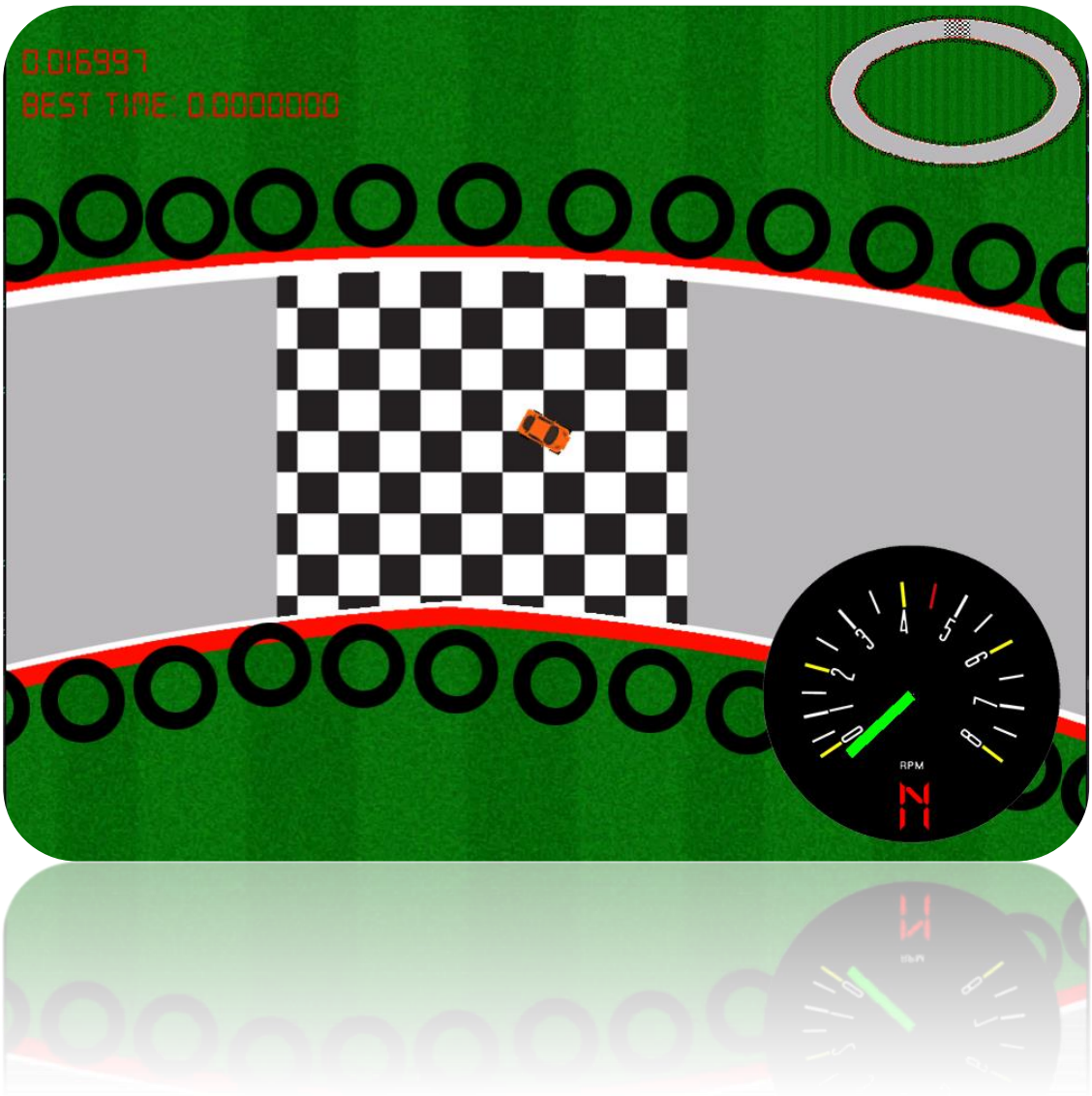


Contents

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 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 collidables.

