# SPL-1 Project Report, 2019

**Racing Game**

(Pseudo 3d)

**SE 305: Software Project Lab 1**

**Submitted by**

**Junaid Mansur Ifti**

**BSSE Roll No. : 1027**

**BSSE Session: 2017-18**

**Supervised by**

**Dr.Noushin Nower**

**Designation: Associate Professor**

**Institute of Information Technology**

**Institute of Information Technology**

**University of Dhaka**

**[29-05-2019]**

**Table of Contents**

**1. Introduction** **3**

**1.1. Background Study** **3**

**1.2. Challenges** **3**

**2. Project Overview** **3**

**3. User Manual** **3**

**4. Conclusion** **3**

**5. Appendix** **3**

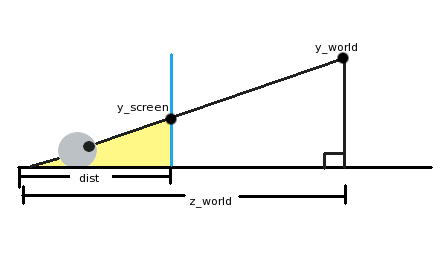
**References** **3**

**1.Introduction**

The **two-and-a-half-dimensional** (**2.5D**, alternatively **three-quarter** and **pseudo-3D**) perspective is either [2D graphical projections](https://en.wikipedia.org/wiki/2D_computer_graphics) and similar techniques used to cause images or scenes to simulate the appearance of being [three-dimensional](https://en.wikipedia.org/wiki/Three-dimensional_space_(mathematics)) (3D) when in fact they are not, or [gameplay](https://en.wikipedia.org/wiki/Gameplay) in an otherwise three-dimensional [video game](https://en.wikipedia.org/wiki/Video_game) that is restricted to a two-dimensional plane with a limited access to the third dimension

Now that every system can produce graphics consisting of a zillion polygons on the fly, why would you want to do a road the old way? Aren't polygons the exact same thing, only better? Well, no. It's true that polygons lead to less distortion, but it is the warping in these old engines that give the surreal, exhilarating sense of speed found in many pre-polygon games. Think of the view as being controlled by a camera. As you take a curve in a game which uses one of these engines, it seems to look around the curve. Then, as the road straightens, the view straightens. As you go over a blind curve, the camera would seem to peer down over the ridge. And, since these games do not use a traditional track format with perfect spatial relationships, it is possible to effortlessly create tracks large enough that the player can go at ridiculous speeds-- without worrying about an object appearing on the track faster than the player can possibly react since the physical reality of the game can easily be tailored to the gameplay style.

**A Mathematical Detour: 3d Perspective Projection**  
There are ways to get rid of the oatmeal effect. However, some traditional 3d mathematics are needed to make them possible. What we need is a way of translating 3d coordinates so that they fit onto a 2d surface.



In the picture above, an eyeball (lower left) is looking through the screen (the blue vertical line) at an object in our 3d world ("y\_world"). The eyeball is a distance "dist" from the screen, and a distance "z\_world" from the object. Now, one thing you might have noticed if you've spent some time with geometry or trigonometry is that there are not one but two triangles in the picture. The first triangle is the largest one, from the eyeball over to the ground on the right side and up to the object we're looking at. The second triangle I've colored yellow. This is from the eyeball to where on the screen we'll see our object, down to the ground, and back.

These two triangles' hypoteneuses (the line from the eye to the object) are at the same angle even though one is longer than the other. They are essentially the same triangle, but the smaller one is just scaled down. What this implies is that the ratio of the horizontal and vertical sides must be the same! In math terms:

y\_screen/dist = y\_world/z\_world

What we need to do now is juggle the equation to get y\_screen. This gives us:

y\_screen = (y\_world\*dist)/z\_world

In summary, to find the y coordinate of an object on the screen, we take the y world coordinate, multiply that by the distance we are to the screen, and then divide it by the distance it is in the world. Of course, if we just do that, the center of our view is going to be the upper-left corner of the screen! Just plug in y\_world=0 to see this. What we can do to center it is add half of our screen resolution to the result to put it right in the middle. The equation can also be simplified a little bit by pretending that our noses are right up to the screen. In this case, dist=1. The final equation then is:

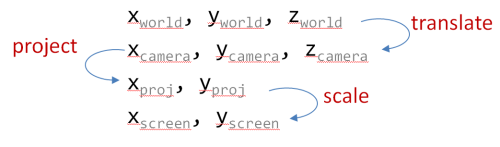
y\_screen = (y\_world/z\_world) + (y\_resolution/2).

