

Contents

* Introduction
* Code structure
* Car Mechanics
* Track creator
* Texture Management
* Collision Detection and impulse resolution
* Testing and efficiency
* Conclusion

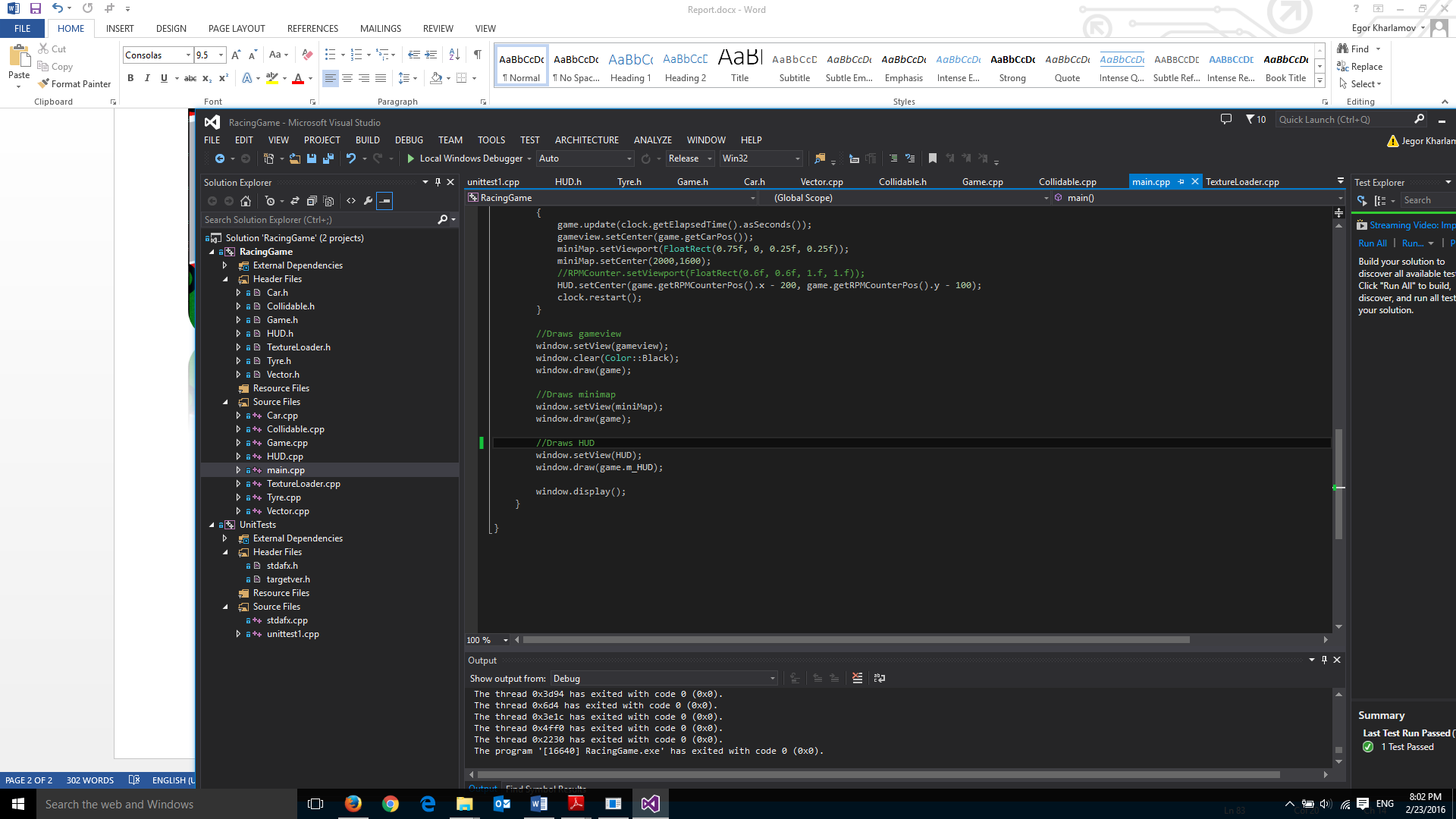
Introduction

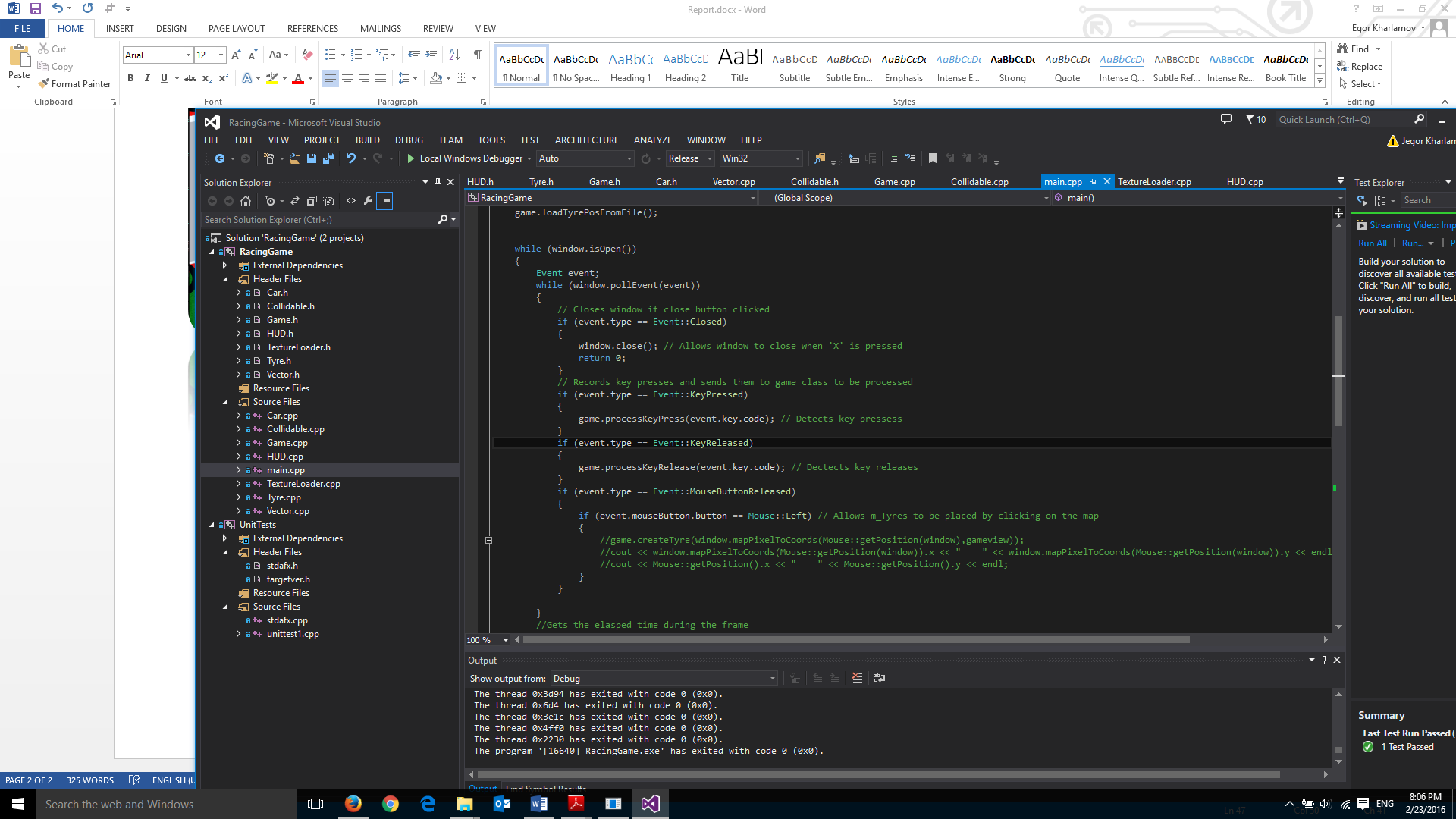
My final product is a high quality racing game with realistic driving physics and polymorphic collision impulse resolution. I have met all required criteria from the coursework specification and have gone beyond it to add additional features such as my advanced heads up display, track creator and RPM/Gear based acceleration. I am very happy with the final product I have produced and although I will continue to expand on it in my free time but in its current state it is a fun and very playable game that I thoroughly enjoyed creating.