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 unlike the Game View is 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.cpp* so I can draw it specificly in only the HUD window in main.

```
//Draws gameview
window.setView(gameview);
window.clear(Color::Black);
window.draw(game);

//Draws minimap
window.setView(miniMap);
window.draw(game);

//Draws HUD
window.setView(HUD);
window.draw(game.m_HUD);

window.display();
```

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

```
// Records key presses and sends them to game class to be processed
if (event.type == Event::KeyPressed)
{
    game.processKeyPress(event.key.code); // Detects key pressess
}
if (event.type == Event::KeyReleased)
{
    game.processKeyRelease(event.key.code); // Dectects key releases
}
if (event.type == Event::MouseButtonReleased)
```

```
void Game::processKeyPress(Keyboard::Key code)
{
    if (code == Keyboard::W || code == Keyboard::Up) // If 'W' or 'Up' pressed then accelerate
    {
        if (m_Car.m_iGear <= 0)
        {
            m_Car.m_iGear = 1;
        }
    }

    if (code == Keyboard::D || code == Keyboard::Right) // If 'D' or 'Right' pressed then rotate clockwise
    {
        m_Car.m_bRotatingRight = true;
        m_Car.m_bRotatingLeft = false;
    }
```

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

```
if (m_iGear > 0) // If accelerating
{

    m_iRPM += timestep * m_aiAccelRates[m_iGear - 1]; // RPM increases by t

    if (m_iGear == 1)
    {
        if (m_iRPM > 6000) // First gear changes to second at 6000 RPM
        {
            m_iGear++;
            m_iRPM = 3500;
        }
    }
    if (m_iGear == 2) // Second gear changes to third at 6500 RPM
    {
        if (m_iRPM > 6500)
        {
            m_iGear++;
            m_iRPM = 4000;
        }
    }
```

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 gears and RPM. Here is how I create my acceleration vector before using it in Euler in order to accelerate.

```
void Car::Accelerate() //Accelerates car by RPM multiplied by the rotation of the car after being affected by friction
{
    m_acceleration = m_RotationVector.multiply(m_iRPM * m_afGearRates[m_iGear]).subtract(getFriction());
}
```

As you can see my acceleration is equal to my rotation vector (Which ensures 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.

```
m_velocity = m_velocity.add(m_acceleration.multiply(timestep));
m_position.setX((fWheelPos[0].x() + rWheelPos[0].x()) / 2);
m_position.setY((fWheelPos[0].y() + rWheelPos[0].y()) / 2);
m_fRotationAngle = atan2f(fWheelPos[0].y() - rWheelPos[0].y(), fWheelPos[0].x() - rWheelPos[0].x()) / g_kfDegToRad;
m_render.setPosition(Vector2f(m_position.x(), m_position.y()));
```

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 a challenge 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. If no keys are held down the wheels will slowly default back to the centre position, I noticed this makes it much easier to drive.

Track Creator

While it is rather simplistic 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.

```
RacingGame                                                    Game
void Game::createTyre(Vector2f Position)
{
    m_Tyres.push_back(new Tyre(myVector::ConvertToMyVector(Position), 0.1));
    m_Tyres.back()->setTyreTexture(m_TexLoader.getTextureIterator(1));
}

void Game::saveTyrePosToFile()
{
    //Open file
    ofstream file;
    file.open("..\\assets\\Maps\\tyrepos.txt");

    //Write to file;
    for (int i = 0; i < m_Tyres.size(); i++)
    {
        file << m_Tyres.at(i)->m_position.x() << " " << m_Tyres.at(i)->m_position.y() << endl;
    }
    //Close file
    file.close();
}

void Game::loadTyrePosFromFile()
{
    ifstream fileHandle;
    string sLineFromFile;

    float fX;
    float fY;

    fileHandle.open("..\\assets\\Maps\\tyrepos.txt");
    if (fileHandle.is_open())
    {
        while (fileHandle >> fX >> fY)
        {
            m_Tyres.push_back(new Tyre(myVector(fX,fY), 0.1));
            m_Tyres.back()->setTyreTexture(m_TexLoader.getTextureIterator(1));
        }
    }

    fileHandle.close();
}
```

