

## **Computer Games Architecture & Optimisation**

Luna Null (25813979)

University of Northampton

CSY1097 Computer Games Architecture & Optimisation

Dr. Anastasios G. Bakaoukas

February 6th, 2026 - May 15th, 2026, 15:00

1. Introduction.....	4
2. Gantt Chart.....	5
3. Design .....	6
3.1. Game Boot.....	6
3.2. Main Menu .....	6
3.3. Tutorial Mode.....	6
3.4. Level 1 .....	6
3.5. End Screen.....	7
4. Implementation .....	8
4.1. Basic overview .....	8
4.2. Collision Detection.....	10
4.3. Player Movement .....	10
5. Optimisation.....	11
5.1. Obstacles Collision Detection Early Exit.....	11
5.1.1 Original Code.....	11
5.1.2 Optimised Code .....	11
5.1.3 Explanation .....	11
5.2.1 Original Code.....	12
5.2.2 Optimised Code .....	12

5.2.3 Explanation .....	12
6. Testing.....	13
7. Conclusions.....	14
8. Further Work.....	15
9. References.....	16

## **1. Introduction**

The goal of this assignment is to show evidence of planning and development of a game featuring two levels, using C++/SFML, with emphasis on properly applying code optimisation techniques and proper game architectural structure.

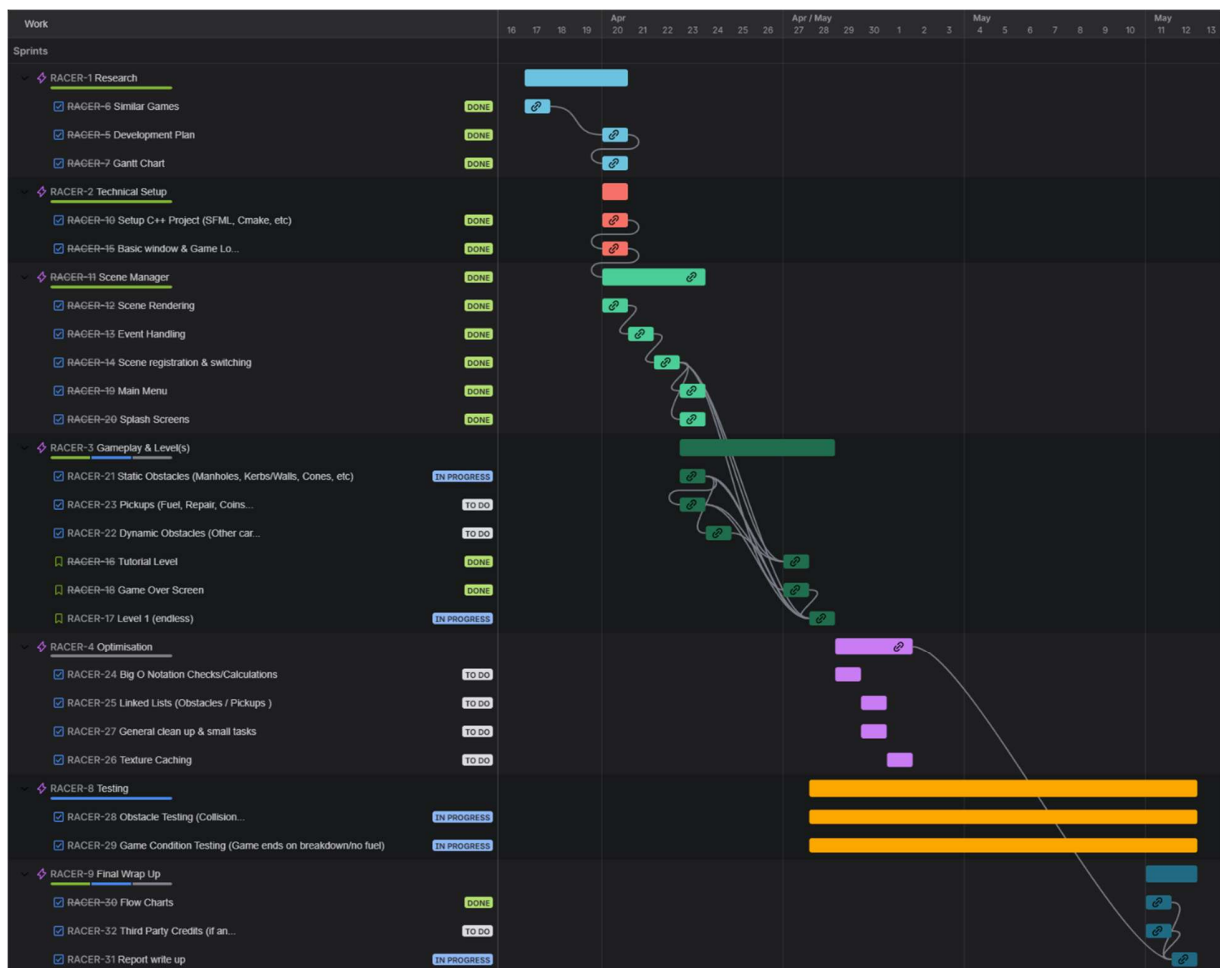
The game I will be recreating is “Dangerous Racers” which is an endless racer where the player drives on a road/track, collecting coins and fuel, while avoiding obstacles such as other cars, manholes, and kerbs/road-dividers.

