Bytom, 05.11.2020

Computer Programming Laboratory
III semester

Final Programming Project Report

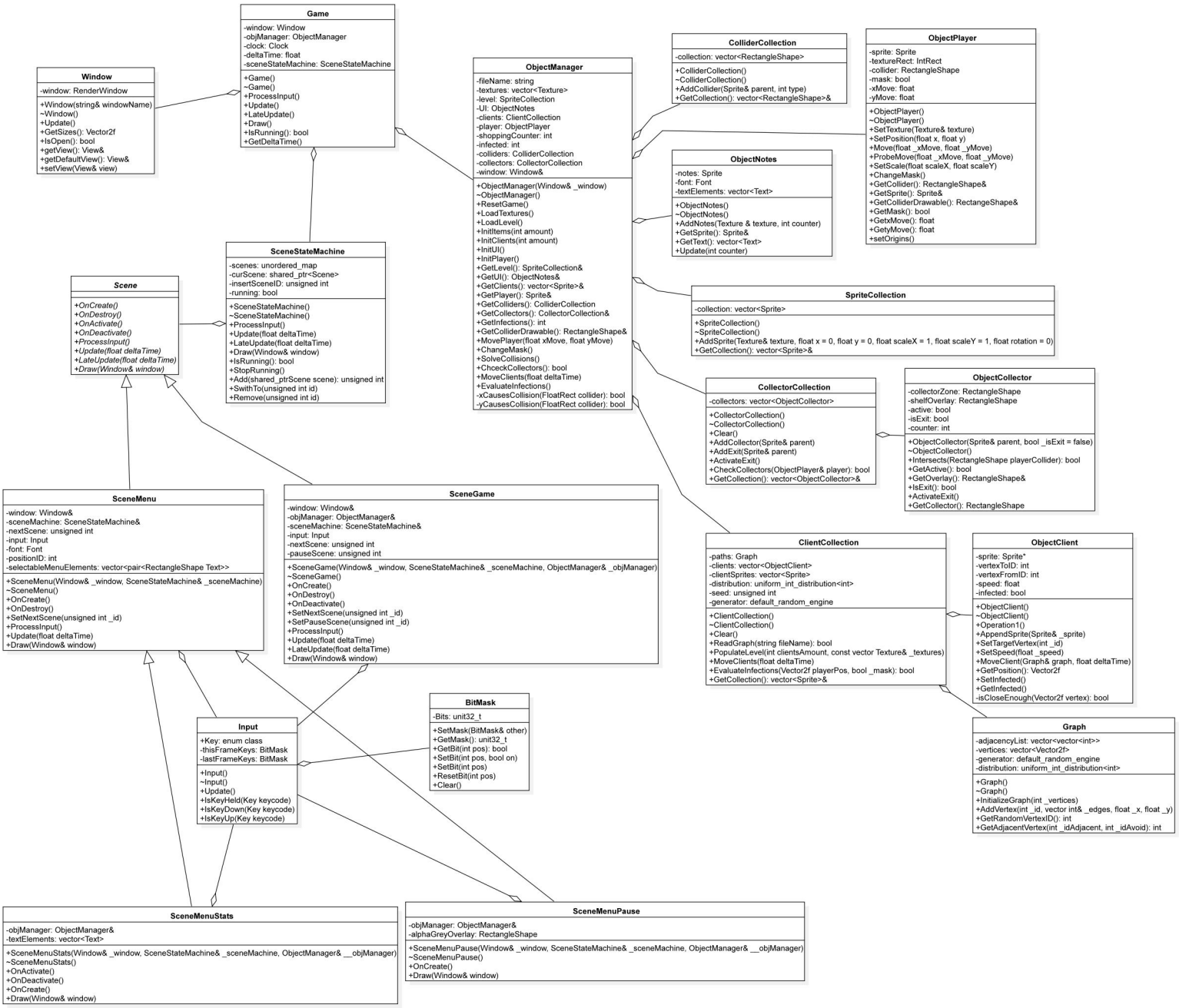
Author: Radosław Rzeczkowski
Email: radorze093@student.polsl.pl
Tutor: Krzysztof Pasterak