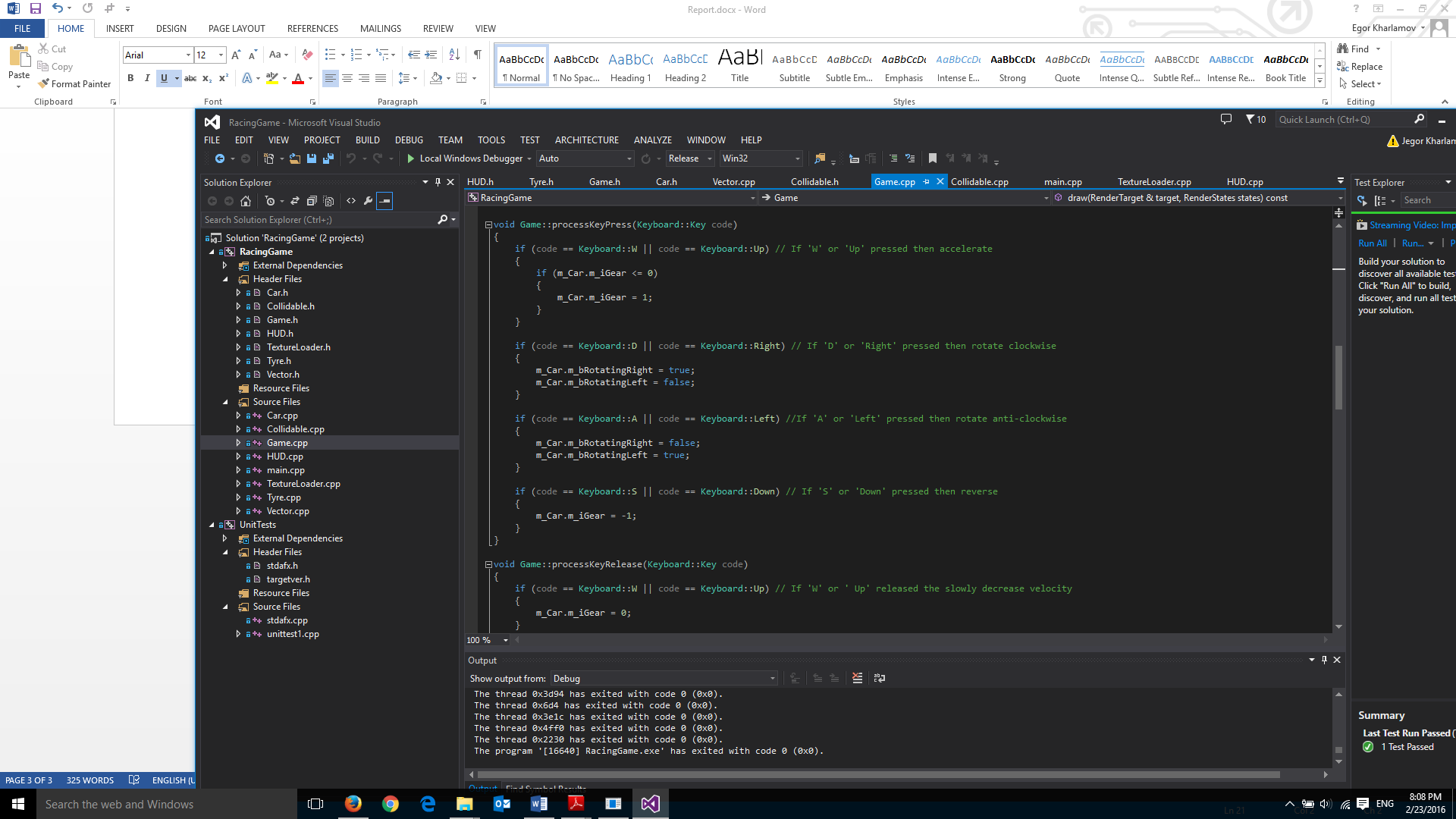
Code Structure

The structure of the code is like a pyramid. My *main.cpp* is in charge of all the sf::Views and drawing all the drawables given to *Game.cpp* to the screen while all other classes are controlled within the *Game.cpp* in order to create the actual gameplay such as acceleration of the car or collisions between collidable.

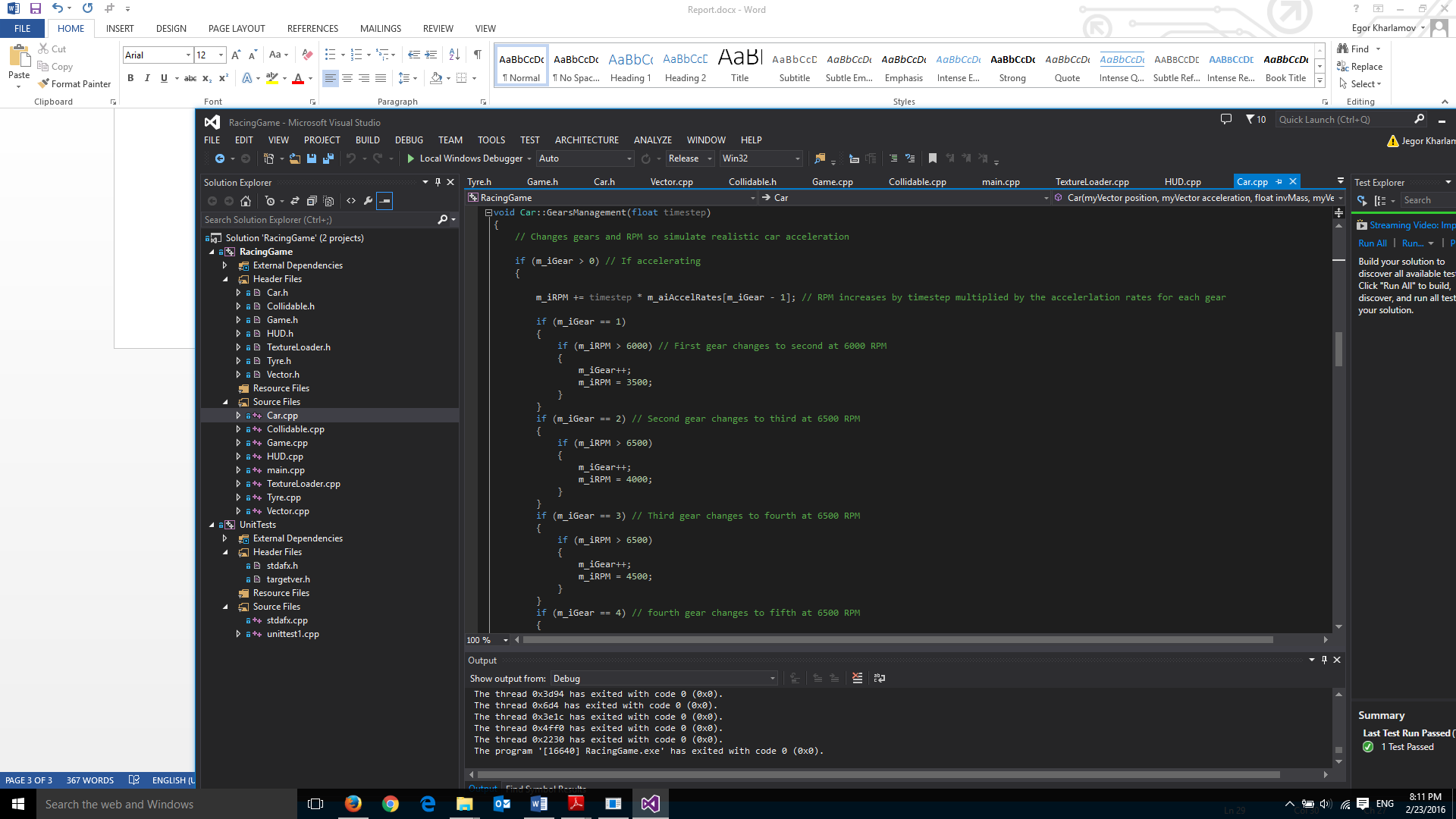
*Main.cpp* has three sf::Views. Once of which is for my Game View which is centred on the player car and is the big view that takes up the majority of screen space; the Heads Up Display view is instead not centred on the car so that the HUD remains constant on the screen. The HUD incorporates my RPM counter, Gear counter and timers. The final sf::View is the minimap which is located at the top right of my screen and is actually the same screen as Game View but zoomed out 4 times as much. I feel all the screens work well in unison in order to show the player all the information he/she needs to play the game.

