School of Electronics and Computer Science
Faculty of Engineering, Science and Mathematics
University of Southampton

Tim Jones
May 2004

Benchmarking the Performance of C# and C++ in the Context of a DirectX Game

Project Supervisor: Dr Andy Gravell
Second Examiner: Dr Paul Lewis

A project submitted for the award of
BSc Computer Science

Abstract

The purpose of this project was to show that the combination of the C# language and Managed DirectX can be used to create a complete 3D video game with a speed favourably comparable to that of a game created using the C++ language and unmanaged DirectX.

In order to show this, a car racing game has been written in both C# and C++. This game uses several APIs available within DirectX, including Direct3D, DirectInput, and DirectSound. These APIs allowed the development time to be spent creating the game itself without needing to write low-level code for the underlying hardware.

The physics model for the game was designed to simulate the movement of a vehicle across rough terrain. The longitudinal, lateral, and suspension forces acting on each wheel have been modelled. The combined effect of these forces on the car body was used to move the car. Collision detection and response between the car and track have also been implemented.

The graphics for the project were rendered using Direct3D, using a variety of rendering techniques designed to add realism and enhance believability. Sounds were implemented using DirectSound, and these include engine, road and impact sounds.

The game was written in C# first, and then converted to C++, with identical functionality for both versions. After both versions were completed, they were benchmarked against each other using two methods – simple frames per second (FPS) comparisons and code profiling.

It was found that at medium screen resolutions (1024x768) the C++ version generally outperformed the C# version by up to 30%. However, at higher resolutions (1400x1050), the performance of both versions was almost identical. These results indicate that as average screen resolutions increase, the performance of games written in C# will converge with the performance of C++ games.

Table of Contents

Acknowledgements

The following people contributed the 3D models used in this project. The Subaru Impreza model was provided by Ramiro Alcocer. The Ford Focus model was provided by Johny. The Eville Rally track was provided by Cubits. The Four Oaks track was provided by Gavin Clark.

I would like to thank David Paterson Watts for composing the two music tracks in the game.

Dr Andy Gravell has given invaluable help and encouragement as project supervisor.

I would also like to thank these people for their support: my dad, for asking insightful questions as well as proofreading; Gemma, for helping me keep going; Jason, for being willing to help with coding problems; and Kevin Murphy, for assisting with my gathering of knowledge about car physics.

Chapter 1 – Introduction

Game development costs are rising rapidly, while technology is becoming increasingly complex [1]. Following the Hollywood trend, the games industry is making fewer games for more money. Hardware accelerated graphics cards mean that more and more special effects can be included. It is important for the games industry to find ways to cut back on costs.

It has been claimed that only 20 per cent of developer time is used in the creative process, with the other 80 per cent spent writing routines and software to create the backbone of the game [2]. It follows that if the time spent on repetitive programming can be lessened, games could become much cheaper to develop – or they could include more features for the same cost. Games are traditionally programmed using some combination of C and C++, with the graphics API depending on platform. The focus of this report is on the PC platform, where the two prevalent graphics APIs are Direct3D and OpenGL. Since Direct3D is currently more common, this is the API that we shall focus on.

In January 2001, Microsoft released the first version of .Net, their new programming framework. Similar in many respects to the Java platform, .Net is a run-time environment that makes it much easier for programmers to write robust code quickly. It provides automatic garbage collection, can be targeted by many different languages, and comes with a large class library. The sophisticated garbage collection algorithms used by .Net require more computation than traditional heap allocation [3]. Additionally, any time a piece of "managed" code (code that runs entirely within the .Net framework) calls an "unmanaged" method (code that exists outside of .Net), it incurs a performance penalty.

DirectX is a set of APIs for Windows that allow programmers to directly access the underlying hardware without needing to go through the Windows API. These APIs mean that high-performance video games can be written to run within Windows. DirectX itself is unmanaged, but the latest version of DirectX includes a set of APIs known collectively as Managed DirectX, which allow .Net programmers to access the underlying DirectX methods but without most of the performance penalty usually associated with this type of operation. Microsoft have asserted that the speed of Managed DirectX is between 2% and 5% less than the speed of unmanaged DirectX [4].

In order to test this, the goal of this project was to create a car simulation game in both C# and C++. It was intended that each version use appropriate class libraries and coding methods for the language it was written in, to ensure that the eventual benchmarks are as fair as possible. For C#, this meant using Managed DirectX, discussed in section 2.3, and for C++, unmanaged DirectX. There were also other differences, discussed in section 4.3. The only previous attempt of this nature was a port of the Quake II source code from unmanaged C++ to managed C++, where performance of the two versions was then compared [5]. However, the managed C++ version still uses unmanaged DirectX, and so it is not a true .Net application.

Since the goal of the project was code-oriented, it was decided to use track and car meshes that were already available instead of creating models specifically for this game. To this end, two tracks (called Eville Rally and Four Oaks) were converted for use in the game, and two cars (the Subaru Impreza and Ford Focus) were used. The authors of these models are listed in the Acknowledgements.

After the broad goals of the project were established, the architecture of the game was put in place. This design is documented in Chapter 3. After the design had been finished, the coding phase could begin. Initially, the code was written in C# and then ported to C++. This presented some interesting issues, which are addressed in Chapter 4.

Once both versions had been coded, they could be benchmarked in order to compare performance. External testing also took place to ensure that the game was of a suitable standard for the benchmarks to be valid. The benchmarking and testing process is detailed in Chapter 5.

Chapter 2 – Background

There is no shortage of open source games available on the internet, written in both C++ and, increasingly, in C#. However, none of these games are available both in C++ (with unmanaged DirectX) and C# (with managed DirectX). Although it would have been possible to convert an existing game in C++ to C#, it would have been difficult to ensure that both games were operating in the same way, for the purposes of benchmarks. It was therefore necessary to implement a new game. This game needed to use all the main elements that would be present in most 3D games – along with 3D graphics, these include physics, keyboard input, and sound.

It was decided to write a 3D rally car game, since this would test not just Managed DirectX, but also the performance of .Net when running physics code, which tends to be floating point-intensive. Although previous tests have indicated that the floating-point performance of .Net is on a par with C / C++ [6], these tests were not performed in the context of a high-performance game, and the results may be different here. A car game also allows for the inclusion of engine and road sounds, as well as the usual keyboard input. These components allow the game to be a representation of the majority of games.

2.1 Previous Car Simulations

Several previous car simulations were studied, including Racer [7] and TORCS [8]. Racer is a free car simulation that aims to provide realistic car physics using professional car physics papers. It has a large community providing track and car models; indeed, it was from this community that the track and car models used in this project were obtained. Racer uses Pacejka's Magic Formula [9] as the basis for its physics model. This is the standard that is used in many non-game racing simulations. It uses a set of constants that relate to the capabilities of a specific tyre. However, these constants are not made available by tyre manufacturers, and it is not trivial to guess them.

TORCS is a racing car simulator that allows programmed robot drivers to race against each other. It is also possible for humans to race. The emphasis of this project appears to be on the robot programming rather than the car physics, resulting in some fairly arcade-like physics.

