ICS4U1 Culminating User Manual

Introduction

Our culminating is a 3D first person shooter video game. The game takes place in a dark underground maze. The player is supposed to travel around the maze using keyboard input, and shoot incoming enemies using mouse input. The enemies have their own walking, punching and dying animation. They always take the shortest path towards the user. There are collision detection for the player and enemies; they can't walk through each other or walk through the walls within the map. Additionally, there are texts that display statistics such as health, ammo count, elapsed time and kills. Periodically, ammo packs and health kits spawn on the floor, so that the player can pick them up.

Evidence of Dealing with Possible Errors

We have given small test cases to the algorithms of our program as well as playing through the game ourselves. Additionally, we asked other classmates to try the game in order to discover bugs within the program. As a result, we were able to detect and fix bugs. Many of the bugs were described in the journal entries.

Running the Program

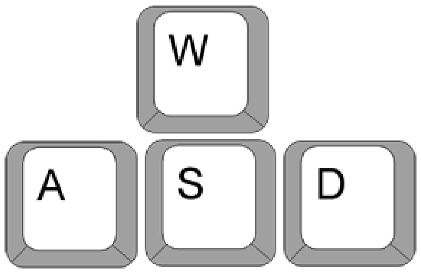
You can run the program via mainMenu.java, this class launches a javafx window, which asks the player to type their username. The game begins when the player clicks the "start game" button. The button launches the actual game and it usually takes around 10 seconds to completely load. A short video showing the actual gameplay can be seen here. Note that the video is meant to be a brief demonstration. There have been a few minor changes to our program after the video was taken.

Objective of the Game

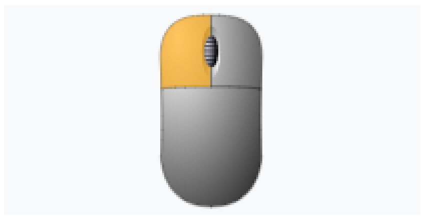
Within the game, the player is supposed to survive for as long as possible by battling enemies (stormtroopers) on the map. The player can also pick up ammo pack and health pack that spawn on the floor to restore their ammo count and health, respectively.

Controls

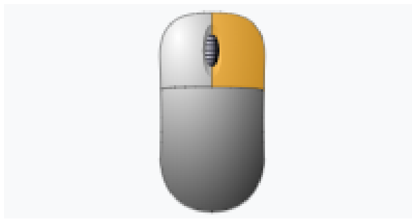
The player can attack and move around the map by using a combination of keyboard and/or mouse input.
The input(s) are defined as follows:



The W-key moves the player forward.
The A-key moves the player leftward.
The S-key moves the player backward.
The D-key moves the player rightward.

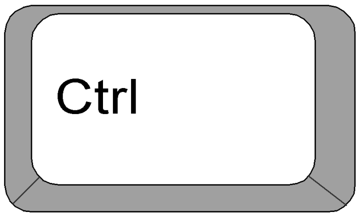


The left mouse button shoots a bullet towards the direction the player is facing when it’s clicked.

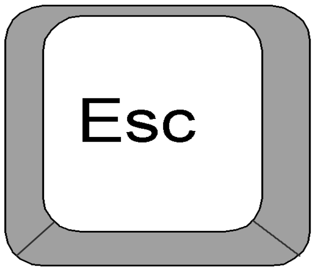


The right mouse button reloads the gun when it’s clicked.

The player’s view follows the mouse movement.



Holding the left ctrl key along with W-key, A-key, S-key, or D-key allows the player to sprint in that direction.



Exits the program during the game.

Evidence of Research to Solve Problems

-Found the inverse of a 4x4 matrix using Gauss-Jordan elimination by visiting this website. (https://www.geeksforgeeks.org/program-for-gauss-jordan-elimination-method/)
-Looked through the web to find libraries for displaying text in lwjgl 3. Didn't have any luck finding one, since many libraries and methods are outdated. Acquired an idea for displaying text through this article. (https://learnopengl.com/In-Practice/Text-Rendering)
-After getting a font texture, I looked on youtube to figure out how to render textures on lwjgl 3 (https://www.youtube.com/watch?v=crOzRjzqI-o)
-For loading the animations, I referenced (www.youtube.com/watch?v=HyVvi-TjHIM) for some help.
-For the objects that are included in the program, credits go to Rasul Hasan for the Stormtrooper object.
(https://sketchfab.com/3d-models/stormtrooper-obj-1c63624047634e58970f246dbbc9d332)

Analysis of Program

The main loop of the program runs under a "while" loop, which continues running until the game is closed. With openGL, there's no direct way of controlling the exact frames per second for the program. However, an average of 60 frames per second is specified by "glfwSwapInterval(1)". Therefore, I may assume that the average frames per second is around 60.

Before the Main Loop

Before entering the main loop, the program first handles key input and mouse input. Those methods only get called when a key is pressed or the mouse is clicked. Next, the walls of the map are loaded via a "txt" file. This is used for all the collision detection later on in the code. After that, the image textures are loaded. In fact, the image texture process takes a relatively long time. This is because for each texture, the original image gets broken down into pixels. For example, a picture resolution of 256×256 would result in a total of 65,536 pixels; storing the RGBA values for each pixel results in 262,144 values. A total of 6 image textures are loaded, leading to a total of 1,572,864 values computed. Finally, the majority of the loading time is spent loading the enemy "obj" files. There are a total of 360 files to load, and each file has close to 20,000 lines. This leads to almost 7,000,000 lines to process. This is the major reason why the program takes close to 10 seconds to initialize (start up).
Conclusion: O(C), C = A large constant value around 8.6 million

Displaying Text on LWJGL 3 Window

Finally, inside the loop, the program first displays the text. For each letter inside the text, I looped through the width of the texture (text.png) to find which pixel the letter is located. At the top of (text.png), there is a blue pixel a yellow pixel that indicate the start and end position of every character.