Coordinate Systems

This sounds nice and simple in diagram form, but once you start coding its easy to get a little confused because we have been a bit loose in naming our variables and its not clear which represent 3d world coordinates and which represent 2d screen coordinates. We’ve also assumed that the camera is at the origin of our world when in reality it will be following our car.

More formally we should be:

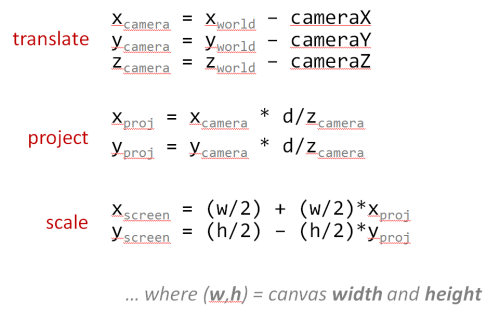
1. **translating** from world coordinates to camera coordinates
2. **projecting** camera coordinates onto a normalized projection plane
3. **scaling** the projected coordinates to physical screen (in our case canvas) coordinates



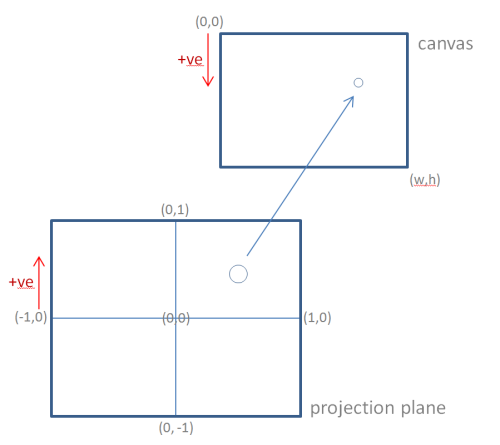
*NOTE: in a true 3d system a rotation step would come between steps 1 and 2, but since we’re going to be faking curves we dont need to worry about rotation*

Projection

And so we can present our formal projection equations as follows:



* The **translate** equations calculate the point relative to the camera
* The **project** equations are variations of our ‘law of similar triangles’ above
* The **scale** equations take into account the difference between:
  + *math* - where 0,0 is at the center and the y axis goes up and
  + *computers* - where 0,0 is at the top-left and the y axis goes down, as shown below



*NOTE: In a full blown 3d system we would more formally define a Vector and a Matrix class to perform more robust 3d mathematics, and if we were going to do that then we might as well just use WebGL (or equivalent)… but thats not really the point of this project. I really wanted to stick to old-school ‘just-enough’ pseudo-3d to build an outrun-style game.*

Simple and Fast Multimedia Library:

**Simple and Fast Multimedia Library** (**SFML**) is a [cross-platform](https://en.wikipedia.org/wiki/Cross-platform) software development [library](https://en.wikipedia.org/wiki/Library_(computing)) designed to provide a simple [application programming interface](https://en.wikipedia.org/wiki/Application_programming_interface) (API) to various multimedia components in computers. It is written in [C++](https://en.wikipedia.org/wiki/C%2B%2B) with [bindings](https://en.wikipedia.org/wiki/Language_binding) available for [C](https://en.wikipedia.org/wiki/C_(programming_language)), [Crystal](https://en.wikipedia.org/wiki/Crystal_(programming_language)), [D](https://en.wikipedia.org/wiki/D_(programming_language)), [Euphoria](https://en.wikipedia.org/wiki/Euphoria_(programming_language)), [Go](https://en.wikipedia.org/wiki/Go_(programming_language)), [Java](https://en.wikipedia.org/wiki/Java_(programming_language)), [Julia](https://en.wikipedia.org/wiki/Julia_(programming_language)), [.NET](https://en.wikipedia.org/wiki/.NET_Framework), [Nim](https://en.wikipedia.org/wiki/Nim_(programming_language)), [OCaml](https://en.wikipedia.org/wiki/OCaml), [Python](https://en.wikipedia.org/wiki/Python_(programming_language)), [Ruby](https://en.wikipedia.org/wiki/Ruby_(programming_language)), and [Rust](https://en.wikipedia.org/wiki/Rust_(programming_language)).[[3]](https://en.wikipedia.org/wiki/Simple_and_Fast_Multimedia_Library#cite_note-SFML_Bindings-3) Experimental mobile ports were made available for [Android](https://en.wikipedia.org/wiki/Android_(operating_system)) and [iOS](https://en.wikipedia.org/wiki/IOS) with the release of SFML 2.2.[[4]](https://en.wikipedia.org/wiki/Simple_and_Fast_Multimedia_Library#cite_note-Mobile_ports-4)