2.2 Physics

It is important that the physics in a car simulation game be fairly accurate so that the player feels that they are actually driving a car. There are many different ways of simulating physics, a few of which are discussed here.

At its most basic, the physics model must allow the player to accelerate, brake and turn the car. The simplest model accelerates the car linearly up to a limit, and turns the car directly based on player input. However, this is very unrealistic. For more realistic simulations, it is necessary to use an approximation to real-world car physics.

The first of these methods has already been mentioned, Pacejka's Magic Formula. Providing that the correct set of constants is supplied, this method results in a very accurate model. However, it was decided that choosing these constants would be time consuming and could detract from the general aim of the project.

The second physics model is a combination of slip ratio and slip angle. This is the model that was initially used in this project. Slip ratio is the ratio between the linear speed of the wheel and the speed of the car. Slip angle is the angle between the wheel and the current direction of the car. These concepts are described in more detail in section 3.2.

The third model is known as Rigid Body Dynamics [10]. Dynamics is the study of forces and masses that cause the kinematic quantities to change as time progresses. "Rigid Body" means that a constraint is placed on the objects, namely that its shape does not change.

Rigid Body Dynamics still uses slip ratio and slip angle to calculate per-wheel forces, but it is also able to simulate suspension accurately, and thus account for the changing height of terrain in a 3D environment. Although suspension can be modelled without using rigid body dynamics, they provide an integrated method of calculating physics. Rigid Body Dynamics were chosen for the physics engine for these reasons.

2.3 The DirectX API

Microsoft DirectX is a suite of multimedia application programming interfaces (APIs) that are built into Microsoft Windows. DirectX provides a standard development platform for Windows-based PCs by enabling software developers to access specialized hardware features without having to write hardware-specific code.

At the core of DirectX are its application programming interfaces, or APIs. The APIs act as a bridge for the hardware and the software to "talk" to each other. The DirectX APIs give multimedia applications access to the advanced features of high-performance hardware such as three-dimensional (3D) graphics acceleration chips and sound cards. They control low-level functions, including two-dimensional (2D) graphics acceleration; support for input devices such as joysticks, keyboards, and mice; and control of sound mixing and sound output [11].

Direct3D is the name of the DirectX API that allows developers to write graphics applications (such as games) that utilise the power of graphics accelerated hardware. DirectSound enables the playing of sounds and gives applications a high level of control over hardware resources. The DirectInput API is used to process data from a keyboard, mouse, joystick, or other game controller. There are other DirectX APIs, but these three APIs provide the core functionality that is necessary for this project.

2.4 Microsoft .Net

In 2002, Microsoft released the first version of the .Net framework, which fundamentally altered how developers write programs. Instead of writing directly for the underlying architecture, developers write code which is compiled to the Intermediate Language, which is then compiled at runtime to native code by the JIT (Just-In-Time) compiler. This introduces a level of abstraction from the complexities of the underlying platform, and means that development time can be significantly reduced. Code written for the .Net platform is known as "managed code", and traditional native code is known as "unmanaged code".

The latest version of DirectX includes Managed DirectX, which allows fast access to the native DirectX interfaces. Microsoft's aim for Managed DirectX was that it should perform at

between 95% and 98% of the speed of unmanaged DirectX. Part of the aim of this project is attempting to prove that this is the case. However, because there is more involved in a car simulation game than simply rendering graphics, this project also allows a comparison between C# and C++ for other types of code, such as mathematical and physical calculations.

2.5 Benchmarking

Benchmarking is the practice of identifying quantitative metrics against which the performance of a piece of software can be measured. For this project, benchmarking will involve finding metrics to compare the performance of the C# and C++ version of the game. These metrics will be discussed in more detail in Chapter 5. However, one of the techniques that will be used is code profiling, and it was necessary to find a piece of software that would allow both managed (C#) and unmanaged (C++) code to be profiled, so that like-for-like comparisons are possible.

Several free code profilers exist, but it was Compuware's DevPartner Profiler Community Edition that offered the best mix of features and cost (it was free). It was able to profile both C# and C++, which met the requirements for this project.

Chapter 3 – Design

3.1 Game Architecture

Since the game is a rally driving simulation, the core part of the game is obviously going to be the physical simulation of a car moving around a track. However, as with any game, there are other components to think about – for example, the GUI, engine and road sounds, and in-game music. The diagram below shows how all these components fit into the game architecture.

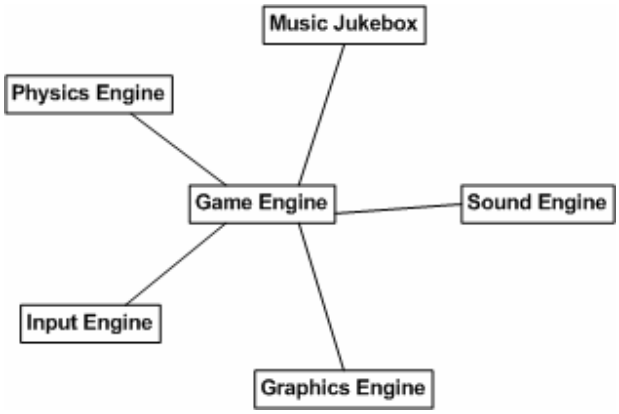


Figure 1 – Overview of the game architecture

These five main components (Physics Engine, Input Engine, Graphics Engine, Sound Engine, and GUI) are described in more detail in the following sections. The Music Jukebox simply loads and plays music files, with different music depending on where the player is within the game, and is not discussed here.

Almost all games share the same basic architecture [12], shown in the figure at the top of the next page. Once all necessary resources have been loaded, the program enters the "game loop". This loop is responsible for polling for player input, updating the state of the objects within the game world, and finally rendering the current representation of the game world to the screen. The visible result on screen after each of these loops is known as a "frame", and thus the number of frames per second generated gives us a useful indicator of performance. The target is to generate 60 frames per second to ensure smooth visual effects [13].

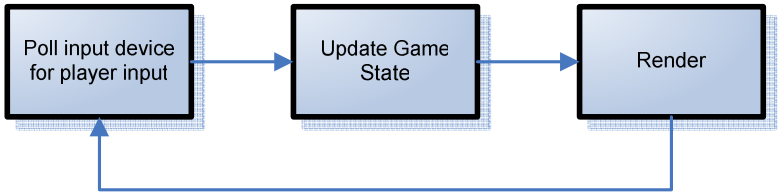


Figure 2 - Shows the basic game loop

The specifics of the "Update Game State" step shown in the diagram above depend on where within the game the player currently is. For instance, if they are navigating the menus, then only the GUI will be updated. However, if they are in the middle of a race, then this step includes collision detection and all the appropriate physics calculations.

3.2 Physics Engine

Probably the most important aspect of a car simulation is the physical modelling of the car's movement through its environment. Many different techniques have been used in previous car games, with increasingly realistic physics becoming more common each year. At its most basic, the physical simulation needs to allow the player to move the car by accelerating, braking, and turning left and right. However, in a 3D environment, this becomes more complex, because the car must respond to the changing height of the terrain it is travelling across, as well as colliding with solid objects.

