Thibaut Van Goethem

S0171259

Report RoadfighterGame

When I started this project I wanted to make sure I wouldn’t make a mistake I used to make in a lot of my projects, which was to not think about a good design and to make sure everything is separated and logical. This made it extremely difficult to add new features without somehow “hacking” them in. Luckily the specification that were given for this project required the use of certain design principals which came in handy, but this was nearly not enough for the whole project thus some extra thought was required to ensure the design I wanted. This design is what I'm mostly going to talk about in this report.

First of I made a split between the game logic and the graphical implementation of the game as was required by the specification. Thus I'm also going to talk about them separately, but because you kinda have to keep the graphical implementation in mind when creating the logical one I will sometimes refer to the graphical implementation when talking about the logical one.

Game logic library(GLL):

The first thing I did when creating the GLL was to create a good inheritance structure for all the entities so that the composite design pattern is followed. In this structure the entity interface class sits at top with only 3 virtual functions, 2 of which are for the tick system (more on this later) and 1 is the draw function which you should only override in the graphics implementation. From this entity class 2 classes inherit, the World class and the Collision Object class. The collision object class is the class that actually holds the 2 position of an entity, and makes sure they don’t go out of bounds. This class furthermore makes it possible to provide simple rectangular collision detection, It also has 4 virtual functions (crash, win, bonus, shot) which can be overridden to make certain effects happen on different entities. The world Didn’t need all of this functionality thus it only inherits from the entity class. From the collision Object class another 2 class were made. The first being the End class which denotes the finish line, This class had no need for movement thus it being a collision object was sufficient. For the classes that did need movement I created the Moving Object the class which inherits from the collision object class. The moving object class handles all the logic behind the accelerating and changing the positions accordingly each moving Tick. Lastly there are 5 classes that inherit from the moving object which are all the cars and the bullet class.

Now I will talk a bit about each “leaf” classes in the GLL.

The End class is as earlier said a stationary object in the game world, each object that collides with it gets the win condition called on it.

The Bonus Car and the Passing Car class have mostly the same functionality as in they have a set speed which should not be changed throughout it’s lifespan, once they go bellow y value 4 in the game board or they hit something they are deleted at the next logic tick. The only difference is that the bonus car calls the bonus function on all cars colliding with it ,while the passing car calls the crash function.

The Racing Car class is used as the car you race against in the game, these cars use the random number generator singleton class to choose a movement between left and right each logic tick. It also constantly accelerates. When something crashes against this car the crash function is called upon them. If this car gets shot its speed will be set to 0. If it crashes it stops for a certain time and then gets an immunity for a certain time. If it gets a bonus then its current speed gets a multiplier of 1.2 which can go beyond it max speed. Lastly if it wins then it slows down and cant move anymore.

The player Car has a lot of the same functionality as the racing car but changed slightly hence why no inheritance from the racing car was used here. The big differences here are the fact that It can shoot once in a while and that the movement is controlled by a move Controller which will be talked about later. Because these bullets are created by the car itself, the car needed a shared pointer to the entity factory and the entity transporter.

The world is a class that is supposed to hold a bunch entities and call certain function on it each tick. The collision detection and deletion of the entities is also done here each logic Tick.

The way entities are send to the world is trough an entity transporter. The logic behind this class is inspired by the observer pattern. All the classes that need to transport entities to the world have a shared pointer to the transporter as data member and gives entities to it. Then at each logic tick the world looks in this transporter and adds it’s entities to it’s own class and flushes it.

The way the Input works is also somewhat the same way, the Player Car gets an Input controller given with him in which he looks each movement tick for new input. Each object that can give input can get the controller out of the factory and give it movements.

Factory driven design was also followed in this project trough the entity factory. I first made a class that is the base for all entity Factories then to you are supposed to implement it which I did in the GLL with the class GLL\_Entity\_Factory. The entity factory also creates the entity transporter, The ScoreObserver and the movement controller and gives them to the right entities.

