**Roadfighter**

Lander De Roeck

University of Antwerp

Belgium

*Lander.DeRoeck@student.uantwerpen.be*

**Road Fighter Entities:**

Keeping track of the size and location of entities can be done in a few different ways, but since we only need to work with rectangles, we only need two corners to be able to reconstruct the entire rectangle, which would translate in 4 floats, 2 for each corner (x and y), rather than keeping the corners, I have chosen to keep the centre of the entity, and half the width and length, this way I can easily calculate all 4 corners. The reason for this is so I can easily move the entity around, I only need to change 2 values, the two that indicate the centre of the entity.

The entity collision itself is handled by calculating if a corner of an entity is inside of another entity, if this is the case the entities will act accordingly.

**Abstract factory:**

The abstract factory has abstract virtual functions for each entity type that needs to be able to be made, the concrete factory overrides these functions and returns smart pointers to the SFML variants of the base entities. At first, I had an abstract factory for the base entity class, and derived all factories from this abstract factory, a different one for each derived factory, but this caused needless clutter and inconvenience.

**Transformation singleton:**

The Transformation singleton is hardcoded to work with the [-4:4] [-3:3] window. It will make sure that the background can stack 3x times on the screen. The transformation singleton is one of the only pieces of code that isn’t dynamic, which means that it will need to be changed to be able to support different size backdrops, all other entities will automatically change their size (and hitbox) if the size of the texture changes.

A small problem I noticed with this is that due to small rounding errors (because of converting floats to int values), the background doesn’t always render smoothly, small 1px gaps can pop up between the background sprites. (This is more noticeable with odd screen sizes.)

**Random singleton:**

Used srand(seed) and rand(), while these won’t give true randomness, it will give me the same sequence every time, so the “random” events will be the same each time, while this might not be ideal for a game, it is easier to check. If we want to make the game more random we can remove the seed or use the c++11 random library. Clion will give a warning about this, but this is intended.

**SFML Entities:**

All the SFML entities keep their own sprite and texture data, while it probably is better to store the texture data in the factories, so it can be reused, next time I’d prob change this.

The SFML entities all have a constructor, this constructor handles the loading of texture data, loading from sprite, and the position and size.

Because of a weird issue where the textures disappeared randomly, I load the texture in the sprite each time it gets drawn, this seemed to fix the issue.

SFML throws a warning in console: “Setting vertical sync not supported”, I can’t seem to get rid of it, I suspect it’s a limitation on Linux, it seems other students have this warning as well. Even disabling vertical sync with sf::RenderWindow::SetVerticalSyncEnabled(false) doesn’t get rid of the warning. This warning is still present.

The world class keeps all the entities, UI information (like text, score, …), the distance the player has traversed and the eventual finish distance. I also decided to integrate the keyboard input in the worldSFML class, this looked like the easiest way. The world keeps track of speed because this is a side scroller, this means that the car doesn’t physically move along the y axis, rather the background and other entities will move slower or faster depending on the speed of the car. Because the other entities are dependant on the speed, the draw function for those entities allows for a speed integer to be passed.

While passing cars have a predefined speed (200km/h), race cars do not, so they have their own speed integer, with this it will calculate the relative speed at which the race car should move along the screen. Furthermore, race cars also need to be able to steer, unlike passing cars (although they will still move in a collision), there is a very basic AI, which will randomly pick if it will drive straight, turn left or turn right every game tick, while this is not ideal, it works, though the car looks jittery sometimes, since it’ll move back and forth.

**Shared Pointers:**

I have tried to have a consistent use of smart pointers throughout the project, where possible I tried to use std::make\_shared over std::shared\_ptr, since the first is more efficient.

**Travis:**

While this isn’t really part of the project, getting Travis to work properly was a challenge in its own, a lot of my commits are solely to get Travis to work, the first challenge was getting the right SFML version and getting it to be recognised. Eventually I wound up downloading the binaries from the SFML site directly and moving them to /usr/.

**Part 2:**

**File reader:**

Because the project required much more interaction with external files (configuration, level files, highscore) I implemented a file reader class. To keep the code as flexible I decided to parse the input into a map, and have different functions for requesting different data types. So far there only is a return function for int, bool and string type, but this is easily expanded upon to allow for more data types. The class will throw custom exceptions if you request an invalid key or the incorrect type for the value which corresponds to the key.

**Configuration (file):**

The configuration class is a singleton, this way I can easily read the values throughout the entire project. It uses the file reader to read the configuration file, most of the values get their own function, this is to make it easier since you don’t need to remember keys. While I might have missed some “magic values” to add to the configuration file, due to the way it’s coded it’s really easy to add it.

**Different levels (with configuration files):**

Now implemented is a menu screen, where you can select one of 3 different levels. Each level has their own config file, in the config file you can change the distance the player needs to travel, whether or not the player needs to face obstacles, if there are computer-controlled entities and enable the boss or not. After the level has ended you will receive a victory screen with your score, after a 10s delay the next level will begin.

**Exception class:**

While exception classes have only been used limited in the project, I have used them where it would make the most sense. The file reader is the place where most unexpected errors might occur, also the configuration class has an exception, used to specify if the config data is missing.

**Notable changes:**

Game class has been split in Game and GameSFML, to keep separation between graphical library and core mechanics. The game class is responsible for the game state, it handles the menus, levels and victory screen. The world class now represents a level.

Lastly some fixes that happened since last time.

* “Screen tearing” where black borders would appear around the road occasionally have been fixed. [1]
* Collision detection has been improved
* High scores now get saved to a file
* Bullets now have a cooldown timer, rather than being able to be spammed

[1] The tearing was caused by converting to the incorrect type. All of the virtual coordinates have now changed to doubles for increased precision and the transformation class has been changed accordingly. It also returns the actual location on the screen in float value rather than int, which was what caused the tearing issue, because SFML used floats. So, there were rounding errors that occurred.