {
        up: {
            z: completeMap[key].position.z - 230,
            lowerX: completeMap[key].position.x - 20,
            highX: completeMap[key].position.x + 20,
        },
        down: {
            z: completeMap[key].position.z + 230,
            lowerX: completeMap[key].position.x - 20,
            highX: completeMap[key].position.x + 20,
        }
    }
}
```

I defined edgeX to be the edges on the X axis, both left and right, and edgeZ on both up and down. The left side of planeRight, has a hallway on the left side of the plane that connects with the planeCenter. So I created the variables lowerZ and HighZ that represents the space of that hallway, will be explained later in wall collision.

Camera + movements:

I just read that I have to keep this simple and short. I used pointerLockControls and followed the moveRight and moveFoward, along with some examples that I cannot give links to because LUMA took away all power again. But I used polling to move the direction I wanted.

Wall Collision:

I did not know what a raycaster was, and learned basically that **it is a tool that casts a ray from a given point, towards a direction, and when you call the function intersectObjects(array, true/false) it will return the objects in the array that were intersected. If the second parameter was set to true, it will check their children.** I used this to cast a ray directly underneath the player (later after so much work, I did the same for the enemies), this will give me one object, that represents which plane I am in, I take the uuid, and use uuidMapping to check for wall collisions. The uuidMapping[uuid] will return the specifications of the edges on X and Z, with the hallways, and then I used this code, to prevent the player from going beyond walls

```
//first check for x
var edgeX = uuidMapping[uuidForMap].edgeX;
var edgeZ = uuidMapping[uuidForMap].edgeZ;

if (camera.position.x <= edgeX.left.x && !(camera.position.z >= edgeX.left.lowerZ && camera.position.z <= edgeX.left.highZ)) {
    camera.position.x = edgeX.left.x;
}
else if (camera.position.x >= edgeX.right.x && !(camera.position.z >= edgeX.right.lowerZ && camera.position.z <= edgeX.right.highZ)) {
    camera.position.x = edgeX.right.x
}

if (camera.position.z <= edgeZ.up.z && !(camera.position.x >= edgeZ.up.lowerX && camera.position.x <= edgeZ.up.highX)) {
    camera.position.z = edgeZ.up.z;
}
else if (camera.position.z >= edgeZ.down.z && !(camera.position.x >= edgeZ.down.lowerX && camera.position.x <= edgeZ.down.highX)) {
    camera.position.z = edgeZ.down.z;
}
```

Where the first condition checks if im close to the edges of the plane and the second condition checks if the player is **not** between the lowerZ/X and highZ/X, because if they are, that means they are going through a hallway.

Later I realized that I could have made a raycaster go to the direction that I wanted to move, and if the distance between the camera and the wall was very close, don't let the camera move, but raycaster was foreign for me at the time.

HUD:

I was introduced to THREE.Sprites, where it was basically something that was always facing the camera. For the healthbar, I drew in paint 10 images representing the healthBar, then created the SpriteMaterials and stored them in a healthBars dictionary, where the key was the currentHealth of the player, and the value is the SpriteMaterial that is going to be the healthSprite.material. This healthSprite variable, when the material is changed, the health on the screen will be changed. For example in this code, that the enemy punches the character

```
punch() {
    if (this.hitting == false) {
        this.hitting = true;
        this.moving = false;
        this.animationMixer.stopAllAction();
        this.animationMixer.clipAction(this.animationFrames[5]).play();
    }

    if (model.clock.elapsedTime - this.lastHit >= this.hit_rate) {
        this.lastHit = model.clock.elapsedTime;
        model.currentHealth -= 10;

        if (model.currentHealth >= 0)
            healthSprite.material = healthBars[model.currentHealth];
    }
}
```

When they hit me, my health depletes, and it changes the material. The images can be found in moreTextures.

In this HUD, I also added the animations of the gun firing, which I divided the fire_rate in 3, to represent the time each frame is going to be shown. So for example, if the gun has a fire rate of 1, then each image of the gun will be shown for 0.33 seconds, the first image being the initial fire, the second one being the recoil and the third one being going back to the initial spot.

Enemies:

So the enemies were complex, initially there were going to be sprites, Doom Sprites to be exact. In moretextures, there are images called enemy_*_editted.png that were the sheets that I was going to use for the animation for them to walk towards me. I was having fun, I got them to move, I created an EnemyController Class that controlled the pngs and the offsets of the animation sheet using uv coordinates and stuff, but the raycasting, or shooting at enemies, was giving me a very hard time, so I decided to drop that and add a GLTF instead as the enemy. Later I realized that the sprite wasn't the one giving me problems, it was actually the PointerLockControls, so the shooting raycaster had to be done differently. Instead of using mouse coordinates, I have to use the direction matrix and the camera position to shoot the ray.

Now it is a simple robot with programmed animations, that runs, (I programmed the movement, and their wall collisions), I gave them attributes such as health and hit_rate. The movement was particularly challenging since the raycasting did not want to work, but basically I programmed that if they are not close to me, or the camera, then they run towards the forward direction. When they are close to me, play the punch animation and start punching me. When they die, it has a small chance of dropping something, either bullets or health. The player collects it automatically.

Usage:

The player moves at WASD, reloads at R, mouse moves the camera and shoots at left click. Be sure to open it inside a Web Server

Conclusion:

There is a lot of stuff that I did not add to this documentation because of the limited pages, but on the presentation I can give details and rant about every single problem I found, but overall, I have learned a lot about 3D programming.