NightSimpleMaze

Thanik Sitthichoksakulchai - 1804397

NightSimpleMaze is a simple first-person maze game that the player has to collect all the items and get to the exit flag.

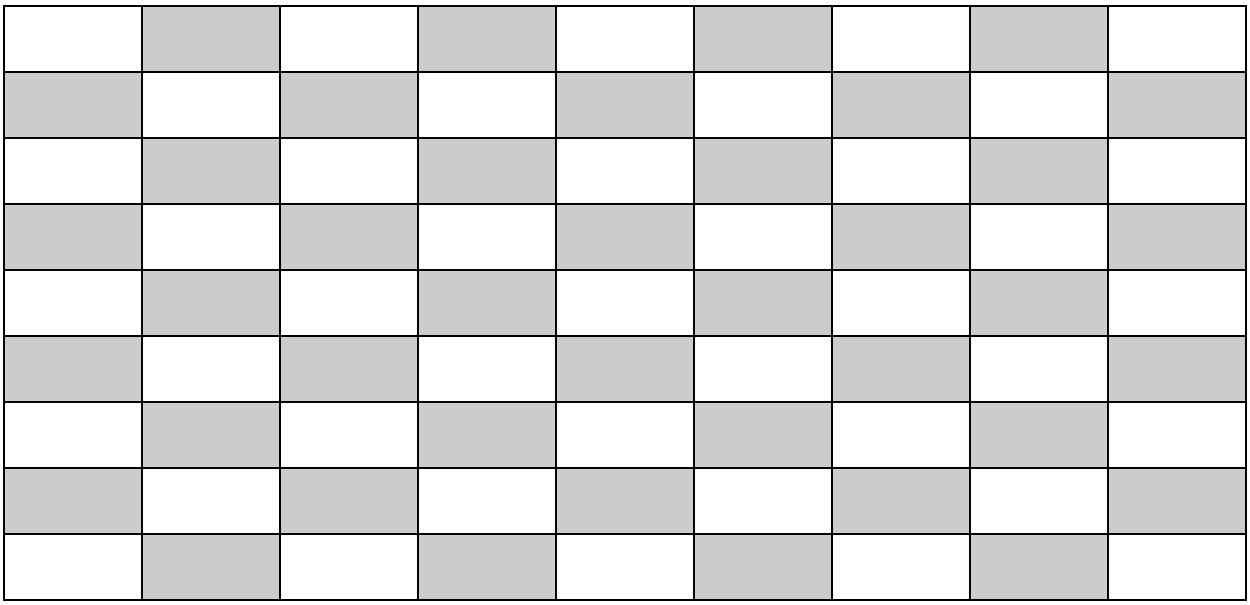
Controls

- Use the mouse to look around
- Use W, A, S, D key to walk (in the first-person view) and to move the camera (in map view)
- Use M key to show top-down map
- Use Up Arrow Key to zoom in (in map view only)
- Use Down Arrow Key to zoom out (in map view only)

Features

Maze Generation

The maze is generated by using the depth-first search algorithm. Starting from a 2-dimension array of booleans acts like a grid, the value of the boolean indicates the cell in the grid is a wall or not. The array size is ((maze width * 2) - 1) * ((maze height * 2) - 1) because each cell has its 4-directional walls.



An array of booleans visualized as a grid for generating maze. (white as a normal cell and grey as a wall)

Mike Gold (2007) described the depth-first search algorithm steps as
“1. Pick any random cell in the grid.
2. Find a random neighboring cell that hasn't been visited yet.
3. If you find one, strip the wall between the current cell and the neighboring cell.
4. If you don't find one, return to the previous cell.
5. Repeat steps 2 and 3 (or steps 2 and 4) for every cell in the grid.”.

From Mike Gold’s description, some adaptations will be done like an unvisited cell or a wall has a boolean value as false and is set to true when the cell was visited or the wall was stripped.

After the maze is generated within the generator’s grid, the walls will be spawned in the game level based on the grid data. The player start point, a goal flag, and collectible items will be randomly generated later. If there’s a position conflict for these objects in a grid, the position will be randomized again.

Objects Collision

The objects (walls, collectible objects, and the goal flag) use axis-aligned bounding boxes for collision detection every frame. Each object has a 0.1x0.1x0.1 bounding box. The game will loop through a list (vector) of walls and a list of interactable objects (walls and collectible items) then check for collision one-by-one. If there’s a collision between a wall and the player (using player’s position point vs bounding box), the game will prevent player movement. If there’s a collision between a collectible item and the player, the number of collected items increases. For a goal flag, if there’s a collision, the game will check for collected items and show the level complete menu.