1. Topic and description

The aim of this project was to create a simple game/interactive experience raising awareness around COVID-19 safety measures and showing the importance of physical distancing. The code had to use the OOP paradigm. The game, called ‘Beware’, accomplishes that goal by tricking the player. At first it sets a simple goal for the player: buy some items in a shopping mall. The player moves around the level containing shelves of goods and other clients. After finding and collecting all the items the player goes to the cash registers and finishes the game. At this point the true intent of the game is revealed: the final screen tells the player that before they went into the shop they had been infected with COVID-19. It also contains a counter of how many people the player has infected during shopping. The game constantly monitors the distance from other clients in the shop and based on that it can infect the clients. The real goal of the game is to infect as few people as possible, keeping a safe distance. The player is also informed that they can put on a mask to lower the chances of spreading the virus. The player can then choose to play again or exit the game.

This topic poses many problems to be solved since it’s quite complex. The most general structure of the game can be described by a Final State Machine and an Objects Manager. The final State Machine contains Scenes and controls them. It can also switch between the current Scene to a next Scene. The Scenes have shared access to the Objects Manager and use it’s methods to accomplish some specific functionality.
The problem of controlling the clients by the game was solved by loading a pre-made graph structure from a text file that represents the paths that clients can take. A vertex is described by an ID and some coordinates in the game and an edge between two vertices means that the clients can move between them in a straight line, with no obstacles in the way. The clients are spawned at random vertices at the start of the game and pick a random adjacent vertex to move to. Once they get there they repeat the process of picking a new adjacent vertex with the exclusion of the vertex they just came from. This results in nicer, more coherent movement. The graph is represented by an adjacency matrix. Thanks to this approach there was no need to implement a path finding algorithm. This way the clients also don’t have any collisions which simplifies the implementation and improves performance.
The collisions for the player were implemented by storing the last change in position and using it to correct the players position in case of a collision with a Collider object.
The items that the player has to pick up are visually represented by flashing shelves and are spread randomly across the whole level. The player can collect an item by pressing E while being close to the shelve, that is when the player’s collision box is in contact with the shelve’s collection box.
The level is built out of tiles. The overall layout of the level is also loaded from a text file which contains sequences of pairs of numbers representing the type of texture to be used for the tile and it’s desired rotation. This approach was very useful both in the early and late stages of development where changes to the map could be easily introduced without altering the code.