**Note:** My Heads Up display is not drawn in game so I can draw it specificly in only the HUD window in main.

All keyboard keypresses are taken in my *Main.cpp* to be proccessed in *Game.cpp* .e.g.



You can clearly see that if the player presses the ‘W’ key and the player is in a gear less than 0 then it will cause the game to enter first gear and begin accelerating. From here *Car.cpp* will take over with its GearManagement() function.



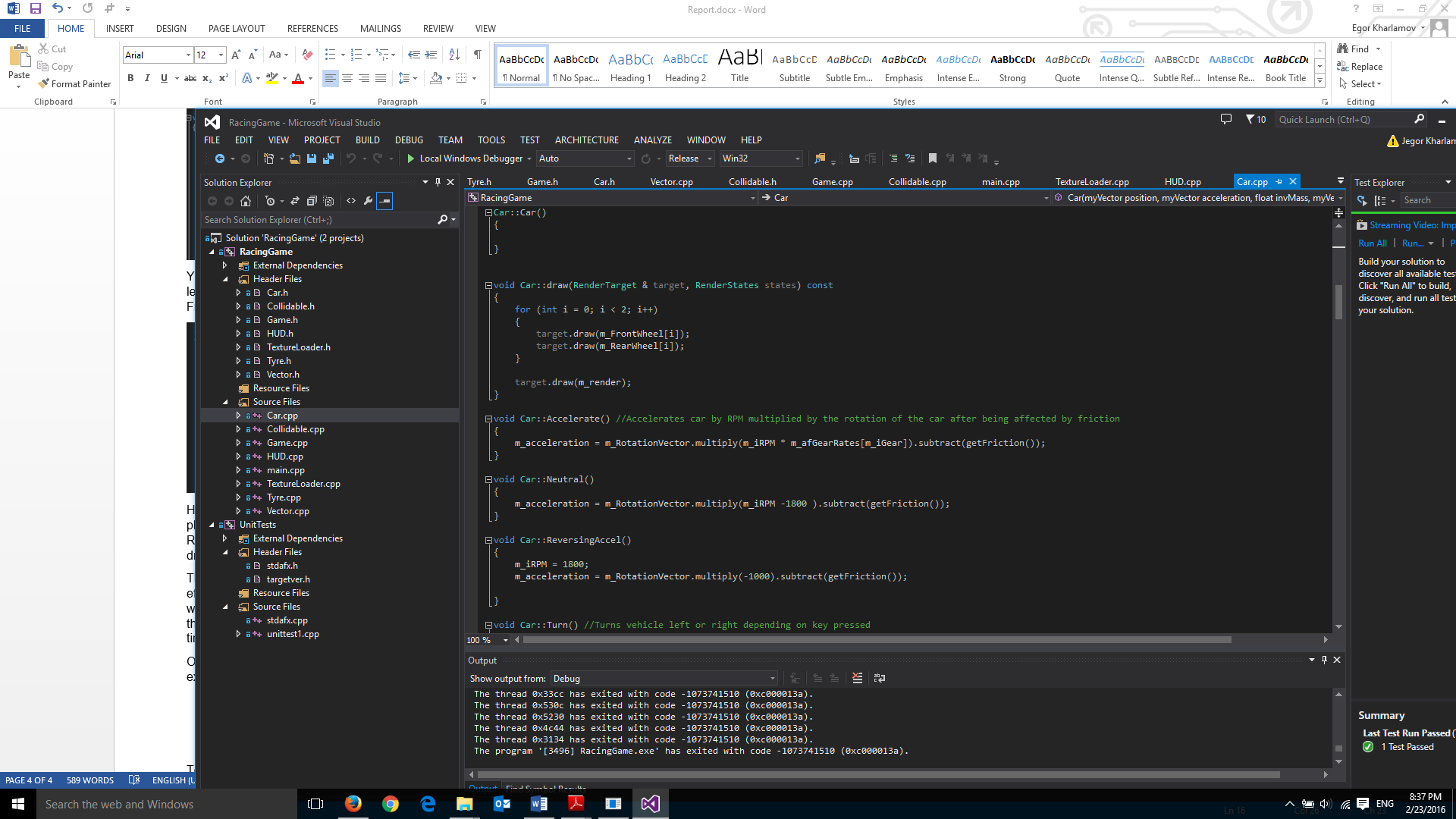
Here you can see my *Car.cpp* function which will manage the gears as long as the player continues to hold down the accelerate button by revving up the car. The cars RPM will increase until it hits a boundary then the gear will change and the RPM will drop simulating a gear change.