In the first version of the physics engine, which was described in the progress report for this project, a primarily 2D simulation was used, where the longitudinal and lateral movement of the car was calculated based on the slip ratio (the ratio of the longitudinal velocity of a wheel and the car) and the slip angle (the difference between a wheel's angle and the angle of the car's movement).

However, in a 3D environment, there is the added complexity of modelling a car's suspension as it moves over bumps and jumps. This introduces a 3D rotational component to the car's orientation, which affects both the slip ratio and slip angle.

Lastly, the car needs to be able to collide with solid objects on the race track (collision detection), and respond to these collisions in a physically accurate manner (collision response).

In order to include suspension and collision detection and response, it was decided to switch to a rigid body simulation, as discussed in section 2.2. The final version of the physics engine treats each wheel of the car and the car body itself as separate rigid bodies. The wheels are attached to the car using dampened springs.

The following pseudo-code provides a high-level overview of how the physics are calculated each frame. The forces are calculated in a bottom-up fashion – first the wheel forces are calculated, and these are then used to calculate the car body forces.

```
For each frame
   Check all collision points for track intersection
   For each wheel
      Update position of suspension
      Apply effect of acceleration and turning
      Add wheel force to car body
   End For
   If collisions have occurred, calculate response
   Apply collision response forces to car body
   Apply wheel forces to car body
   Apply drag and gravity forces to car body
   Calculate new position of car body and wheels
End For
```

3.2.1 Rigid Body Dynamics

Using a rigid body system for the physics in this project meant that the car physics and collision detection and response could be integrated into one set of calculations. Unfortunately, space does not permit a mathematical explanation of how the rigid body system is implemented, but a high-level overview is presented here.

In a rigid body, there is no migration of mass – in other words, the relative positions of the particles composing the rigid body do not change. This fact simplifies the equations used. The motion of the rigid body is split into the linear motion of the centre of mass (CM), and the angular motion of the body about its CM. This is possible because all particles share the same relative angular motion about the CM. In other words, the motion of the rigid body can be described as a whole rather than for each particle individually. Thus, the rigid body can be described by its mass and moment of inertia.

The rigid body's position in space is defined by a position vector and a rotation matrix. The linear and angular momentum of the rigid body are also necessary to describe its state. The mass of the rigid body and inertia tensor (the scaling factor between angular moment and angular velocity) are constants, which are defined when the game is loaded.

Each frame, the rigid body is updated using the time that has elapsed since the last frame. Euler integration is used to calculate the change in angular velocity and momentum that occurs due to the forces exerted on the rigid body during that frame.

3.2.2 Wheel forces

Since the wheels are the means by which the power of the engine is transferred to the ground in order to move the car forwards, they are the logical place to begin a discussion of the physics engine. The forces on each wheel are calculated separately. Several websites [14,15,16,17] were used to gather the equations used below. Some of these equations have been optimised, since the important goal in a car game is for the physics to feel playable, not necessarily to be realistic.

3.2.2.1 Suspension forces

The model that is used to simulate suspension is shown in the following diagram. It is an approximation to the real-world suspension systems used in cars, but it is close enough to be realistic for the player. The mass of the car body is m_1, and the mass of the wheel is m_2. k is the spring coefficient, and c is the damping coefficient. In the simulation, all of these constants are set in the configuration file for the car. The positions of the car body and wheel are denoted by x_1 and x_2 respectively.

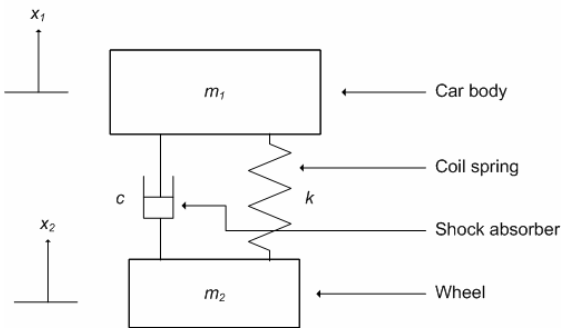


Figure 3 – Suspension system

In order to calculate the forces in effect, Newton's law is applied to the two masses (car and wheel). First, for the car body, the only two forces acting are due to the main suspension (*k*, *c*). The spring force on m_1 will depend on both x_1 and x_2. The force is proportional to the distance through which the spring has been stretched or compressed, which is $x_1 - x_2$. For example, if $x_1 = x_2$, then there is no spring force. Therefore, the spring force $F_s = -k(x_1 - x_2)$. Using the same principles, the damping force $F_d = -c(\dot{x}_1 - \dot{x}_2)$ where $\dot{x}$ is the velocity of x. For example, if m_1 and m_2 are moving upwards at the same velocity, then there is no damping force. Since $F = ma$, we can write the equation of motion for the car body as follows:

$$F = m_1\ddot{x}_1 = F_s + F_d = -c(\dot{x}_1 - \dot{x}_2) - k(x_1 - x_2) \qquad \text{(Eq. 1)}$$

The force that the suspension exerts upon the car body is calculated with this equation. This force is calculated for all four wheels, and then these forces are applied at the appropriate points on the car body. The rigid body dynamics system is then used to calculate the resulting effect.

3.2.2.2 Longitudinal forces

Longitudinal force has been approximated in the simulation by wheel force, brake force, rolling resistance, and drag.

In a rally game such as this, it is most common for cars to be four-wheel-drive. However, the simulation is flexible enough to cope with a variable number of drive wheels.

Each time the program runs through the game loop, the following steps occur:

Get the current angular velocity of one of the drive wheels

The angular velocity of each wheel is stored from the previous loop so this is a trivial step. When the car is stationary (for instance when the simulation starts), the angular velocity will obviously be zero.

Calculate the RPM (revolutions per minute) of the engine

The RPM of the engine is calculated using this formula:

$$RPM = \omega_w \cdot C_{gear} \cdot C_{diff} \cdot \frac{60}{2\pi} \quad \text{(Eq. 2)}$$

where ω_w = wheel angular velocity (rad/s), C_{gear} = gear ratio (depends on current gear), and C_{diff} = differential ratio.

Calculate the engine torque

The engine torque is calculated using the following formula:

$$T_{engine} = P_{acc} \cdot T_{\max} \cdot C_{gear} \cdot C_{diff} \cdot E_{trans} \quad \text{(Eq. 3)}$$

where P_{acc} = position of the accelerator (0..1), $T_{\max}$ = Maximum torque of the engine at the current RPM, C_{gear} = gear ratio (depends on current gear), C_{diff} = differential ratio, and E_{trans} = transmission efficiency.

Calculate engine torque distribution

This is done by dividing the engine torque by the number of drive wheels. This assumes that engine power is divided equally, which isn't usually the case in real life. However, for the purposes of a simulation it is a suitable assumption.

Calculate the slip ratio for each drive wheel

When a car is moving, the non-drive wheels are simply rolling. However, the drive wheels are actually rotating slightly faster than this. This introduces a friction force between the tyre and the road – and it is this friction force that pushes the car forwards.