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

My texture loader helps to hugely improve the efficiency of my game. The game in final state has approximately 200 tyres loaded in when you start the game, it would be seriously inefficient to load in a texture for every single tyre so the fact I have a texture loader that only loads in each texture once and uses that texture only once for every single tyre hugely increases efficiency and the difference can be seen on the CPU test in Visual Studio.

```
    vector<string>fileNames;
    fileNames.push_back("HUD\\RPM.png");
    fileNames.push_back("Sprites\\Tyre.png");
    fileNames.push_back("Maps\\Track.png");
    fileNames.push_back("Sprites\\Car.png");

    for (int i = 0; i < fileNames.size(); i++)
    {
        if (!m_Holder.loadFromFile(m_sBaseDirectory + fileNames.at(i)))
        {
            cout << "Could not load " << fileNames.at(i) << endl;
        }
        textures.push_back(m_Holder);
    }
}
```

The texture loader pushes back all the textures I need for my game into a vector of textures as seen above. I can then assign these textures to sprites by referring to the point in the vector where my desired texture is.

```
vector<Texture>::iterator TextureLoader::getTextureIterator(int index) //Allows the assigning of textures to sprites
{
    return vector<Texture>::iterator(textures.begin() + index);
}
```

Above function which takes the index as a parameter and then returns a texture through an iterator. Overall, this class is relatively simple but hugely improves the efficiency of the game.

Vector Class

As vector2fs lack mathematic functions such as Dot Product I have created my own vector class quite colloquially titled “MyVector”. I ensured these were interchangeable between vector2fs as many parameters in SFML require vector2fs e.g. positions for sprites.

```
Vector2f myVector::Convert2f() // Converts one of my vectors into a Vector2f. Necessary for specific functions in sfml
{
    Vector2f ConvertedVector(m_adVector[0], m_adVector[1]);

    return ConvertedVector;
}

myVector myVector::ConvertToMyVector(Vector2f vector)
{
    return myVector(vector.x, vector.y);
}
```

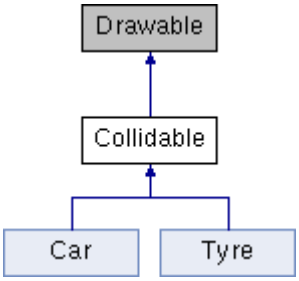
Note: Convert to myVector is a static function meaning I can call it without referring to a myVector.

I use this class very often throughout my code as many mathematic operations in my game are vector equations.

myVector contains overloaded functions that give me the ability to multiply by either another vector or a scalar without any extra effort. It also contains the dot product function which is used fairly commonly in my *Car.cpp* so a huge amount of code is not rewritten and instead reused by the existence of this class.

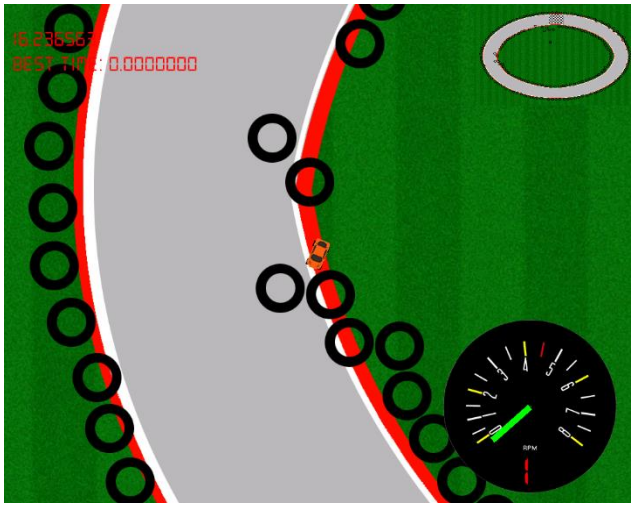
Collision and impulse resolution

The collision in my game is polymorphic. All classes that can collide with another object inherit from collidable.