In this report, you will find sections detailing the implementation of my code and the reasonings behind it. As well as sections related to optimisation techniques I used while developing and/or used when reworking my code to improve it.

There will also be a “Further Work” section, which details with hindsight what I might have done differently and/or other changes/additions I would make if I had more time to work on my project.

Finally, the report will end with references to sources I borrowed from to complete my project, such as code borrowed from the Lecturer’s classes and/or examples borrowed directly from the SFML Games Programming library.

## 2. Gantt Chart



For this project, to keep my work more organised I tried using Jira for task tracking. This allowed me to set my start and due dates for tasks, creating the above timeline. While not explicitly a spreadsheet Gantt Chart, I believe it gives enough information and overview of the project schedule to be treated as one.

### **3. Design**

Design elements of game – logic behind placement of obstacles, assets, etc.

#### **3.1. Game Boot**

The Game boots with a splash screen featuring a studio logo that fades in and out, this screen is skippable by pressing any key, and this transitions to a Main Menu.

#### **3.2. Main Menu**

The Main Menu features three options, Tutorial Mode, Play, and Quit. Tutorial Mode transitions the player to the tutorial so that they can learn how to play the game, play starts level 1 and quit safely exits the game.

For the brief, Tutorial Mode and Level 1 are considered 2 separate levels. With Tutorial Mode being a scripted, meaningless choice level with a forced end and Level 1 featuring an endless runner.

#### **3.3. Tutorial Mode**

Tutorial Mode has a forced path with meaningless choices to navigate the player from start to finish so that they understand avoiding obstacles, their fuel and health, and pickups that increase the two stats, before forcing them to ultimately end by running out of either of the two.

This helps them understand how the game works, before they progress into Level 1.

#### **3.4. Level 1**

Level 1 is an endless level, which lasts for as long as the player is skilled. Obstacles spawn at random that the player must avoid, while also collecting randomly spawning fuel/repair-kits to increase their stats.

Progress is rewarded as a Distance score, the higher the score the faster the level becomes until eventually the player runs out of health or fuel.

### **3.5. End Screen**

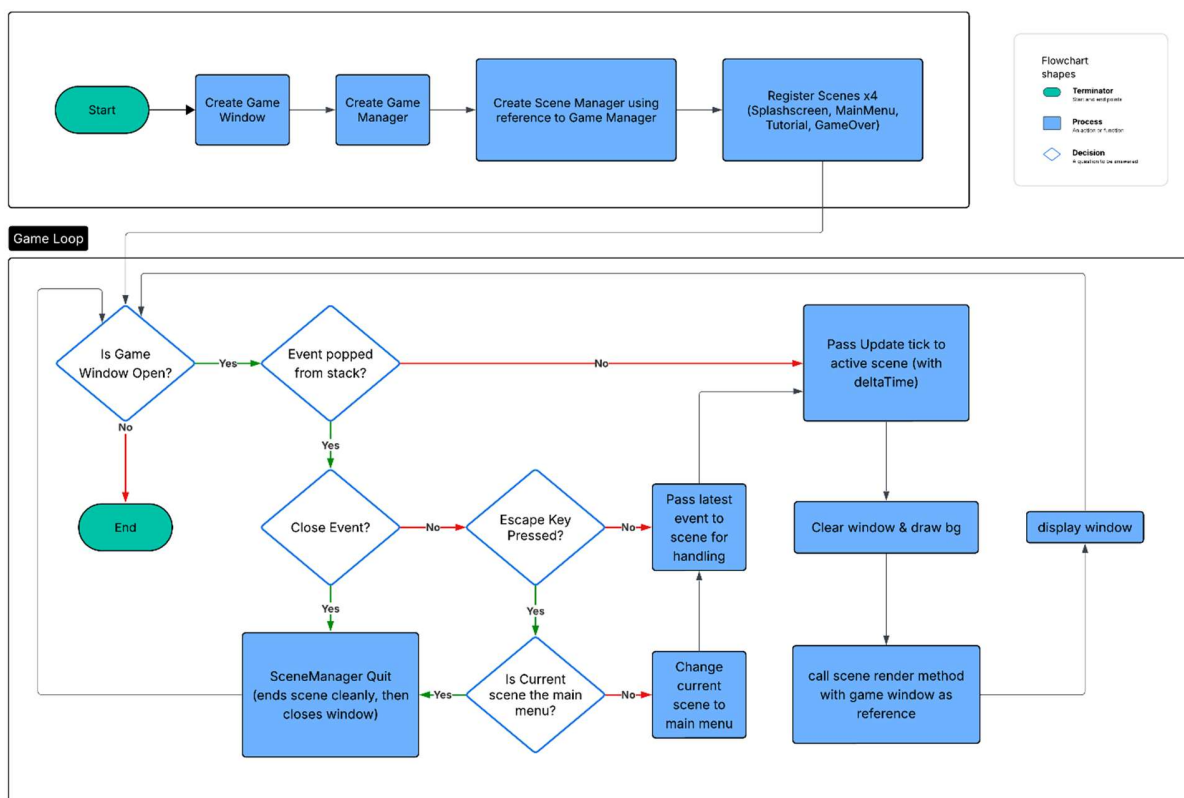
Shared between the Tutorial level and Level 1, this end screen shows the player's final distance score and their highest achieved score. It also gives the player buttons for retrying or returning to the main menu.