Changeable Camera

The player camera can be changed into a top-view camera. The changing animation utilized linear interpolation between player camera position, rotation, and hardcoded top-down view position, rotation. The linear interpolation between Vector3 is done by using Vector3::Lerp function in SimpleMath namespace.

Skybox

The cube model and the skybox texture are loaded from files. The cube model is also scaled to 10 times to original size and rendered with its texture.

Blur post-processing effect

The blur post-processing effect is done by using BasicPostProcess class that is included in DirectXTK. BasicPostProcess provides multiple post-processing effects like Monochrome, Sepia, Blur. For this project, the blur effect is applied to a render texture before showing it to the game screen.

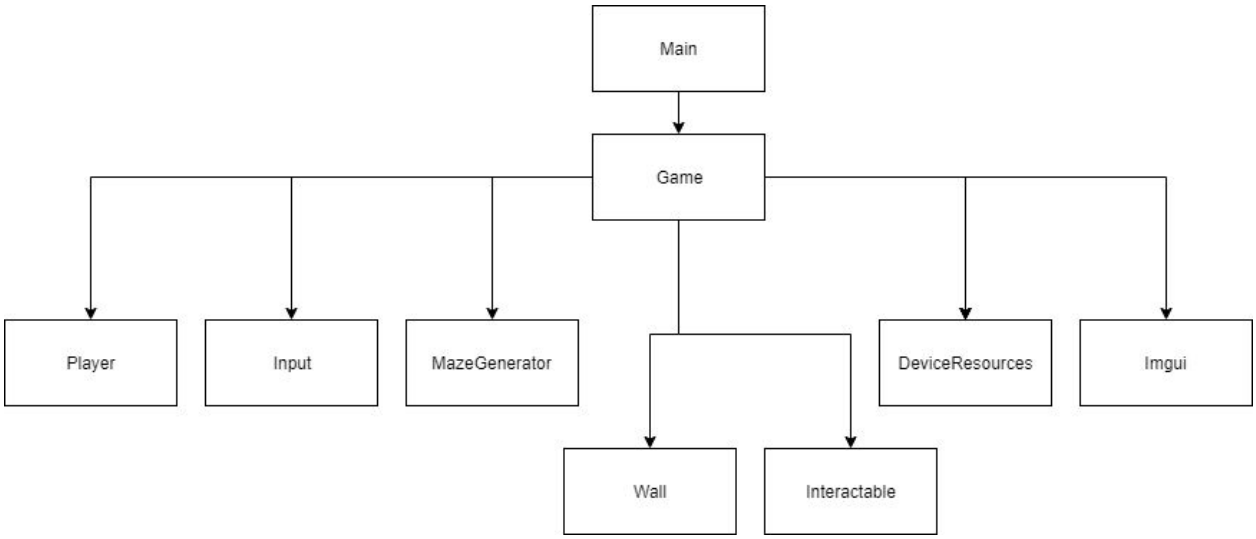
Game UI

The game user interface (main menu and level complete screen) utilized Dear Imgui library.

Code Organization

Classes

The classes structure in this project is described as in the diagram as follows:



Classes Description

Main class - Initialize the game window
Game - Manage game logic
DeviceResources - Manage DirectX rendering and resource
Imgui - Used for game user interface implementation
Player - Used for storing player's position and rotation
Input - Manage the player's input (mouse and keyboard)
MazeGenerator - Generate a maze within a built-in grid.
Wall - A class represents a cube wall in the game that has position and size data for rendering and collision detection.
Interactable - A class represents a collectible item or a goal flag in the game that has position and size data for rendering and collision detection.

Data Structure

- A grid in MazeGenerator used vector<vector<int>> for storing integer values in 2-dimensional that acts like booleans.
- Both the list of walls and interactable objects used vector<Wall*> and vector<Interactable*>.

Evaluation

On the maze generation, the depth-first search is a simple algorithm compared to the others. It's fast and easy to implement and it can produce a maze efficiently. After the maze layout is generated, spawning objects on top of it can make sure that the level is solvable.

References

3D collision detection. (2019) Available at: https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection (Accessed: May 12, 2020).

Chuck Walbourn (2018a) *BasicPostProcess.* Available at: https://github.com/microsoft/DirectXTK/wiki/BasicPostProcess (Accessed: May 12, 2020).

Chuck Walbourn (2018b) *PostProcess.* Available at: https://github.com/microsoft/DirectXTK/wiki/PostProcess (Accessed: May 12, 2020).

Mike Gold (2007) *Generating Maze using C# and .NET.* Available at: https://www.c-sharpcorner.com/article/generating-maze-using-c-sharp-and-net/ (Accessed: May 12, 2020).