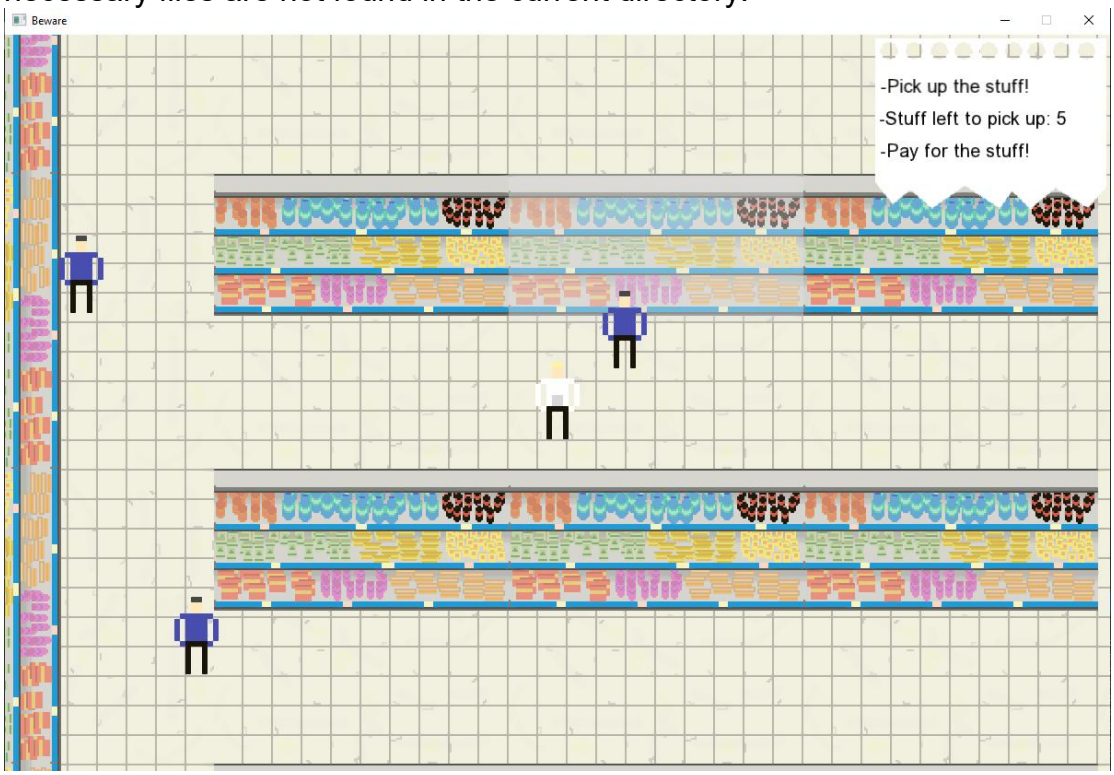
2. Class diagram



3. External Specification

The game in controlled using the keyboard. The key bindings are as follow:
- W : up
- S : down
- D : right
- A : left
- E : pick up
- M : put on/ take off the mask
- Escape : pause menu
- Enter : choose option

In menus W and S is used to navigate.
The game executable needs to be placed in a directory along with appropriately named textures and two text files following a specific format: one containing the level and one containing the AI paths. The game will not run if necessary files are not found in the current directory.



The game screen. Player in the middle, clients moving around in blue.
The shelve above the player is blinking and contains an item to be collected.
In the top right corner: information about objectives.

4. Description of chosen classes

-Game: This class is the main class of the program. This class is not used by any other class and all other classes are used directly or indirectly by this class. It is the only class directly in the main function. It's methods are general and are used in a loop to generate a frame. It's methods mainly call functions of the SceneStateMachine. In it's constructor the Scenes are created (eg. SceneGame, SceneMenu), configured and added to the SceneStateMachine. This class also keeps track of deltaTime, the time between generated frames,

that is used widely in other classes for achieving smooth motion regardless of the frame rate fluctuations.

-SceneStateMachine: This class stores and manages Scenes. It is also used to change the current scene. Using smart pointers this class calls generalized methods of classes that inherit from the virtual class Scene. This allows for easy switching between game scenes (eg. Menu, game, pause) and also for generalization of control and processes that are quite different in each scene.

-ObjectManager: This class stores and manages everything needed for the gameplay. It holds the textures, the sprites and all other elements like a graph or colliders. Most game scenes have reference access to it which ensures continuity of the state of the game between for example the game scene and pause scene, since they both have access to the same data. It's methods provide a more specified interface to be used by scenes. They allow loading and initialization of the game, retrieving the contents of this class for drawing to the screen, manipulating it's contents more directly (like moving the player by given amount of pixels) or indirectly (like moving the clients, which is controlled by the client itself). As it's attributes it mostly holds objects of Collection classes eg. ClientCollection or ColliderCollection.

-ObjectPlayer: This class is an example of this program's low level class since it doesn't use any classes created in this project, only std and sf classes. It represents a player object that has a sprite, a collision box and some other properties like a boolean for whether the player character has their mask on or not. Its methods are meant to crate an interface for the sprite and collider box, like a move function called on a player object moving both the sprite and collider in according ways.

5. Testing

The game's testing comprised mostly of playing it in non typical ways. Things like trying to glitch the collisions in the corners of the map or pressing E quickly to try to collect multiple items at once. The game was also played by a few other people that didn't find anything unexpected. The balance of the game was also tested so that it wasn't too easy or too hard. General performance was also tested with extreme amounts of clients on the map and showed that the game still ran smoothly with much more clients then it makes sense to play with.

Most test yielded expected results. Found bugs were fixed.