## 4. Implementation

### 4.1. Basic overview

My Implementation of this game uses what I call a “SceneManager”, this was inspired by scenes from other game engines such as Unity and Godot, the concept being that I could separate visually separate “scenes” of my game, i.e splash screen, main menu, levels, etc, into classes that derive from a base Scene class, so each of my scenes had methods such as start, leave, update, render and eventHandler. This allowed me to cleanly separate code per scene and only run the necessary code.

The overall concept works like this:



The game starts and does its initial setup such as; Creating the game window, the game manager which is a class that keeps record of data that I want to persist between scenes, i.e. the player’s score, creating the scene manager which handles switching the scenes, and is the



middleware between the game loop and the currently active scene, and then finally we register the scenes into the sceneManager using the addScene method, which takes a string which we use as a key for when we want to switch scenes, and a reference to a method for creating the scene, which we refer to as the sceneFactory.

After this initial setup is complete, the game loop begins which consists of the usual; Check if the window is still open, if not we exit the code with a return 0 for successful exit. Check if there is an event in the SFML event queue, and do some top level checks such as if it was a close event, in which case we run the sceneManager `_quit` method which does a safe cleanup of the currently active scene before finally closing the active game window. Check if the Escape key has been pressed, if so we check if the current scene is the main menu – if it isn't, we switch the scene to the main menu, otherwise if we are on the main menu we trigger the same `_quit` from the sceneManager.

## 4.2. Collision Detection

Collision detection for obstacles is checked by passing the global bounds of the player sprite to the `ObstacleManager`.

```

1. // tutorial_level.cpp
2. if (!reverseTime && !player.isInvincible() &&
obstacleManager.obstacleCollides(player.getSprite().getGlobalBounds())) {
3.     // ... Logic handling when a collision occurs
12. }
13.
14. // ObstacleManager.cpp
15. bool ObstacleManager::obstacleCollides(const sf::FloatRect &collisionObject) const {
16.     for (const auto& obstacle : obstacles) {
17.         if (obstacle->getYBottom() < collisionObject.top) return false; // If obstacle is above
collision object, skip all.
18.         if (obstacle->getSprite().getGlobalBounds().intersects(collisionObject)) return true;
19.     }
20.     return false;
21. }
22.

```

This function returns a Boolean depending on if an obstacle is close enough to collide, and if it does. There was optimisation done to this function, see 5.1.

## 4.3. Player Movement

The player does not move up/down in my game. Instead, only left to right. I chose this method as I felt it would simplify the necessary event handling logic as I only need to check the mouse's x position when a move event is detected, and I only need to cap these values between values I chose which keeps the player on screen.

```

1. // tutorial_level.cpp
2. void TutorialLevelScene::eventHandler (sf::Event& event) {
3.     Scene::eventHandler (event);
4.     /* ... Logic related to exiting the scene, as well as time reverse ... */
5.     // Checking if the mouse moved, and moving the Player on the X axis.
6.     if (event.type == sf::Event::MouseMove) {
7.         player.moveTo(std::clamp(event.mouseMove.x, 32, 373), player.getPosition().y, 0.025f);
8.     }
9. }

```

## 5. Optimisation

### 5.1. Obstacles Collision Detection Early Exit

#### 5.1.1 Original Code

```

1. // ObstacleManager.h
2. std::vector<std::unique_ptr<Obstacle>> obstacles;
3.
4. // ObstacleManager.cpp
5. bool ObstacleManager::obstacleCollides(const sf::FloatRect &collisionObject) const {
6.     for (const auto& obstacle : obstacles) {
7.         if (obstacle->getSprite().getGlobalBounds().intersects(collisionObject)) return true;
8.     }
9.     return false;
10. }