The characters inside the texture are in order of ASCII value. Therefore, locating the character takes O(n), where n is the width of the texture. After finding the start and end position, displaying the text only requires adding that portion of the texture onto a “glQuad”, the time complexity is close to constant.
Conclusion: O(n), where n represents the width of the texture in terms of pixels

Keyboard Input

After displaying the text, the program handles the keyboard input. As previously mentioned, whenever the “WASD” or Control keys are pressed, the values are changed to “true” inside the boolean array. After storing the values, I physically move the user around. The user is only moved if the new spot is not a wall and is not occupied by an enemy. Also, the program includes a feature: when the user is indirectly facing the wall, the method moves the user into the direction that’s not into the wall. In other words, without this feature, if the user is turned 1° towards a wall, he/she would not be able to walk in any direction. This feature makes keyboard movement a lot smoother, and prevents the user from being stuck beside a wall or in a corner. The only part of this function which takes up non-constant time complexity is when checking if the position overlaps with any enemies. Looping through all the enemies results in a time complexity of O(n), where n represents the number of enemies.
Conclusion: O(n), n = # enemies

Bullet

The method for handling bullets only runs when the left mouse button is clicked. I designed this method so that when a bullet is fired, the impact is instant. As a result, each bullet path is tracked until it hits a wall or an enemy. I set the distance that the bullet travels 0.2 units each time. A “while” loop is used to track the path. In its worst case, the bullet must be tracked for the longest distance inside the map that doesn’t pass through any walls. The longest straight path would be the longer of the width and height; with the bullet travelling 0.2 units, the total number of computations is 24/0.2=120. Thus, it’s safe to say that the while loop would have a maximum of 120 runs before breaking.
Conclusion: O(n), n = height of map [24]

Ammo and Health Packs

Next, the program creates ammo and health packs that randomly generate around the map. The packs are generated every 5 seconds. There is a limit of 3 ammo packs and 3 health packs, so that the map doesn’t get overcrowded with packs. First, the program finds all the spaces available for packs to generate. This eliminates packs from generating in walls or spaces already occupied with packs. Eliminating the occupied spaces takes O(n) time complexity, where n represents the number of empty spaces inside the map, which is most likely to be around 100. Also, every space already occupied with a pack is removed from the ArrayList. Theoretically, this process would take another O(n) time complexity, but since there can be a maximum of 6 packs in total, this value can be treated as a constant.
Conclusion: O(n), n = # spaces in map [100]

Drawing Objects and Enemies

Each drawing method draws the faces and vertices, while mapping the texture onto the faces. The performance of this part heavily relies on the number of faces and vertices of the object. For example, the gun has over 30,000 lines, leading to around 10,000 faces. In addition, each enemy has

over 16,000 lines to process. If the program ever slows down or lags, the major cause would probably be drawing all the objects. I've tested the duration taken for all the objects and enemies to load: the time varies, but it can take up to 0.2 seconds (if the time exceeds 0.17 seconds, the program starts lagging).
Conclusion: O(c), c = a large constant (around 80,000)

Add new enemies

Every 15 seconds, a new enemy is added. Also, a maximum of 4 enemies are allowed at one time. As explained before, more enemies may result the program to slow down significantly. Also, enemies are not allowed to generate right next to the user. Thus, it's not guaranteed that the first chosen space is available for generating. The program keeps on taking new spaces until it's available for generating. This process results in a worst-case time complexity of O(n), where n is, again, the number of empty spaces inside the map.
Conclusion: O(n), n = # empty spaces inside the map [100]

Removing Ammo and Med Packs

Whenever a pack is generated, the user can walk over to the pack and use it to boost their health or ammo count. After using the pack, the program must remove that pack from the ArrayList tracking their positions. This process must check every pack there is inside the map. As previously mentioned, there are a maximum of 6 packs at one time. Thus, this process can be summarized as O(n), where n is the number of packs present at that point of time.
Conclusion: O(n), n = # packs inside the map at the time [6]

Update enemies

Finally, each of the enemies is displayed. Inside the "for" loop for all the enemies, the program checks if the enemy is still alive. If not, all there is to worry about is completing the death animation. The death animation check takes constant time. Otherwise, updating the frame of the enemy, the position, and the rotation takes constant time as well. Drawing the enemy should be constant time, but since the enemy has a lot of faces and vertices, it takes quite some time to draw out. Finally, if the enemy is looking for a path to the user, BFS is performed. The time complexity of BFS in this case is O(m×n), where m and n represent the width and height of the adjacency matrix, which in this case are 24×21. Lastly, removing the dead enemies takes O(n) time complexity, where n is the number of enemies there can be at one time. However, since the BFS part has a higher time complexity, the final complexity would not include this part.
Conclusion: O(a×m×n), a = # enemies [4], m = width, n = height