Pacman Project

by Nils Van de Velde 20240345

Our main function creates a Game object and starts the game loop by running the render function. In our Game class you can find 4 static variables which are used throughout the project. Some of these are made inline to prevent undefined reference errors. The window which is used by almost every State is static for easy access, so that not every State contains a copy or pointer to the window. Fonts and the spritesheet are made static so that we only have to initialize them once and from then on we can use them wherever we want. As previously stated the render function contains the game loop which does the following:

1) Check if the window is still open.
2) Update the Stopwatch and check for SFML Events.
3) Clear the screen and render the current State.

The StateManager uses a stack of States to manage them. Through the StateManager we can directly call the update and render functions of the States. Every State has 1-3 update functions and the StateManager decides which update to call. This way a State doesn't have one giant update function and we can clearly separate key, mouse and text input without the manager needing a bunch of different functions. Every State handles its own rendering and is able to do this via the render function and the public static window member inside Game.h. I could've made it so that Game and Camera are the only owners of the window, but then Camera had to handle the rendering of States, which is a bit complicated for MenuState thanks to the scoreboard. I wanted the Camera to handle the rendering of the actual game elements namely: entities, current score and current amount of lives. Since the window is a public static variable inside Game.h, Camera doesn't have its own member variables, so making it a Singleton is in this case not optimal. Because we don't need a Camera object/instance, Camera only contains static functions, which are called where needed.

When we are in the MenuState there are 2 ways to start the game. First you need to type in a name for the scoreboard, but after that you can either press the play button or press enter, which I feel is more user-friendly than only having the play button. Having names by the top 5 scores is also a pretty fun addition, since you can see who gained the highest score. I added a Victory and Defeat State to have a pause in between two levels and to prevent the render function of LevelState from becoming too big. It is possible to put the victory and defeat screen inside LevelState, but this would overcomplicate it. LevelState's render function is the only render function of a State that does something else than just rendering. It also calls the World tick function. The world is created inside LevelState's constructor and a new World is created when you complete a level, while the old one is removed. The world uses the EntityFactory to create entities and we give World the current level so that it can set the Ghosts difficulty. The EntityFactory creates both a model and view of all entities, except Wall, Coin and Fruit. These 3 have no animations and only use a static image, so they are handled via a base EntityView object. The factory also contains a stack of all "ghost types", so that the ghosts always have the same logic, color and wait time combination. The last thing the factory does is give the Views the correct sprite(s).

I implemented my own Event system for easier communication between Observer and Subject. It is a pretty robust system that isn't optimally used but can be further expanded on and used differently with different representation code. Subjects or EntityModels use the notify(event) function in two ways. It handles events that have an effect on logic, such as DirectionChangeEvent for Pacman, and it can notify an Observer of the event. The Events contains almost all the information needed to handle it, such as direction and facing in TickEvent, so that the Observer and Subject can properly handle it.

The World contains 3 separate vectors for the entities, to better separate logic, and a PacmanCollsionHandler. The walls vector is used for the isWalkable function and for rendering, which is done via tick and the notification of a RenderStaticEvent. The ghosts vector is used for collision with ghosts, to give them the different functions they need and to reset their position when Pacman loses a life. Lastly the collectables vector is used to check for the amount of collectables, collisions and to check for expiration. I decided to mark collectables as expired when eaten so that they are removed from memory, which makes the game more memory efficient. The PacmanCollisionHandler contains Pacman, so that its tick function can be called. I have a different CollisionHandler for Pacman to prevent the Pacman model from being overcomplicated and having too many tasks.

The ghosts are given functions which give them access to Pacman's position, the location of walls and a gridmap. Since I use BFS for Ghost AI, I gave them a gridMap to be able to implement it better. Either way a game like Pacman works better with grid coordinates. I decided to use BFS, because I was irritated by the fact that Ghosts often walked against walls to try and get through it and so that I could easily make the return to their spawn. The movement code in general is split into multiple functions for better organization and to make things like target manipulation easier.

In the file Util.h I have some utility enums and structs like Coords for coordinates. Coords also contain height and width for a hitbox, even though in my project every entity's hitbox is the same. These are also used to calculate a cell size for the projection with the Camera.

I also made a separate AnimatedView class to better generalize the animation code for animated entities. Lastly I implemented a map system using text files for easy reading and writing, which is also why I use a json file for score, and the game works best in a 16:9 aspect ratio, rescaling messes with game logic and will create a difference in horizontal and vertical speed. I especially recommend running it at 1080p or 1440p, which is the resolution of the monitor I mainly tested on. There are improvements that could still be made, but I personally think I've crossed many boundaries for myself, I improved a lot making this project and it turned out pretty well.

More/other information can be found in the project README and for more information on the map system you can find a separate README inside the maps folder.