```

#### 5.1.2 Optimised Code

```

1. // Obstacle.h
2. float getYBottom() {
3.     return this->getPosition().y + this->getSprite().getGlobalBounds().height
4. }
5.
6. // ObstacleManager.cpp
7. bool ObstacleManager::obstacleCollides(const sf::FloatRect &collisionObject) const {
8.     for (const auto& obstacle : obstacles) {
9.         if (obstacle->getYBottom() < collisionObject.top) return false; // If obstacle is above
collision object, skip all.
10.        if (obstacle->getSprite().getGlobalBounds().intersects(collisionObject)) return true;
11.    }
12.    return false;
13. }

```

#### 5.1.3 Explanation

I implemented my collision detection using a for each loop that iterates over my Vector of obstacles, returns true if a collision is detected otherwise false. This is fine if there is a small number of obstacles on screen, but if the number of obstacles was to increase and a collision wasn't occurring, the code would have to loop through all obstacles before it could exit.

Because the obstacles are in order from lowest to top (visually on the screen), by comparing the lowest y Value of on obstacle to the highest y Value of the player we can determine if the obstacle is still above the player, out of reach, and we know that no further obstacle could possibly collide with the player as they are all higher up the screen – allowing us an opportunity to exit the loop early.

### 5.2.1 Original Code

```
1. // tutorial_level.cpp
2. if (event.type == sf::Event::MouseMoved) {
3.     int x = event.mouseMove.x;
4.     if (x < 32) x = 32; else if (x > 373) x = 373;
5.     player.moveTo(x, player.getPosition().y, 0.025f);
6. }
```

### 5.2.2 Optimised Code

```
1. // tutorial_level.cpp
2. if (event.type == sf::Event::MouseMoved)
3.     player.moveTo(std::clamp(event.mouseMove.x, 32, 373), player.getPosition().y, 0.025f);
```

### 5.2.3 Explanation

I originally implemented my player mouse movement check by first storing the x position into a new integer value, performed between 1-2 condition checks and re-assignments, before finally moving the player.

The optimised code now skips the assignment of the value and the clamping code and instead performs the clamp in the parameter of the moveTo method.

This creates easier to read, organised code and removes unneeded extra calls.

### 6. Testing

Test No	Description	Input Data	Expected Outcome	Actual Outcome	Notes / Action Required
1	Check that the player cannot move off the window.	Mouse movement	Player stops when reaching the edges of the window and does not push past them.	As expected, player stops between bounds of the window and does not move out of them.	N/A
2	Colliding with a static obstacle in tutorial mode causes a bounce and/or time rewind		At the start of the tutorial, hitting the starting obstacles rewinds time, but future obstacles cause damage.	As expected, obstacles cause the time to rewind and give the player another chance to progress.	Collision on traffic cones is poor due to box-shaped collision not matching art – considering replacing with something that fits better in a box collider.
3	Check if the timings in the Tutorial Level are appropriate.		The player has enough time to react to obstacles. The helper text lasts an appropriate amount of time.	Obstacles are avoidable but the is tight and requires very quick movement for a first-time tutorial.  Text helpers last for too long, which is boring having to wait.	Place more space between obstacles in Tutorial Level, and reduce the amount of time the text helpers last for and/or add the ability to click skip the text via click/key-press.

## **7. Conclusions**

The final game is not to the quality of what I had planned from the beginning, it's missing features, still has bugs, and whether I would call it a "fun" game I'm unsure...

However, what I did manage to do was implement a working scene system, correctly transition between scenes safely by moving to the next one and destroying the previous, wasting little to no resources.

I have shown evidence of optimisation, while I didn't improve as much as I wanted and know there is plenty I could've touched on such as multiple loops occurring in the obstacle manager that could've maybe been merged, I'm happy with the progress I've made.

## **8. Further Work**

If I had more time, I would invest more in optimisation, starting with the Texture Caching I had planned but unfortunately did not manage to implement in time. I think having to load the texture every time an obstacle is spawned is a waste of resources and is potentially causing slow down in my game at high speeds.

I would also invest more time into learning how to program an actual speed curve with a maximum value, for the speed of Level One, the endless version which just keeps speeding up.

I would also want to redo all the randomness, as I don't feel it's giving appropriate values.

I would also spend more time on this report, being more detailed – unfortunately I can only blame myself more poor time management despite my best attempts and lack of ADHD meds, and burn out.

## 9. References

Bakaoukas, Anastasios G. (2025). CPP-SFMLTemplate.zip.

[https://nile.northampton.ac.uk/ultra/courses/\\_155065\\_1/outline](https://nile.northampton.ac.uk/ultra/courses/_155065_1/outline)