Slip ratio is traditionally given by this formula:

$$\sigma = \frac{\omega_w \cdot R_w - v_{long}}{\left| v_{long} \right|} \quad \text{(Eq. 4)}$$

where ω_w = wheel angular velocity (rad/s), R_w = wheel radius, and v_{long} = longitudinal velocity of car (m/s).

However, when the longitudinal velocity is very small then this results in instability in the simulation (dividing by a number very close to zero). Therefore, the slip ratio for drive wheels is approximated with this formula:

$$\sigma = \omega_w \cdot R_w - v_{long} \quad \text{(Eq. 5)}$$

Calculate the traction force for each drive wheel

The traction force is the friction force that the road surface applies to the wheel surface. It is this friction force that causes acceleration.

For each drive wheel, the simulation calculates the traction force of that wheel. This traction force is then used to calculate the acceleration as described later.

The traction force is calculated using this formula:

$$F_{long} = F_{norm,long} \cdot W \quad \text{(Eq. 6)}$$

where F_{long} = longitudinal force (traction force), $F_{norm,long}$ = normalized longitudinal force for a given slip ratio, and W = weight over the tyre.

The normalized longitudinal force is derived from the slip ratio according to a curve that is similar to that shown in figure 4.

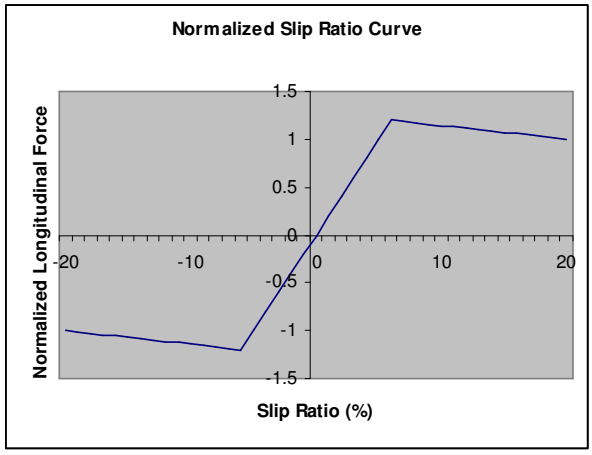


Figure 4 – Slip ratio calculation

Calculate the traction torque for each drive wheel

The traction torque is simply the traction force multiplied by the wheel's radius, as shown by the following formula.

$$T_{traction} = -F_{long} \cdot R_w \qquad \text{(Eq. 7)}$$

where F_{long} = Longitudinal force (traction force) and R_w = wheel radius.

Calculate the total torque for each drive wheel

The total torque is the sum of the engine torque, traction torque, and brake torque, as shown in figure 5 and equation 8.

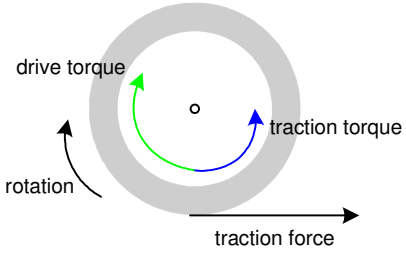


Figure 5 – Components of torque for a wheel

$$T_{wheel} = T_{engine} + T_{brake} + T_{traction} \qquad \text{(Eq. 8)}$$

Calculate the angular acceleration and angular velocity for each wheel

The angular acceleration is calculated by dividing the total torque for the wheel by the inertia of the wheel. The angular velocity is integrated (using Euler integration) from the angular acceleration.

3.2.2.3 Lateral forces

To calculate the lateral forces acting on the car, the following steps are performed each time through the game loop.

Calculate the car sideslip angle

The sideslip angle is the angle between the car's orientation and the car's velocity vector. In other words, if the car is pointing in one direction but moving in another direction, there will be a sideslip angle.

$$\beta = \tan^{-1}(\frac{v_{lat}}{v_{long}}) \quad \text{(Eq. 9)}$$

where v_{lat} = lateral velocity of car (m/s), and v_{long} = longitudinal velocity of car (m/s).

Calculate slip angle for each wheel

The slip angle is composed of the sideslip angle of the car, the speed of the angular rotation of the car around the up axis, and the steering angle (only applicable to the front wheels). The slip angles for each set of wheels are calculated using these formulae:

$$\alpha_{front} = \tan^{-1}(\frac{v_{lat} + \omega \cdot b}{|v_{long}|}) - \partial \cdot \text{sgn}(v_{long}) \quad \text{(Eq. 10)}$$

$$a_{rear} = \tan^{-1}(\frac{v_{lat} - \omega \cdot c}{|v_{long}|}) \quad \text{(Eq. 11)}$$

where v_{lat} = lateral velocity of car (m/s), v_{long} = longitudinal velocity of car (m/s), ω = angular velocity of car around the up axis, b = distance from centre of gravity to front axis, c = distance from centre of gravity to rear axis, and ∂ = steering angle.

Calculate lateral force for each wheel

The lateral force is calculated using the same principle as the longitudinal force, with the difference being that lateral force is derived from slip angle and longitudinal force is derived from slip ratio. The curves for both look very similar, as the figure below shows.

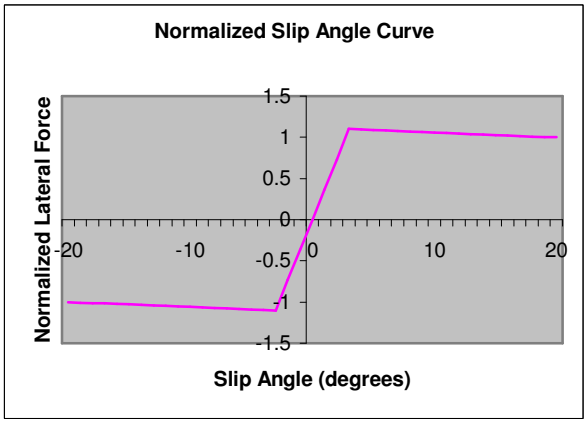


Figure 6 – Slip angle calculation

3.2.3 Car forces

After calculating the forces for each wheel, calculating the forces for the car body is a simple matter of adding the wheel forces to the rigid body. There are two additional forces that act on the car body as a whole – gravity and drag. Drag is linearly proportional to the speed, and it is the drag force that stops the car accelerating indefinitely and imposes a limit on the maximum speed.

This diagram shows where the wheel and car forces act. Only one wheel is shown. The angled arrow for gravity represents a vector pointing straight down. The angled arrow for suspension represents a vector pointing straight up, but in the wheel's coordinate system.

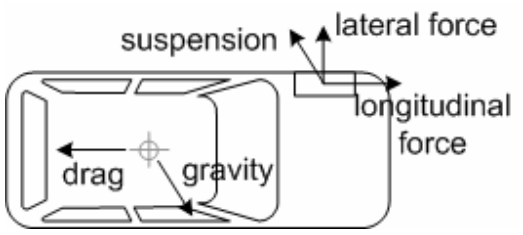


Figure 7 – Summary of car forces

3.2.4 Collision detection

Collision detection is needed for more than simply testing when the car hits a solid object. It is also used to determine the height of the terrain under each wheel. Thus, there are two separate instances where slightly differing variants of collision detection are used:

- Per-wheel collision detection. This works by extending a ray from the centre of the wheel downwards, where the down direction is based on the current orientation of the wheel. In order to perform this type of collision detection, ray-triangle intersection is used.