Currently there are only two collidable objects: cars and tyres. However tyres can collide with each other e.g. Car crashes into tyre, tyre gets knocked into 3 more tyres then all tyres move.

```
virtual void Collide(Collidable* Object2){};
```



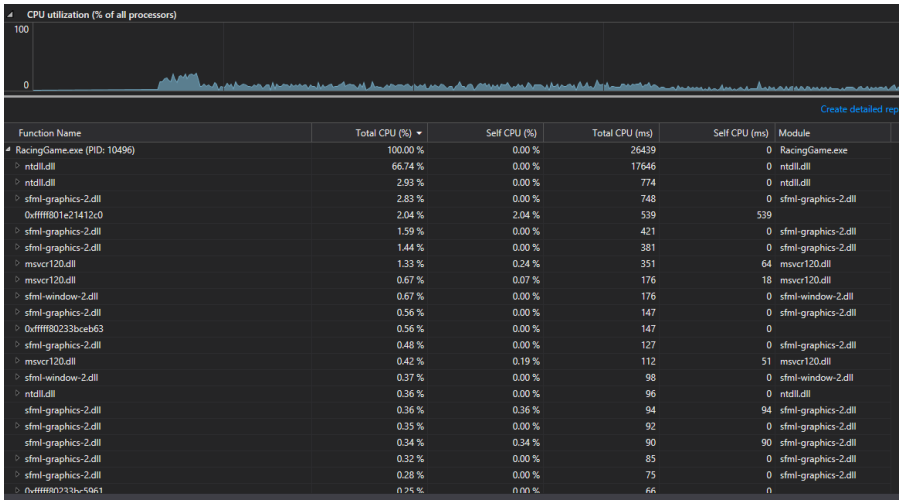
Above you can see the virtual function which allows a collidable to collide with another collidable. It is rewritten in both *Car.cpp* and *Tyre.cpp* as car collision uses the OBB-Circle clamp method while tyre-tyre collision uses Circle-Circle collision. They also resolve impulses differently.

In order to take rotation into account when colliding with the circle I had to rotate the problem. I created a new circle position which was at (0,0) and then subtracted the cars position from it. I then rotated this local tyre position by the negative rotation of my car so that rotation would be taken into account when colliding. Before I implemented this I had trouble with the collision always thinking the car was in the X axis and not taking rotation into account. As for impulse resolution I have a functioning system in place in the final build. It is rather prone to returning overflows which is why I was working on an improved method of collision, however, I did not have the time to get this new version working between car and tyre so I have removed it from the submitted build and let the old functioning method remain. With more time I would have surely implemented the better OOB-Circle collision. My tyre – tyre collision however has worked perfectly as expected and is quite entertaining to mess around with in game e.g. picture above.

Testing and efficiency

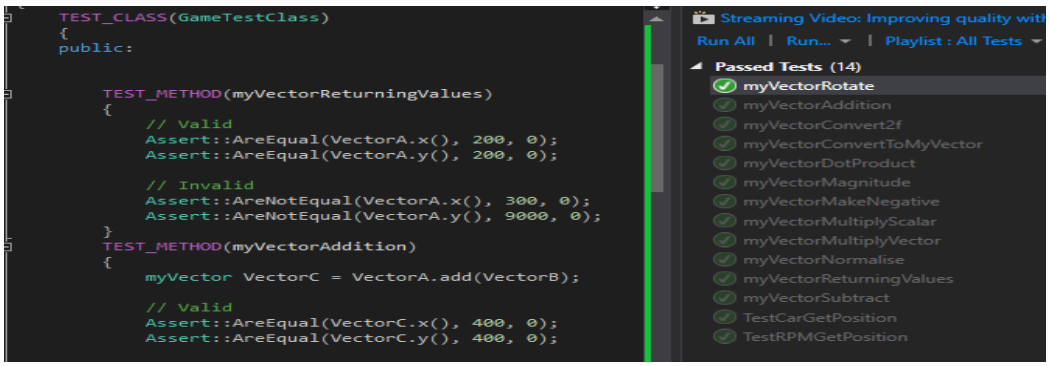
I have ran a CPU diagnostic on my game and found it to be very efficent. The most stressful parts on the CPU is the launch where everything needs to be drawn including the background and 200 tyres so the fact that it only requires 23.2% of the CPU at most is a very good sign towards efficency. The most demanding things a sf::Draw which is to be expected; when colliding more CPU is used but the game runs steadly throughout gameplay which

means I can happily say my code quite efficent. Reasons for my codes efficency include my Texture Loader which stops every sprite having a texture loaded in each time its created and my well planned use of object oriented design and polymorphism for collisions.