This is just one of the examples where I have used the object oriented nature of C++ effectively in order to create the game. Another example would be my texture loader which loads in all the textures that the sprites in my game need and then assigns them to their sprites in game to avoid having a having the texture loaded in every time an new instance of that class is created.

Overall, I feel I have structured my code for this project very well and I shall further expand into the structure as I discuss the more specific aspects of the code below.

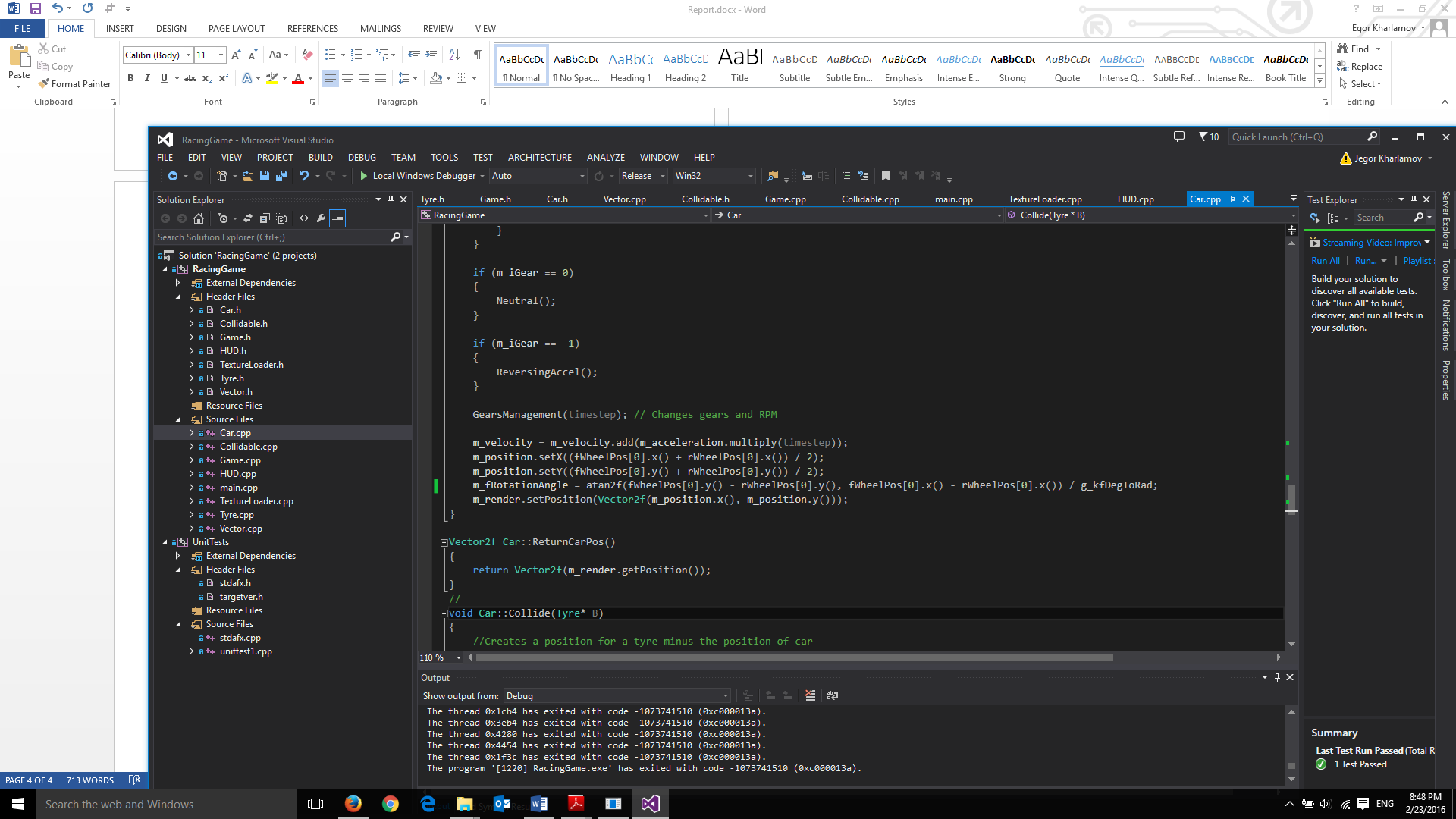
Car Mechanics

To create a car in my game you need to pass it 4 parameters which are a position, acceleration, inverse mass and velocity. These will create a car which can accelerate, turn and collide. I have briefly explained the cars acceleration above but it to quickly reiterate it uses a more advanced method of accelerating by using gear and RPM. Here is how I create my acceleration vector before using it in Euler in order to accelerate.



As you can see my acceleration is equal to my rotation vector (Which ensure the car travels in the correct direction) multiplied by my RPM multiplied by a specific acceleration of a gear rate then I subtract a friction vector from it. This creates a very fluid and responsive driving simulation in my game.

On top of this acceleration method I have also included the bicycle method of steering my car which means the cars position is based on the wheels of the car. I admittedly had some trouble with adding this into my game but in the final build I am submitting happy to say it is fully functioning.