- Car body collision detection. For each car, several collision points are defined. The quantity of these is dependent on the shape of the car. For example, on the Subaru Impreza, there are 12 collision points. These are located at the corners of the top and bottom of the car body, and at the corners of the roof. Because the edges between these points are generally linear, they provide a good approximation to the shape of the

car. All of these points are tested to see if they have collided with the track. In order to perform this type of collision detection, point-triangle collision detection is used.

Thus, for the Subaru Impreza, collision detection must be performed 16 times (4 for the wheels and 12 for the car body). Since there are upwards of 6000 triangles that make up the track mesh, it would be computationally expensive to do this in an unoptimised fashion. Therefore, before the collision detection routines are performed, the set of triangles that are within a fixed radius of the car is first calculated, and then only these triangles are used for the collision detection tests.

3.2.4.1 Ray-triangle intersection

Ray-triangle intersection is used to find the height underneath each wheel [18]. This is used for the suspension system. A ray is drawn from the centre of the wheel downwards, and this ray is tested against the triangles in the track mesh, based on the cached lookup described in the previous paragraph. The basic idea is to find whether the ray intersects the triangle, and if so, to find the point of intersection. The algorithm used involves these steps:

1. Find the point of intersection of the ray and the plane of the triangle. If they do not intersect, then there can be no collision.

2. Test to see whether the point of intersection lies within the triangle.

The input for the collision detection test is a start point (which will be called position in this discussion) in 3D space (the position of the centre of the wheel) and a direction. This is defined mathematically as

$$line = position + (t \times direction) \qquad \text{(Eq. 12)}$$

where t and direction are combined using the vector cross product.

By substituting a value for t, any point along the line can be obtained. The value of t that needs to be obtained is the point where the line intersects the plane of the triangle. Using the plane equation, this can be calculated by

$$t = \frac{normal \cdot dir}{normal \cdot position} \qquad \text{(Eq. 13)}$$

where the dot operator represents the vector dot product, and dir = the vector from position to the first triangle vertex.

The point of intersection between the ray and the plane of the triangle is now known. This point is then tested to see if it lies within the triangle. This is done using an algorithm first proposed by Badouel [19]. This algorithm involves creating a plane for each edge of the triangle, with the normals pointing towards the inside of the triangle. If all the points lie in the positive half-space of all three planes, then the point must lie inside the triangle.

This algorithm uses barycentric coordinates. Any points in the plane V1V2V3 can be represented by the barycentric coordinate (U, V). The parameter U controls how much V2 gets weighted into the result and the parameter V controls how much V3 gets weighted into the result. Lastly, 1-U-V controls how much V1 gets weighted into the result.

The algorithm calculates U and V. If U, V >= 0 and U + V <= 1, then the point lies inside the triangle, and a valid intersection has been found.

Where possible, all the calculations of triangle normals are performed when the game is first loaded, to avoid unnecessary calculations on the fly.

3.2.4.2 Point-triangle intersection

Point-triangle intersection, as used for collision points on the car body, works in a very similar way to ray-triangle intersection, described in the section above. However, instead of testing to see whether a ray intersects a triangle, it is necessary to find whether a single point in 3D space lies behind a triangle. The definition of "behind" comes from the direction of triangle normal – "behind the triangle" means that the point lies in the half-plane denoted by the negative normal direction.

3.2.5 Collision response

After a collision has been detected for a point on the car body, the response to this collision needs to be calculated. In some cases, there might be no response – this would happen if the object that the car has collided with is considered "soft", for instance a small bush. These soft objects are specified in the track configuration file.

Assuming that a hard collision has been detected, the next step is to determine the velocity of the point on the car that collided. From this, the collision force is calculated. There is also a friction force that operates in the plane of the triangle. These forces are combined to create a collision response force that is applied to the rigid body at the point of collision.

The pseudocode for the complete collision detection system for the car body is listed here.

```
For all collision points on car body
  If point has collided with track (point-triangle intersection)
    Calculate collision force in triangle normal direction
    Calculate friction force in plane of triangle
    Add collision and friction forces
    Add total force to rigid body at point of collision
  End If
End For
```

3.3 Input Engine

In order to keep the interface as simple as possible for players, only keyboard input is used. The mouse is not often used in a driving game since it is far easier to control a car by using, for instance, arrow keys on the keyboard to control the car's acceleration and turning. After it was decided to only use the keyboard for player input, it was necessary to find a high performance method of accessing the keyboard data.

Each programming language tends to have its own means of retrieving this data – in C#, keyboard input is handled through a set of events that are fired when the user presses and releases a key, whereas in C++ the program polls for messages that indicates that a key has been pressed or released. While the method in C++ of getting keyboard data is sufficiently performant, it was found in an early prototype version of the game that the C# method is not

always fast enough for a high-performance racing game, where it is important that all keyboard input is captured quickly.

In order to provide both a high-performance means of accessing keyboard data, and a common interface across both versions of the game, DirectInput has been used. As its name suggests, DirectInput is the component of DirectX that deals with input devices. DirectInput is a very powerful API and allows access to keyboards, mice, joysticks, and steering wheels. However, only the keyboard will be used in this simulation.

Each time the input engine is updated, it polls the keyboard to retrieve the set of all keys pressed. This set is then compared against the set of keys that were pressed in the last frame, and from this, three sets can be calculated – the set of keys pressed since the last frame, the set of keys released since the last frame, and the set of keys pressed in both the last frame and this frame.

These sets are used in different ways by different sections of the program. For example, the GUI only uses the keys pressed and released since the last frame. If the player presses the down arrow while navigating a list in the menu, they should only go down one item in the list. However, if the game simply tests to see if the down arrow is pressed, and moves down one item, and since the game is updating upwards of 50 times a second, it likely that the player will move down many more items than they were intending to. The simplest solution to this problem is to respond only when the key is first pressed.

3.4 Graphics Engine

The graphics engine is responsible for actually drawing everything that the player sees. This includes the track, the car, the dashboard, and the clock. Each of these is described in more detail in the following sections.

3.4.1 Track

Each track consists of a mesh of triangles, and textures to place on to these triangles. Fundamentally, rendering the track is a simple process of sending all the vertices and textures to the graphics card to render. However, because of the large number of triangles in any track

(8000 for the Four Oaks track), some method is needed to draw only the triangles that are actually visible to the player. This process is known as culling. Two different methods have been used to provide effective culling: a clipping distance and frustum culling. Both these methods rely on splitting the track into separate sections. This is because it is more efficient to perform culling on large batches of triangles rather than individual triangles.

When the track is first loaded, a "bounding sphere" is created for each section. The radius of this sphere is chosen such that the sphere encompasses the farthest extents of the section it is created for. The bounding sphere can then be used to test efficiently for visibility and clipping.

Using a clipping distance is a trivial way of culling unnecessary sections of track. Any sections further away from the player than a specified distance (this distance is set for each track) are not drawn. This test is done by comparing the distance from the player to the centre of the bounding sphere of each section of track, with the clipping distance.