SFML handles creating and input to [windows](https://en.wikipedia.org/wiki/Window_(computing)), and creating and managing [OpenGL](https://en.wikipedia.org/wiki/OpenGL) contexts. It also provides a graphics module for simple [hardware acceleration](https://en.wikipedia.org/wiki/Hardware_acceleration) of [2D computer graphics](https://en.wikipedia.org/wiki/2D_computer_graphics) which includes text rendering using [FreeType](https://en.wikipedia.org/wiki/FreeType), an audio module that uses [OpenAL](https://en.wikipedia.org/wiki/OpenAL) and a networking module for basic [Transmission Control Protocol](https://en.wikipedia.org/wiki/Transmission_Control_Protocol) (TCP) and [User Datagram Protocol](https://en.wikipedia.org/wiki/User_Datagram_Protocol) (UDP) communication.

1.2 Challenges

Implementing a new Software application always carries a number of challenges.The process was complex,lengthy and heavy.Some of the challenge I have faced:

* Handling large code for first time.
* Learning a new Library Software SFML was lengthy and challenging.As I had to learn all the new syntaxes,had to understand works of each classes and API Documentations.
* Merging graphical images and C++ code for the first time.
* Had to learn C++ class based operations,instance of classes.Uses of vectors.
* The plane transformation from 3D Plane to 2D plane was the biggest challenge of this project as it was some heavy mathematics and a perspective illusional implementation of Objects which is complex to understand in real world.
* Learning new algorithms.

2. Project Overview

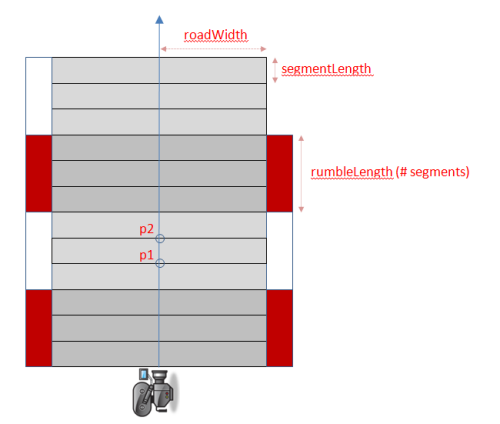
The Whole Project is divided into following parts:

1. Making of Raster Road.
2. Making of the Perspective of a Whole Racing Track.
3. Positioning of the car, Handling Steering by Input Handling & The Race Track Loop
4. Sprites and Texture Rendering.
5. Time maintenance and Collision Detection.

2.1 Making of the Raster Road

## Road Geometry

Each of these road segments will eventually be projected from their world coordinates to become a 2d polygon in screen coordinates. We store 2 points for each segment, **p1** is the center of the edge closest to the camera, while **p2** is the center of the edge farthest from the camera.



Technically, each segments **p2** is identical to the previous sections **p1** but we will find it easier to maintain them as separate points and transform each segment independently.

The reason we maintain a separate rumbleLength is so that we can have fine detailed curves and hills but still have long rumble strips. If each alternating segment was a different color it would create a bad strobe effect. So we want lots of small segments, but group them together to form each rumble strip.