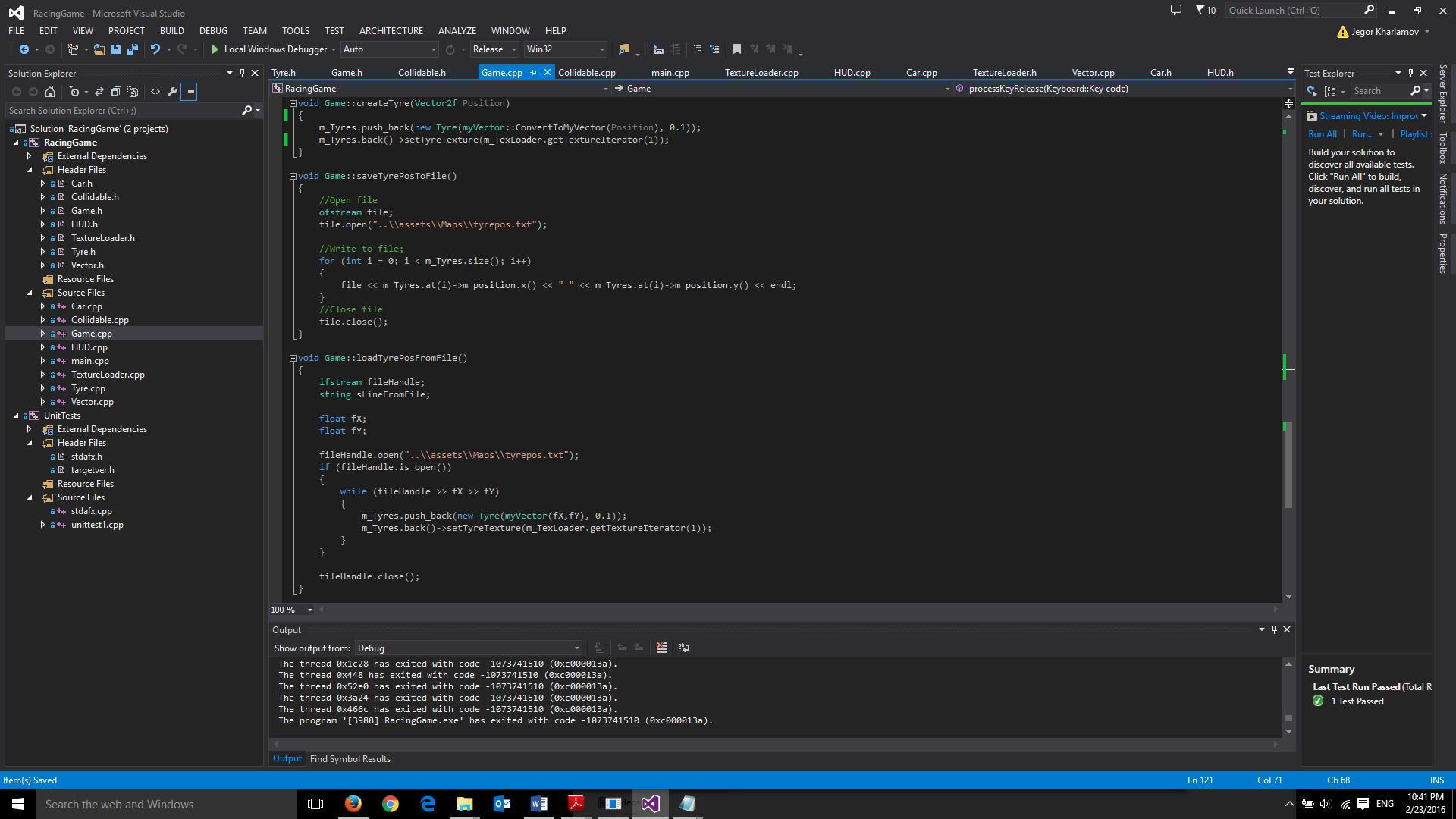


Above is my final Euler intergration using bicycle method. The position of the car is dependent on the midway point between the front and back tyres and the rotation of the car is dependent on the atan of front wheel positions minus the backwheel positions converted to radians. I found the intergration of the byicicle method challeneging but am happy I was able to intergrate it as it adds quite a lot to the game.

The turning of the car in the code works by adding or subtracting from a float which is then limited to 30 or -30 so the wheels do not turn unnaturally and then added on to the rotation vector of the wheels and there for the rotation of the sprite.

Track Creator

While it is rather basic and has much room still left for improvement you are able to build and save to file basic tracks while ingame. It is one feature I would defiantly like to develop further if I had more time but currently the player can place tyres by clicking on a location while ingame. When the player clicks a tyre is created and added to a .txt file in my assets folder. When the game loads it loads from the same file so you are able to place tyres anywhere you like and the next player could race around this course. For the sake of submission I have created a simple background circular race track and filled it with tyres. I have added the ability to save all the tyres you have put down by pressing enter and reload the tyres from the file by pressing backspace. In its current state it is functional but not flashy.



I tried to minimise input/output as its slow so I made sure to only use it when starting the game or pressing key such and enter or backspace.

Texture management