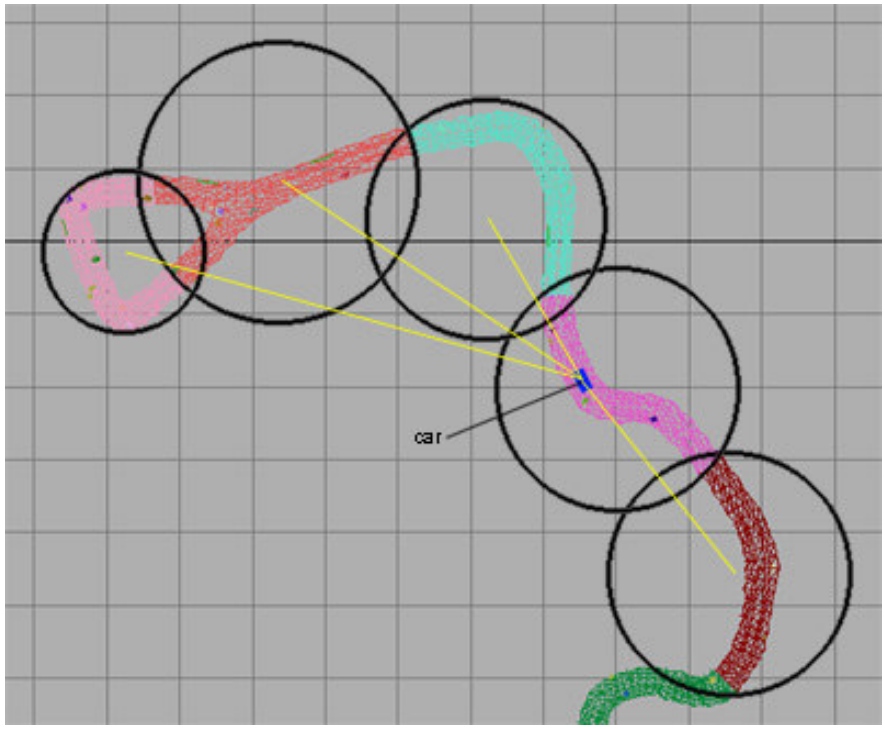


Figure 8 – Distance-based culling

The figure above shows a top-down view of the track. The yellow lines represent the distance from the car to the centre of each sphere. When this distance is less than the clipping distance –

300m for the Eville track, shown in the figure – that section of track will be rendered. The clipping distance is chosen such that from any part of the track, it will not be too noticeable when objects beyond the clipping distance are culled.

Frustum culling is a slightly more complex procedure. The "frustum" is the region of 3D space that is currently visible to the player. Any objects outside this frustum don't need to be drawn, and indeed drawing them would be inefficient. Frustum culling provides a way of testing to see if it is possible for objects to be seen by the player, and if it is not, culling them.

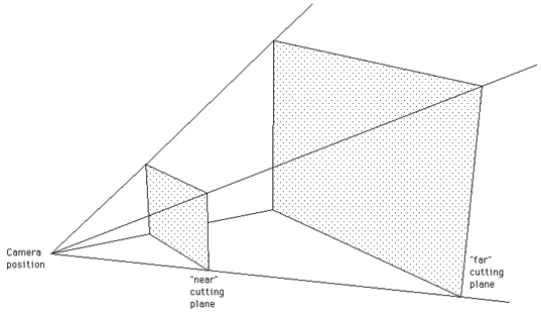


Figure 9 – Shows the view frustum which is used to test for visibility of objects

In order to check the visibility of objects, the six planes of the frustum are extracted from the world, view and projection matrices [20]. These three matrices are used by Direct3D to transform points on to the screen. After extracting the six planes, each object is tested against them to see which half-plane it lies in. If it lies in the positive half-plane of all six planes, it must be inside the frustum. Otherwise, the object is not visible and can be culled.

3.4.2 Car

Along with rendering the basic car mesh, two methods are used that are designed to add realism to the scene. However, before discussing these steps, it is necessary to point out that the player can change the view. There are three different views available to the player – behind car (near), behind car (far), and front bonnet. In the front bonnet viewing position, the car itself cannot be seen and is therefore not rendered. The steps described below thus only apply to the views where the camera is behind the car.

The two methods used to enhance the car rendering are drawing the car's shadow, and using environment mapping. These methods are described in more detail in the following paragraphs.

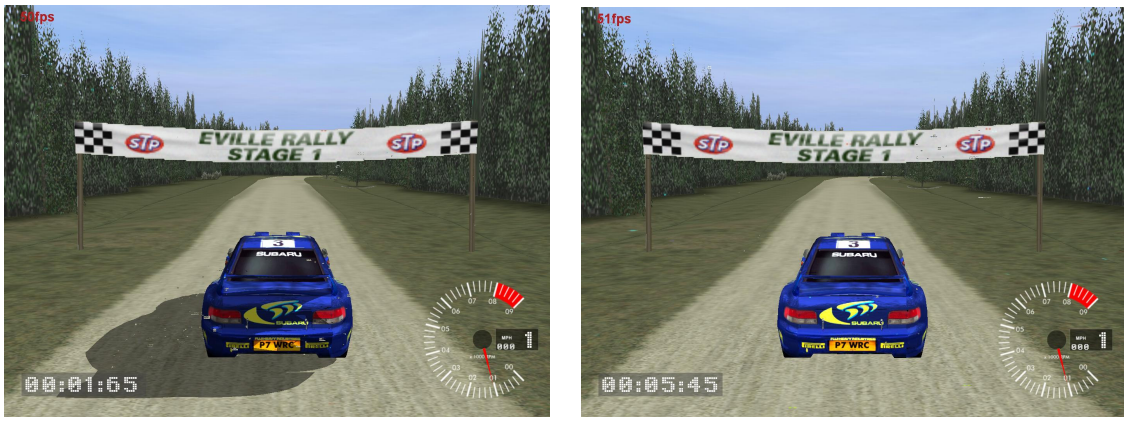


Figure 10 – Shows the difference between a shadowed car (left) and a non-shadowed car (right)

It is clear from the pictures above that shadows are very important in helping place objects in a 3D scene. In the picture on the right, it is not immediately obvious where in 3D space the car actually is. However, the addition of a shadow helps the eye place it.

Various techniques have been used in car games for drawing the car's shadow – including drawing a grey quad underneath the car, shadow mapping, and stencil shadowing. Currently stencil shadowing offers the highest quality shadow, and is the technique implemented here. Since stencil shadowing has been discussed many times before, including most notably Hun Yen Kwoon's exhaustive article [21], the details of the technique will not be discussed here.

The other technique that is used to add realism is environment mapping. In the real world, reflective objects (such as cars) reflect the environment around them, and this gives them an appearance of shininess. In a 3D game, this shininess must be simulated. The normal process for simulating reflection of an object's environment is to create an environment map – a cube texture with 6 faces that represents the environment around the object – and then, for each pixel on the object, find the pixel in the environment map that is the result of reflecting a vector from the eye off the object. This technique is shown graphically on the next page.

