Dungeon Game

Roberto Ockerse Hernandez

A. Background

I Created a dungeon crawler game with different enemies, traps, consumables and objects, the player must have a health bar and an ammo count, there is consumables that the player can use to regain stats, Traveling through dungeons should be seamless and going back to previous dungeons will render the items inside it with the exact same status and positions that they were left with when the player left the dungeon, a map of all the dungeons should be able to be render that will render the exact position of all the dungeons.

This program is made using OpenGL 4.1, GLFW and GLEW for easier use of the OpenGL libraries and it used GLSL shaders for rendering all objects

B. Design Principle

The main requirements for this game is for all the objects within the game to keep their status until death, this means storing the data of all items into a dungeon and updating their data the moment you move away from that dungeon, store it for later use once you return to that dungeon, a way I got around to do this was using double linked list to store the data of all items each dungeon has, a double linked list is used per each type of item for each dungeon node to store the items when leaving the dungeon or to retrieve items when entering the dungeon once more.

To be able to seamlessly travel from one dungeon to another dungeon and being able to come back from where you came from seamlessly what I is I made the dungeon crawling aspect of the game into a graph. Every vertices node in the graph will contain a dungeon object with all of its data stored, every vertex will have a maximum of 4 edges (top, bottom, left and right) which it will use to travers through the many dungeons in the game, this graph will be also used to render a map of the dungeon using a breath first search of the graph, in later stages this program will randomly generate a dungeon with random items and random enemies dropped all around, the goal will be to reach the end of the map or the last edge created, because of this I decided to use the breath first search of the map so that the map may be rendered no matter the size or positions of the graph, so that the player can easily see where they are located.

C. Model Discussion

Figure 1

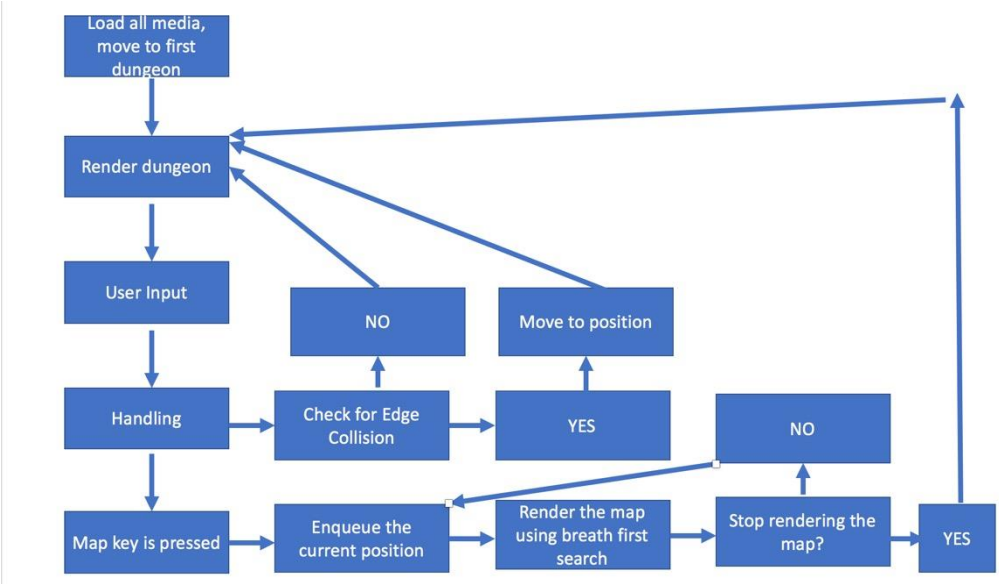


Figure 1 shows a very in a nutshell flow chart of the program, first we load the media of all object sprites, that is the white squares for the mummies, green circles for slimes, etc. The next step the program creates the dungeons within the game and place the enemies, traps and crates in their corresponding dungeon, at this points the dungeons have not been connected with each other so that is what we do next, we connect the graph in a way that every vertex has at least one edge and at maximum it will have 4 edges, this is because the first dungeon needs at least an edge to go to and all other dungeon need an edge to go back to the dungeon the player came from and of course since there are only a TOP, BOTTOM,RIGHT and LEFT edge the dungeons should not connect to more than 4 edges. This edges will be placed in one of the built-in array that is contained within each vertex node (dungeon node) this built in array has 4 slots that are numbered as such

- TOP = 0 index
- BOTTOM = 1 index
- LEFT = 2 index
- RIGHT = 3 index

This allows for easier rendering of the edges in the play area, allowing the user to better understand his location in the game, the graph is made in way so that if

there exist an edge in the TOP position in vertex A that travels to vertex B, there will be an edge in the BOTTOM position in the built-in array of vertex B that will connect with vertex A, this allows for easier transition between vertex that make the transition from one vertex to another to be that of an Adjacency Matrix since we can look directly if there exist a an edge in a certain position in the vertex node by just checking if there is an edge at the top position for example, this is how the graph list should look like.