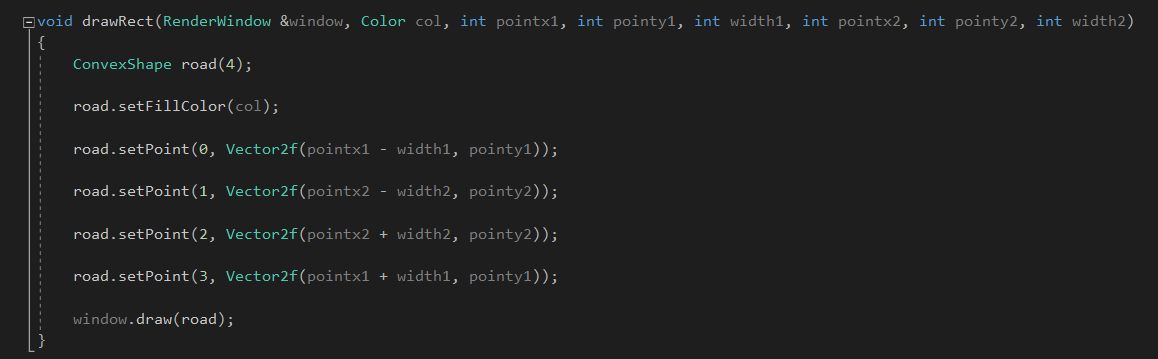
We initialize **p1** and **p2** with only **z** world coordinates because we only need straight roads. The **y** coordinates will always be 0, while the **x** coordinates will always be based on a scaled +/- roadWidth. This will change later when we add curves and hills.

We also setup empty objects to store the camera and screen representations of these points to avoid creating lots of temporary objects during every render - trying to keep our garbage collection to a minimum we want to avoid allocating objects inside our game loop whenever possible.

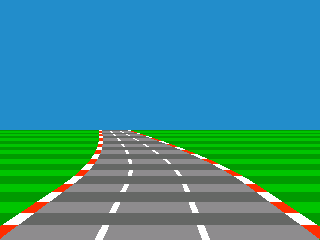
When the car reaches the end of the road we will simply loop back to the beginning. To make this a little easier we provide a method to find the segment for any Z value even if it extends beyond the length of the road:

The Coding Implementation:

SFML ConvexShape class is used here to implement the Road.

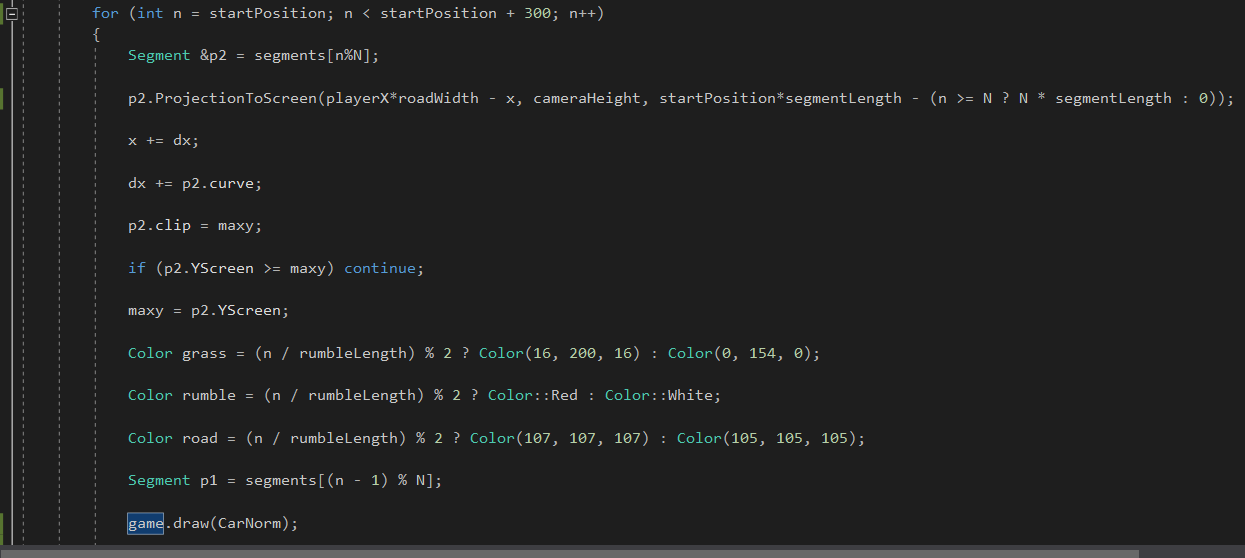


A pseudo 3D Road looks like this:



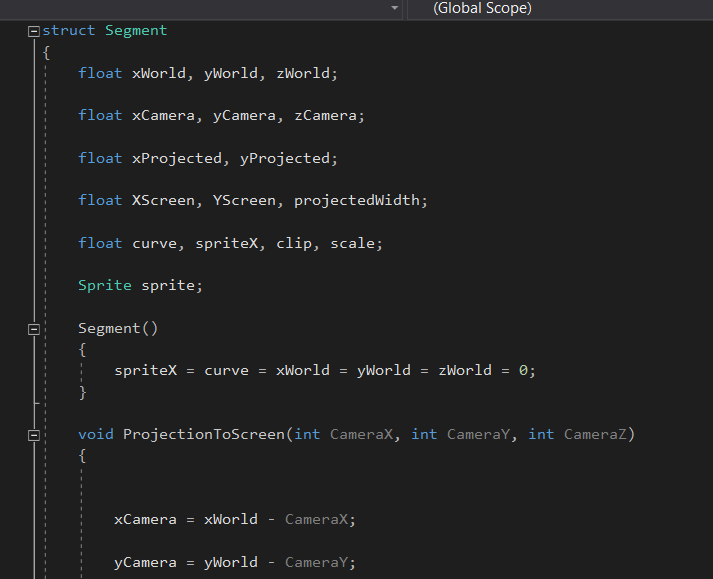
The Green color is Grass,The Red-white color things are rumble and the other half in grey color is the main road.Half of the height of the screen makes a Sky effect.

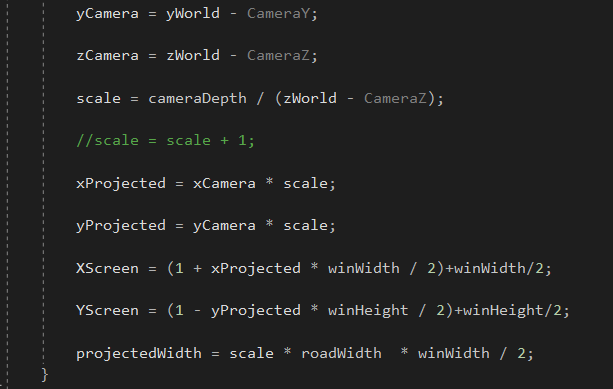
The different parts of road is rendered in this code Portion:



2.2 Making The Perspective of a Whole Racing Track

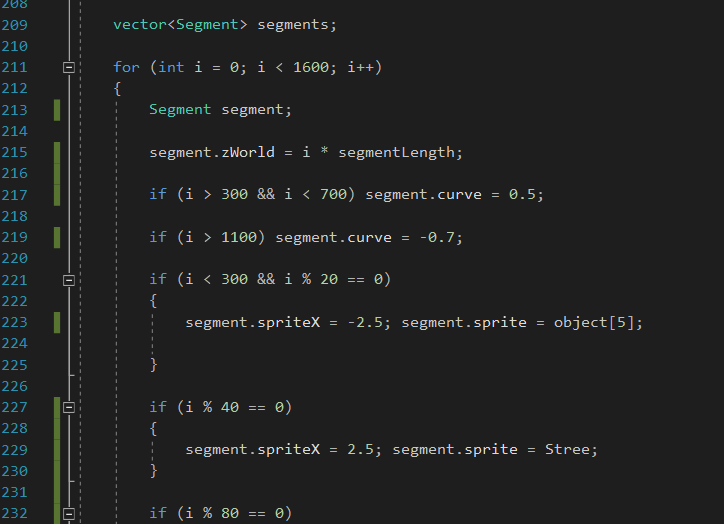
The road is divided into small segments which are confined by two line structures.Both of the line has perspective width of their own.To make a whole racing track some arbitray number of segments(here 1600) are taken and in between this number of segments there are straight roads and curvature and hills.And this 1600 segments are then repeated by maintaing the position of the Player.The Segments are stored in a Vector as the Repetative segments goes on and make the road infinite.