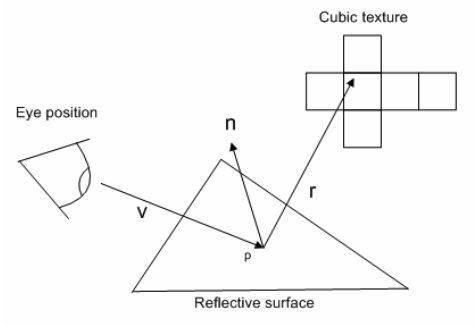


Figure 11 – Shows how the environment map usage is calculated based on the eye position and the surface normal

As the object (in this case, the car) moves in relation to the observer, the part of the cubic texture that is reflected also moves. The result is that the car appears to be reflecting its environment. The pictures below show a comparison of rendering the car with no environment mapping, and rendering the car with environment mapping. It is, of course, much easier to see the effect of environment mapping within the actual game than from a static screenshot.

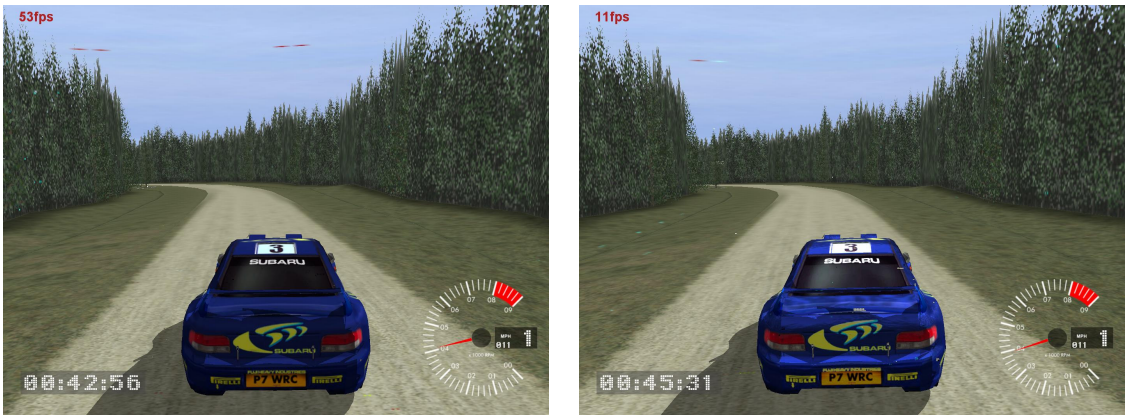


Figure 12 – Shows the difference between a plain car (left) and an environment-mapped car (right)

3.4.3 Dashboard

The dashboard consists of a set of 2D graphics drawn on top of the display. It shows important information to the player, including speed, current gear, current RPM, and race time. The engine information is obtained directly from the physics engine, and the time is obtained from the game timer that is also responsible for calculating the frames per second (FPS) that the game is running at.

3.5 Sound Engine

A large part of the believability of a game comes from sounds. In the case of a racing game, there are three different types of sounds that can be played:

- Engine sounds
- Road sounds
- Impact sounds

3.5.1 Engine sounds

Since we obviously cannot create sounds to match every possible engine RPM, we need a way to create sounds dynamically of the correct frequency. Fortunately, DirectSound allows us to change the frequency dynamically as the sound is played, meaning that the engine sound only needs to be stored at a single frequency. The engine sound can be simulated fairly accurately by changing the pitch of the sound as the player accelerates and slows down the engine. Since the minimum and maximum RPM is fixed for a particular engine, these values can be used to calculate a ratio that is then used to calculate the engine sound frequency. This formula shows the actual calculation:

$$Frequency = K_1 + RPM \times K_2 \qquad \text{(Eq. 14)}$$

where K_1 and K_2 are constants for a particular engine.

3.5.2 Road sounds

Road sounds are slightly more involved than engine sounds, since the sound of the road under the car depends on the type of surface the car is currently travelling over. To be more precise, it is necessary to find the type of surface each wheel is over, and then play the set of sounds that correspond to these surface types.

As an example, imagine a car moving from a gravel surface to a mud surface. The front two wheels of the car are on the mud surface, while the rear two wheels are still on the gravel surface. Sounds for both gravel and mud need to be played. The type of road surface that each wheel is over can be obtained by firing a ray downwards from the wheel, based on its current orientation, and testing to see which triangle in the track mesh the ray intersects. When the mesh is loaded, the type of surface represented by each triangle is stored.

In order to further enhance these sounds, the speed of the car is also taken into account. When the car is travelling faster, the road sound is louder, and vice versa when the car is travelling slower. When the car is sliding (in other words, when the car's lateral velocity is larger than a certain threshold) then a "sliding" sound is played.

3.5.3 Impact Sounds

The last type of sound is the sound that results from the car hitting an object. Each car has several collision points defined – for instance, at each corner of the car and the corners of the roof – and these points are tested to see if they have collided with any section of the track. If they have, then the type of surface they have collided with is tested for, the same way as for road sounds, described in section 3.5.2. The types of surface that the car can collide with depend on the track itself, but usually will include trees, crash barriers, and rocks. Each of these types of objects has a different sound associated with it. The volume of the sound played depends on the speed of the car at the time of impact.

3.6 GUI

Although the main part of the game consists of the player driving a car along a track, the game also needs a means by which the player can choose the car and track before racing. It is also helpful to let the player pause the game. This functionality is provided by the GUI. The GUI has been modelled on the Windows Forms architecture, which is used by the .Net platform to create Windows applications. A set of reusable components were created, such as Labels and Images, that allowed the easy creation of menus. All components are placed onto Forms, and they are all based on a Control base class (using object-oriented inheritance). The GUI is configured by XML files, and is generated dynamically when the game loads. The following

code snippet demonstrates the XML that is needed to show a label with white text on a form which has a blue background. If necessary, any property such as Text or ForeColour can be modified at run-time by other parts of the code.

```
<Form Bounds="20,20,300,300"
      BackColour="Blue">
    <Label Bounds="0,0,280,280"
           FontSize="30.0"
           ForeColour="White"
           Text="Hello World" />
</Form>
```

Chapter 4 – Implementation

This section describes the two implementations of the design. For the most part, the code was a logical progression of the design as described in the previous section. Therefore, only a brief overview is given here. The C# version was written first, and so is discussed first, followed by the C++ version.

Metric	C#	C++
Lines of code	6168	6319
Number of classes	42	54
Number of methods	263	264
Average memory usage	80Mb	40Mb

Table 1 – Several metrics comparing the C# version to the C++ version

4.1 C# version

The unmanaged DirectX APIs consist of a set of interfaces. These interfaces have methods, but the design of each API is not object-oriented. When Managed DirectX was written, Microsoft took the opportunity to provide a properly object-oriented API. This meant that the project could be designed in an elegant fashion that suited the .Net architecture.

Although .Net allows programs to be written in several languages, including Managed C++, C#, Visual Basic and J#, it was decided to use C#, since this is the language of choice for those wishing to write high-performance .Net applications. C# allows "unsafe code" (code which uses pointers) to be written, which was useful for the high-performance timing code.

The overall goal for the C# code was to combine performance with elegance. In general, the C# language allows both these goals to be realised.