Figure 2

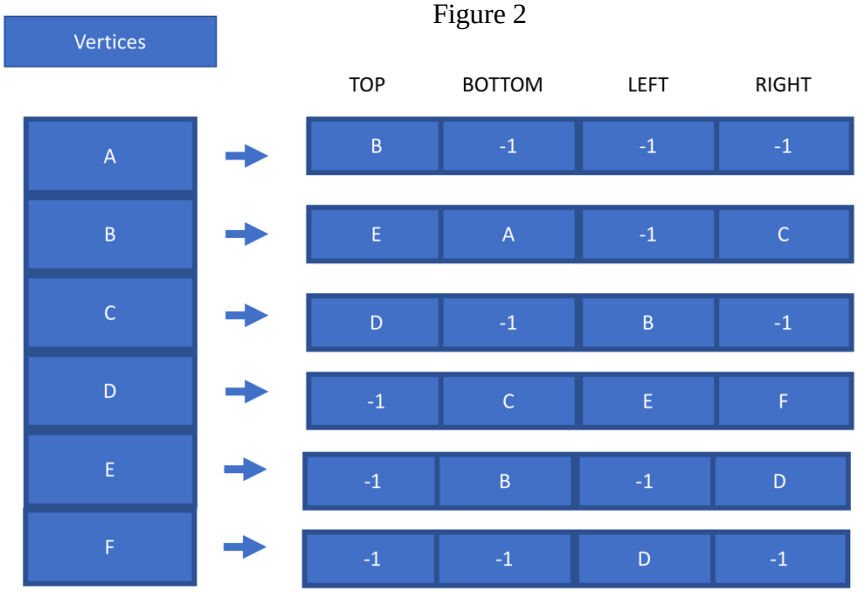
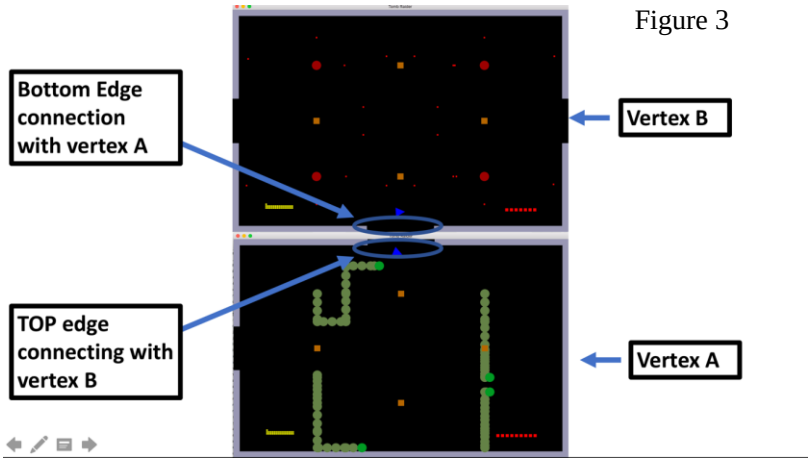


Figure 2 shows how the graph works in the program itself I used integers for ID instead of letter but for easier understanding I change the integers to letters, as you can see if there exist and edge on a vertex A that connect to vertex B, then vertex B should have a vertex that connects to vertex A on the opposite direction here is a picture as an example:

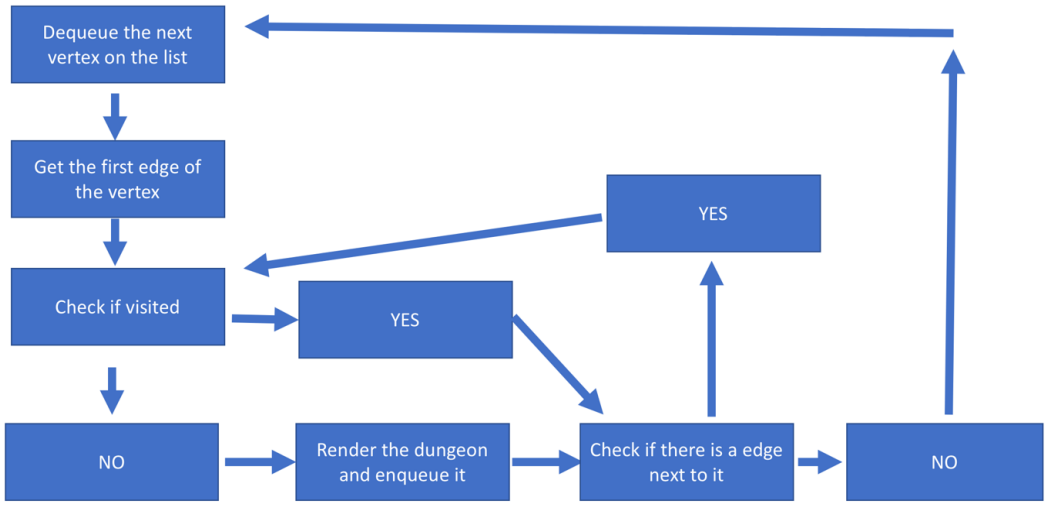
Figure 3



As you can see vertex A has an edge at the top that connects with vertex B, therefor vertex B must have an edge at the bottom that connects with vertex A.

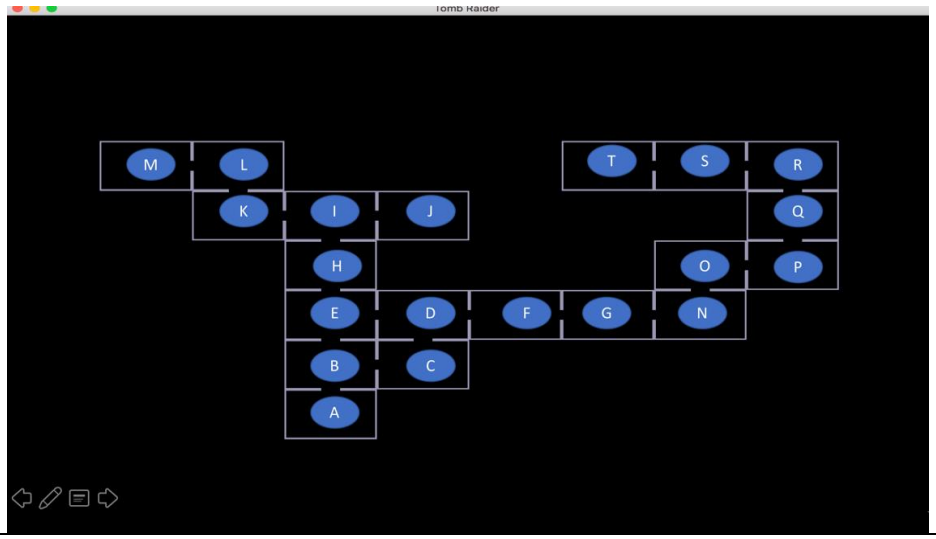
Lastly using this same graph, I was able to render a map of all the dungeon nodes using breath first search, I used breath first function to be able to go back to the parent node and get its edge coordinates(weather is TOP, BOTTOM, LEFT or RIGHT) and use this to find the correct coordinates to render the dungeon node in the map without this when the function needs to render a new dungeon area it will first have to find where this node was first found in the graph to figure out the correct coordinates to be able to render the correct position, the flow chart below shows the process of the breath first search dungeon rendering.

Figure 4



We enqueue the first vertex into a queue then quickly dequeue it and get the first edge of the vertex if it has been visited then check if there is an edge next to it if yes keep checking and going deeper into the queue until we find one not visited, when we found one not visited we enqueue it and render its position based on its parent position, then we continue to check if there is an edge next to it if not then we dequeue the next vertex on the list and repeat.