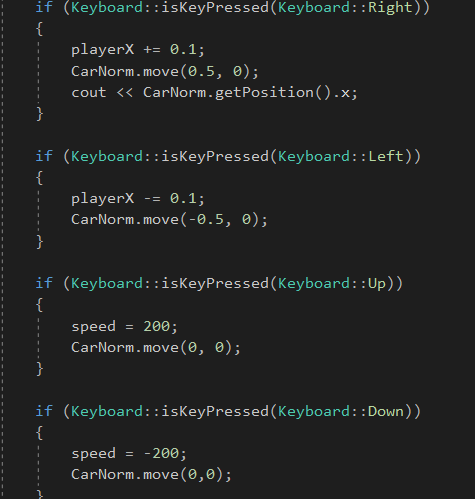


The 3D to 2D projection part is implemented above

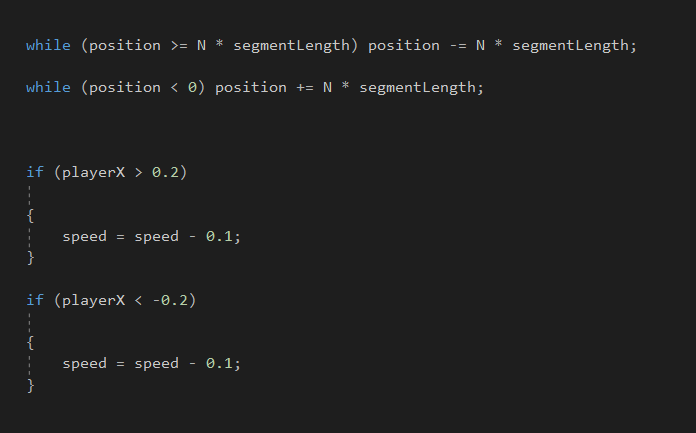
And the track is stored in a Vector



2.3 Positioning of the car and Handling Steering with User Input



Here upon user input the Car entity sprite is moved by some particular scale and also the camera is also moved alongside.



If the road is above some arbitrary value then the position is changed and the road track is repeated from the start if moved forward and repeated from the farthest to nearest if we move backward.

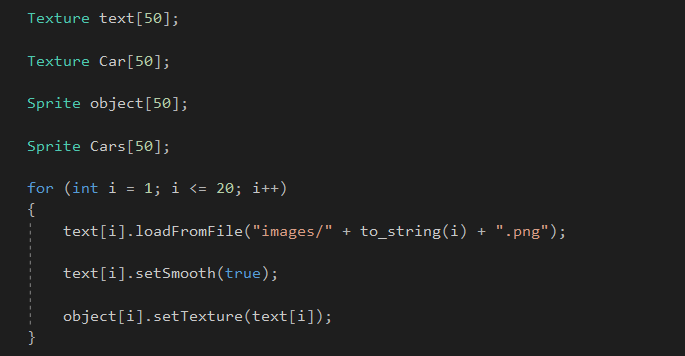
2.4 Sprite and Texture Rendering

A texture is an image. But we call it "texture" because it has a very specific role: being mapped to a 2D entity.

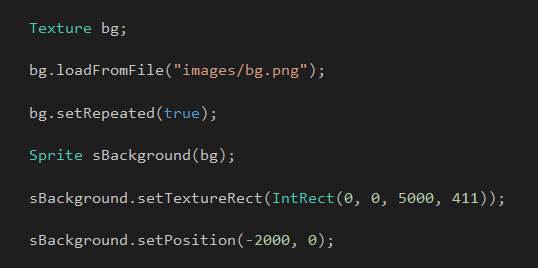
A sprite is nothing more than a textured rectangle.

Sprite Class is used to render any external image in the window.In the code Sprites used with texture to Draw the car,The Environment and Sky.