Observer Design was also followed by first making a observerbase class with a single virtual void function that quickly gets implemented in the ScoreObserver Class that inherits from the observerBase, here the update function just adds to the score. Then to complete the observer design there also is an observable class that is used to attach/detach observers and also notify them when needed. A lot of entities inherit from this observer so they can do things with the score. e.g. when a bonus car gets shot 100 get taken from the observers trough calling notify(-100) which was inherited from the observable class.

the highscoreManager class is made to handle the reading from/writing to and adding of highscores. The reading and writing is done trough json. Here I used the popular nlohmann json parser. When you add a highscore first all scores are read and put in a vector. This vector then gets sorted with help from a lambda function and if the size is larger than 10 the last score gets cut off. Then I just write them back to the same json file.

The two last big things in the GLL is the roadfighterGame class and the tick system. The roadfighterGame class can be seen as the main class. It uses the factory to create the initial state of the game and add new random cars each logic tick. It also sends the score to the highscoreManager after the game is done and can pause the game if it wants to. It also handles the tick system.

For the Tick system I wanted to make it possible to have a high frame rate without checking all the logic each time you tick. Thus I made 2 tick functions in the entity interface a logic tick and a movement tick. Here the logic tick does what the name says, it checks all collision,all states,possible adds new entities,… .The movement tick only updates the movement of all entities. Thus If 0.01 seconds have passed a movement tick will move the car with 0.01 seconds of it’s current speed without actually accelerating and such.

From the roadfighterGame class this tick system is not visible, here you must only tick with the amount of time passed since last tick. The game class itself will decide when a logicTick must occur (currently set at 30 times a second).

Most of the functionality of the GLL library has been tested using the gtest library, these test can be found in the Test directory. The graphics implementation was not tested because it has no special logic that needed it.

Graphics Implementation:

the graphics implementation was pretty simple after the entire logic library was done.

First of the singleton transformation class was created, this class is used to transform the game coordinates to coordinates scaled to the sfml window.

For the entities I first created a class called SFMLEntitySprite which handles the loading and storing of a sprite for an entity, hence why it inherits from the sf::sprite class.

Then to actually make the entities I used multiple inheritance once more to inherit from the EntitieSprite class and it’s respective GLL Entity class. I did this for all the “leaf” classes in the GLL. All these graphical entity classes override the draw functionality to draw themselves on the renderwindow that is given to them from the new SFML\_Entity\_Factory.

This new factory is created in initialization of the sfmlRoadfighter class and is given a renderwindow to give to all it’s created entities. It also doesn’t make GLL entities but the new graphical ones which it then passes trough as normal entities. This factory is passed as a base factory to the GLL where it is then used to create all entities.

Lastly we have the SFMLRoadfighter Class which is made as an object in the main function. During it initialization it creates the sfml\_entity\_factory and the renderwindow. It also takes the input controller from the factory as it is needed latter. The initialization does not start the main game loop, this is done when you call the function “rungame”. In this main loop I have locked the max speed at 120 times per second but it can be made higher if wanted. Also if you computer can’t handle 120 fps it will automatically go lower without affecting the game(unless it goes bellow 30 fps). The game loop runs on a clock that can be found in the utility directory of the GLL. It is a simple wrapper for the c++ clock functionality to make it easier for me.

Anyway each tick it first sees if there is new input, or some input stopped and give this to the right function according the state of the game. Then the window is cleared, the game is ticked with the amount of time that has passed since last clock. Next I call the draw function on the game which calls the draw function on the world and thus all it entities, This draw function also draws certain elements itself like the side statistics. To end the loop I reset the clock and display the newly drawn renderwindow.

Throughout the SFML implementation and the GLL I also used exceptions for when things could go wrong. For the GLL this is called the GLLException, for the SFMLImplementation it is called the pretty verbose SFMLConversionException. Some of the exception were handled but those that should stop the program are caught and handled in the main function.