Figure 5

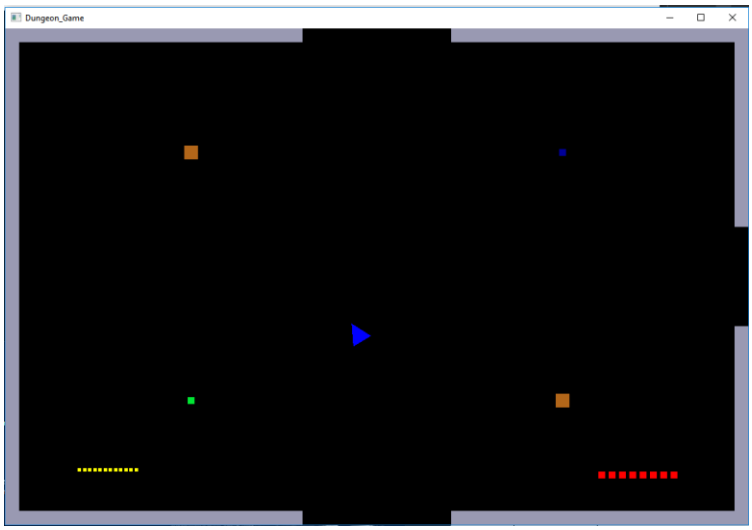


D. Demonstration

This is a Dungeon crawler game where you control the player using WASD to move and right click to shot at enemies there are 2 types of consumables which are

- Health box – a green colored box that will heal you, which drops from crates randomly
- Ammo box = a dark blue colored box that will give you ammo, which also drops from crates randomly

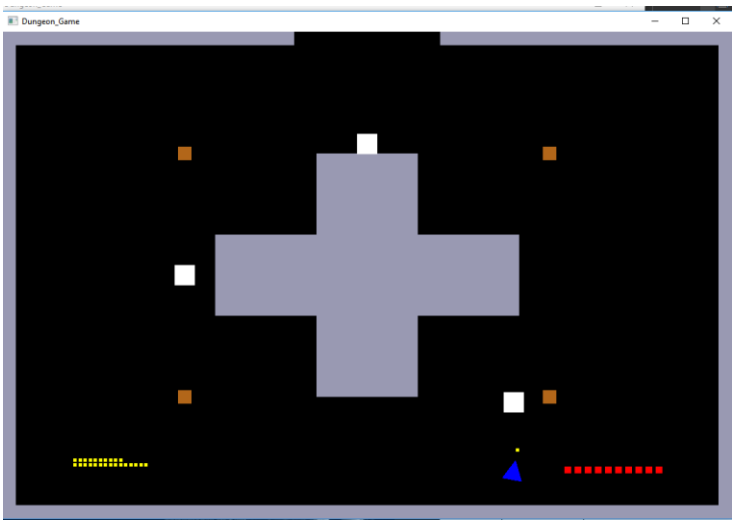
Figure 6



There are three types of enemies in the game which behave differently they are

- Mummy – White colored box that will not move until it has a clear path towards you, then the mummy will leap and try to attack you, they are fast but will not go after you if behind a wall.

Figure 7

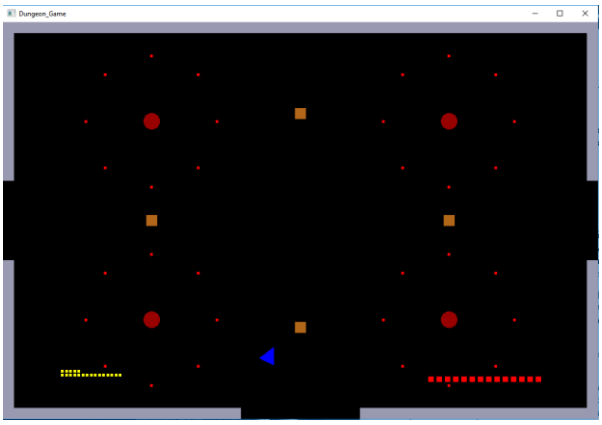


- Slimes- Green circle that will move in random directions dropping goo as they go this goo is a trap and will slow the player down if touched, the slime does not walk in any clear patter but has a lot of health and his hard if touched.

Figure 8

- Pharaoh – Red circle enemies that will shoot 12 bullets at regular intervals this bullet will be shot around the pharaoh at 30 degrees angles

Figure 9



There is no ending to the game yet but I have a plan to add a treasure somewhere randomly in the dungeon and right after I created a way to randomly create a different dungeon every time the game start, making the gameplay different, every time you play.

A. Conclusion

This dungeon crawler game uses double linked list and graphs to seamlessly move around the dungeon with zero data loss all items consumables and enemies will stay where they were upon returning to the dungeon, thanks to the double linked list which we manipulate to update the items in a dungeon upon leaving

the dungeon and getting the items from a dungeon upon entering it to handle them , it uses a graph to travel between dungeon this graph is able to keep track on the dungeons positions in way that we can also use this graph to render a map of the whole dungeon using breath first search.

One last thing if you are going to compile the program remember that the program need to be have the libraries of GLEW GLFW and OpenGL linked for it to work I could write how to link it but it varies from compiler to compiler, also it need to link with the glm headers that are included in the Dungeon_Game folder