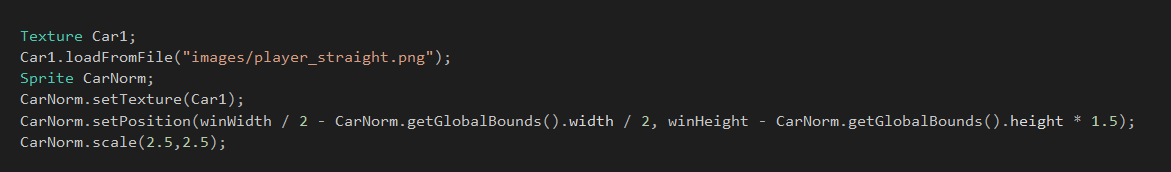
For the Roadside Objects:



For the background Image:

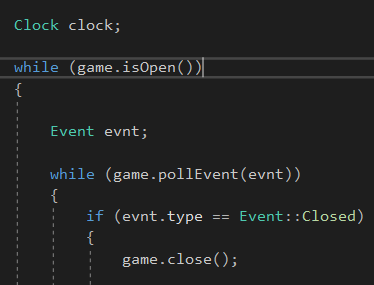


And For The Car:

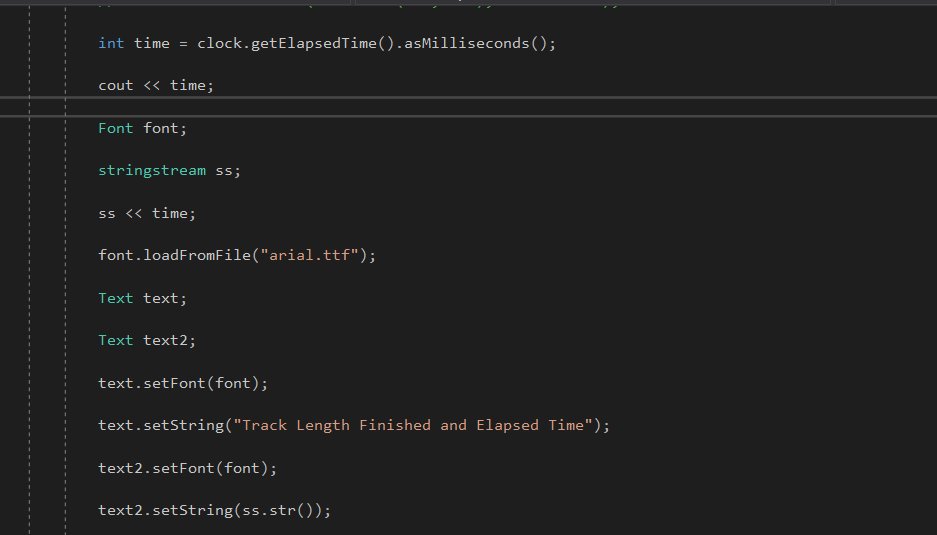


2.5 Time Maintain ace and Collision Detection

Clock Handling and Race End Time Showing:



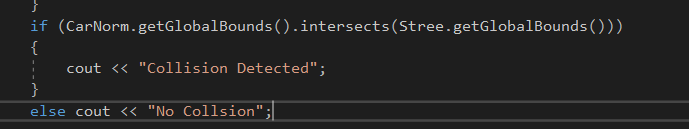
Clock is started above



The race track length is finished and Elapsed time at exact finishing time is shown in the window.

**Collsion Detection:**

Bounding Box Collision Detection:



Here every smallest Entity rectangle around sprite will collide with each other in order to make collision.

3.User Manual

After we open the executable file we will see a game:

1.



The Game will response for user input(Up,Down,Left,Right)

User must travel a particular length in time.The time will be shown and the best timer will be the highest scorer:

2.



Some caurves above.

3.



A particular track length finished and the timing is shown.

4. Conclusion

Implementing this game helped me grow my C++ knowledge as well as ensured a great use of my Mathematics knowledges.Raster road and Pseudo 3d was completely new to me and was quite challenging. It helped me grew confidence and experience.

5. Appendix

In this project, I have implemented the Pseudo3D Conecpt,TheRaster Road Effect,Euler’s

Plane transformation,Bounding Box Collission,Painter’s algorithm.

6. Reference

1. <https://www.sfml-dev.org/index.php>

2. <http://www.extentofthejam.com/pseudo>

3. <https://www.construct.net/en/tutorials/pseudo-3d-games-318>

4. <https://codeincomplete.com/posts/javascript-racer-v1-straight/>