3D Game Programming: Maze FPS Template

Andres Ponce 0616110 andreseeponce@gmail.com

December 22, 2019

I CLAIM THAT THIS IS MY OWN WORK: YES

1 Introduction

The purpose of this assignment was to work on a larger project, and to have some experience working with larger systems and projects. In this project there exists many more moving elements, such as: bars, weapons, sounds, individual meshes, etc...

Creating an FPS game requires calculations on multiple objects, including the sphere, the monsters with each other as well as the player, and finally calculating the physics of the bullets. For each of the objects and systems, every time a frame is rendered, different calculations on their systems have to be updated. A project this large requires the different parts of the game be aware of each other, since their reactions depend on properties of other systems.

Word Count: 117

2 System Architecture

The main system or object which contains all the other ones remains the `TutorialApplication` object. Here we "join", or we manage the different systems. We define all the primary Ogre functions such as `frameStarted`, `createCamera`,etc... This object acts as our anchor point for all the other systems. For all of these, we keep in mind that we need to update the position/state or the player character, the monsters, weapons, bars, sound,...all of which are different classes.

Some of the objects include: main_char, weapons_manager, sound_manager, bars, digitCounters, mesh_manager, particle_system, and read_data. All of these contribute to the overall appearance and behavior, however they are too complex to be defined in a single file, so we have to define their own header and `cpp` files.

Word Count: 134

3 Methods

The main issue that was experienced remained being the physics calculations. We first fix the viewport, since originally the viewport was set to the side. Then, we enable the exponential fog, which is enabled in `read_data.h`. This file then reads the `0616110_game_data.txt`, which defines several of the parameters we use in the scene. The focus then is on creating the scene with the fog, skybox, and the terrain.

Then we proceed to create the main character, which originally uses the `robot.mesh`. According to the specs, the robot is not visible in the viewport 0, so we use `vp->setVisibilityMask()`, and only objects with the appropriate mask will show on the viewport. We also need to keep the main character "stuck", or "clamped" to the terrain, which is handled in the `clampToEnvironment()` method. Then, the position will be changed accordingly depending on the terrain. at the given point. Since this is an FPS game, we have the camera of the main viewport centered around the main character at an offset, and the direction the character is facing becomes the direction of the camera.

When we want to shoot the spheres, we need to calculate the sphere's trajectory. We need to set the velocity according to the equations, and calculate the velocity and position after there has been a collision with the ground or the large sphere. The velocity and energy decreases by a certain amount and the resultant velocity sends the ball in a different direction.

After we calculate the the physics for the spheres, we need to set up the effects on the large sphere. We need to use the particle system to set the "explosion" effect on the large sphere whenever a projectile collides with the large sphere, we also play a sound effect using the `SOUND_MANAGER::getInstance()->play_Explosion()` method.

The next part, and one that was quite challengnig to understand was, which involves using the `WAGO_OGRE_MESH_OBJ::crea` method. This essentially opens a file with some color values. We loop over the units of the picture. Depending on the color values of the image, we can create some of the vertices. There was some difficulty in understanding the process of the code, but we ultimately are creating new vertices based on the colors of a different mesh file.

For the second task, we merely create a new camera which is placed above the main_char. However, to avoid gimbal lock, we add a small offset to our `lookAt()` method. We aim it at the `cpos` object as it is referred to in the function, and we update its position every time a frame is started. When we press `Z` or `C`, we add or subtract from the camera's Y coordinate.

For the third task, we merely ensure that our `read_data` object is reading data correctly from the file modified with our student ID. In my case, this file is called `0616110_game_data.txt`. Since the object already read data from our file properly, then we only changed the name in the `#define` directive in the `read_data.cpp` file.

For the fourth and final task, our job was to implement the bars and the digit counter. A problem I had with the counter was adjusting the position of it within the viewport, becuase changing the `dx` and `dy` values only changed its initial position in the viewport. For the two bars, one representing the energy level and the other representing the speed, we first created a bar and then split it into two sections. Then, based on the current energy level relative to the maximal energy level, we set the proportion of the total bar to be taken by one side. a call to `setInfo(1,2)` would set one half of the bar to one color and similar for the other half.

Word Count: 668

4 Discussion

This assignment proved a useful experience in dealing with a program that spans multiple files and interacting systems. Even for a relatively simple game, the amount of code and problem solving, special considerations, and patience while debugging needed to produce a worthwhile game is immense. Diagnosing a problem can take a long time, especially if the error resides in another location than originally thought.

For me personally, handling the game physics proved to be one of the most challenging aspects of the assignment. Calculating rotations, velocity, gravity, acceleration, etc...took me longer than anticipated, and sometimes it was not even possible to fully fix. An example would be the rotation of the penguins while they follow the player; when trying to use the `lookAt` function on the penguin node, and having it face the main character, it caused the penguins to rotate on other axes as well. Then, when trying to implement the rotation using `mSceneNode->yaw()`, there seemed to be an error involving the angle between the two vectors, as the penguins kept on rotating. These issues highlight that different aspects of daily life that we take for granted in daily life, such as turning our heads towards a new sound, all rely on precise movements which can be even complicated to describe mathematically.

There was always one bug that was hard to solve which involved the playing of sounds. The game would run fine, but after a sound had been playing for a certain amount of time, the game would crash. This remains confusing because after the `SOUND_MANAGER` initiates, there is a static instance which handles playing the sounds from predefined sound arrays. The demo program also contained some of these issues on occasion. Also, the `data_read` class lacked some of the required functions for retrieving some of the sound items. However, there are some that still would not play from the sound file folder. The levelUp sound effect had this particular effect.

Programmers working on large projects should therefore feel comfortable interacting with different systems. The intricacies of these systems require special attention on behalf of the writer, since a mistake in one area can lead to very serious consequences in other, seemingly unrelated areas. Since my physics ability is not incredibly high, this had a particularly harsh effect on this homework assignment. Even finding angle between different objects proved to be challenging, as well as collision detection. **Word Count: 407**

5 Conclusion

To conclude, this homework assignment allowed practice with more complex assignments, and showed that games that are more involved usually involve much more effort to put together as well. Every system sometimes requires multiple classes to work well together, and also if one wishes to add new meshes or terrains, as seen when using the `wago_game_obj`. We also have to take into account how our game interacts with the hardware in the end, when loading custom meshes or terrains, how to most efficiently handle collisions, etc...

This experience has been very helpful in seeing the intricacies of actual software engineering, and I am grateful I got to practice that skill, even if there were still many aspects which were left lacking. **Word Count: 124**