Function Name	Total CPU (%)	Self CPU (%)	Total CPU (ms)	Self CPU (ms)	Module
RacingGame.exe (PID: 10496)	100.00 %	0.00 %	26439	0	RacingGame.exe
ntdll.dll	66.74 %	0.00 %	17646	0	ntdll.dll
ntdll.dll	2.93 %	0.00 %	774	0	ntdll.dll
sfml-graphics-2.dll	2.83 %	0.00 %	748	0	sfml-graphics-2.dll
0xfffff801e21412c0	2.04 %	2.04 %	539	539	
sfml-graphics-2.dll	1.59 %	0.00 %	421	0	sfml-graphics-2.dll
sfml-graphics-2.dll	1.44 %	0.00 %	381	0	sfml-graphics-2.dll
msvcr120.dll	1.33 %	0.24 %	351	64	msvcr120.dll
msvcr120.dll	0.67 %	0.07 %	176	18	msvcr120.dll
sfml-window-2.dll	0.67 %	0.00 %	176	0	sfml-window-2.dll
sfml-graphics-2.dll	0.56 %	0.00 %	147	0	sfml-graphics-2.dll
0xfffff80233bceb63	0.56 %	0.00 %	147	0	
sfml-graphics-2.dll	0.48 %	0.00 %	127	0	sfml-graphics-2.dll
msvcr120.dll	0.42 %	0.19 %	112	51	msvcr120.dll
sfml-window-2.dll	0.37 %	0.00 %	98	0	sfml-window-2.dll
ntdll.dll	0.36 %	0.00 %	96	0	ntdll.dll
sfml-graphics-2.dll	0.36 %	0.36 %	94	94	sfml-graphics-2.dll
sfml-graphics-2.dll	0.35 %	0.00 %	92	0	sfml-graphics-2.dll
sfml-graphics-2.dll	0.34 %	0.34 %	90	90	sfml-graphics-2.dll
sfml-graphics-2.dll	0.32 %	0.00 %	85	0	sfml-graphics-2.dll
sfml-graphics-2.dll	0.28 %	0.00 %	75	0	sfml-graphics-2.dll
0xfffff80233bc5961	0.25 %	0.00 %	66	0	

I have also completed unit testing of my code which was mostly my vector class as almost all of my functions are voids. There are two setters in game which I also unit tested but the vast majority were my mathematical vector tests. I have completed a valid and invalid test for each one, I did not do a borderline test as it seemed unnecessary with such clear cut mathematical operations. I can confirm that all the functions in my Vector class are fully functional. It was quite time consuming to write out the unit tests but I do understand the advantages they can present especially in more complicated software.



```
TEST_CLASS(GameTestClass)
{
public:

    TEST_METHOD(myVectorReturningValues)
    {
        // Valid
        Assert::AreEqual(VectorA.x(), 200, 0);
        Assert::AreEqual(VectorA.y(), 200, 0);

        // Invalid
        Assert::AreNotEqual(VectorA.x(), 300, 0);
        Assert::AreNotEqual(VectorA.y(), 9000, 0);
    }
    TEST_METHOD(myVectorAddition)
    {
        myVector VectorC = VectorA.add(VectorB);

        // Valid
        Assert::AreEqual(VectorC.x(), 400, 0);
        Assert::AreEqual(VectorC.y(), 400, 0);
```


Conclusion

In conclusion, I am very happy with the final product I have produced and believe it is an excellent quality game that would become even better with more work. Had I had more time I would like to have implemented the new collision method I was working on for Car-Tyre collisions, Save to file leader board for high score, more features for the track creator e.g. more collidables, ability to load in other maps and a main menu as well as a soundtrack. I have very much enjoyed working on this game and am proud of the version I am submitting. I shall continue to add these features in my free time but I hope you have as much fun playing it as I did making it.