For example, the input engine described in section 3.3 was implemented in C# using events and delegates. Delegates are C#'s equivalent of function pointers in C++. When the input engine updates and finds that a key has been pressed, released or held since the last frame, an event of the appropriate type (KeyDown, KeyUp, or KeyPress respectively) is fired. Without needing to write any more code, all the methods that have subscribed to that event are called.

4.2 C++ version

The design of the C++ implementation closely followed the C# implementation. However, where greater efficiency could be obtained by restructuring the C++ code (while still retaining the same basic techniques as the C# code), this was done, in order to ensure that the eventual benchmarking would be fair. For the most part this involved manipulation of pointers, a technique that is possible, but not recommended, in C#.

One of the primary differences between the two versions was in the event handling for the input engine. Events and delegates do not exist in C++ in the same way as they do in C#, and so this functionality had to be coded. After a few different techniques were tried, functors [22] were used. Functors (short for function pointers) allow pointers to member functions of classes to be passed between parts of a program. For this project they were used to maintain a list of all the classes that wish to be notified when a keyboard event occurs.

4.3 Analysis of Structural Similarities and Differences

In order to show the type of change that was required when converting the C# code to C++ code, the following code snippets show the same method call in the two languages.

```
// C# code
FontDescription tDesc = new FontDescription();
tDesc.FaceName = "Arial";
if (bBold) tDesc.Weight = FontWeight.Bold;
tDesc.Height = 12;
tDesc.Quality = FontQuality.ClearTypeNatural;
Font pFont = new Font(pDevice, tDesc);
```

```
// C++ code
FontDescription tDesc;
ZeroMemory(&tDesc, sizeof(tNewDesc));
strcpy(tDesc.FaceName, "Arial");
if (bBold) tDesc.Weight = FW_BOLD;
tNewDesc.Height = 12;
tNewDesc.Quality = 6;
ID3DXFont pFont;
D3DXCreateFontIndirect(pDevice, &tDesc, &pFont);
```

The C# code is 6 lines long, while the C++ code is 8 lines long. The C# code is clearer, although this is a subjective judgement.

Error handling is much more structured in the managed version of the DirectX API. Unmanaged DirectX API method calls return a value that indicates success or failure. If this return value is not checked, then it is possible to continue executing the program without realising that an error has occurred. However, Managed DirectX throws exceptions when errors occur, and these exceptions must be explicitly caught, or else the program will terminate. The following code snippet demonstrates this:

```
// C# code
try {
   // test the cooperative level to see if it's okay to render
   m_pDevice.TestCooperativeLevel();
}
catch (DeviceLostException) {
   // if the device was lost, do not render until we get it back
   return;
}
catch (DeviceNotResetException) {
   // reset the device
   m_pDevice.Reset(m_pDevice.PresentationParameters);
}
```

```
// C++ code
// test the cooperative level to see if it's okay to render
if (FAILED(hr = m_pDevice->TestCooperativeLevel())) {
   // if the device was lost, do not render until we get it back
   if (hr == D3DERR_DEVICELOST)
      return;

   // check if the device needs to be reset
   if (hr == D3DERR_DEVICENOTRESET) {
    if (FAILED(m_pDevice->Reset(&m_tPresentParams)))
       return;
   }
}
```

DirectX method calls in C# and C++ are fairly different. For example, the Matrix class in Managed DirectX is implemented as a proper object-oriented class with member variables and methods. In unmanaged DirectX, the Matrix class contains the data, and all the matrix methods are global methods that take a Matrix object as a parameter. The following code snippet shows this in practice.

```
// C# code
Matrix tMatrix = Matrix.Identity;
tMatrix = Matrix.RotationY(Math.PI);
Vector3 tWheelPosition = pWheel.Attach;
tMatrix.Multiply(Matrix.Translation(tWheelPosition));


// C++ code
Matrix tMatrix;
D3DXMatrixIdentity(&tMatrix);
D3DXMatrixRotationY(&tMatrix, PI);
Vector3 tWheelPosition = pWheel->Attach;
D3DXMatrixMultiply(&tMatrix, &tMatrix,
   D3DXMatrixTranslation(&g_tTempMatrix, tWheelPosition.x,
   tWheelPosition.y, tWheelPosition.z));
```

4.4 Replays

In order to provide a short demo that plays back when the player first loads the game, a replay feature was implemented. This allows games to be recorded to a binary format, and then played back by the simulator. All the data necessary to recreate each frame is recorded, including the position and orientation of each of the wheels and the car body.

4.5 Tools

Microsoft Visual Studio .Net 2003 was used during the implementation of both versions. It was very useful because it provided a standard interface, regardless of whether the programming was being done in C# or C++. The debugging features in Visual Studio are very powerful, and were taken advantage of many times during the course of the implementation.

Chapter 5 – Program Benchmarking and Testing

This section describes the benchmarking and testing that took place after the two versions of the game had been completed. The aim of this project was to compare the performance of the two programming languages (C++ and C#); therefore it was very important that the benchmarking was accurate.

In order to verify that the game was of sufficient quality for the benchmarks to be meaningful, testing was also conducted. The game was given to a group of testers, whose opinions are recorded in the section of testing. The questionnaire used to gather these data is shown in Appendix 3.

5.1 FPS Benchmarking

A method that is often used to test the performance of new graphics cards is to compare the FPS (frames per second – the number of times that the screen is updated per second) of a game between older cards and the new card being tested. This technique was applied to this project, where the FPS were compared between the C++ and C# versions at several resolutions and on two different computers. In order to perform this test, code was written that recorded the number of frames rendered per second for a duration of 60 seconds. The average of these 60 seconds was calculated. The test was then repeated for a total of 5 times. Finally, the average of these 5 tests was calculated, and it is this overall average that is shown below. The summarised results of these tests are shown here, with the full results available on the CD that comprises Appendix 4.

Two different tests were performed:

- First, the replay was timed. The replay is a demo that is played when the game is first loaded. The demo file includes all necessary positional data, and therefore no physics are calculated during replay. However, camera positions are chosen dynamically by the game, and it is possible that more of the track will be visible. For this reason, it is possible that the replays run at a lower FPS than the game itself.

- Second, the main game was tested. This includes all parts of the input, physics and graphics engine.

In order to test whether the screen resolution made an impact on timings, the game was tested at two different resolutions on the laptop – 1400x1500 and 1024x768.

The following table shows how all these tests tie together.

Desktop PC				Laptop							
In-Game		Replay		In-Game				Replay			
1024x768				1400x1050		1024x768		1400x1050		1024x768	
C#	C++	C#	C++	C#	C++	C#	C++	C#	C++	C#	C++

Table 2 – The bottom row represents the actual tests performed

The specifications of the test computers are detailed in this table.

	Desktop PC	Laptop
Processor	AMD Athlon 2000+	Intel Pentium P4 3.0GHz
Memory	512Mb 266MHz DDR RAM	1Gb 400MHz DDR RAM
Graphics Card	ATI Radeon 9600Pro	ATI Mobility Radeon M10

Table 3 – Configuration of the test computers

The following charts show the results gathered from the FPS test.

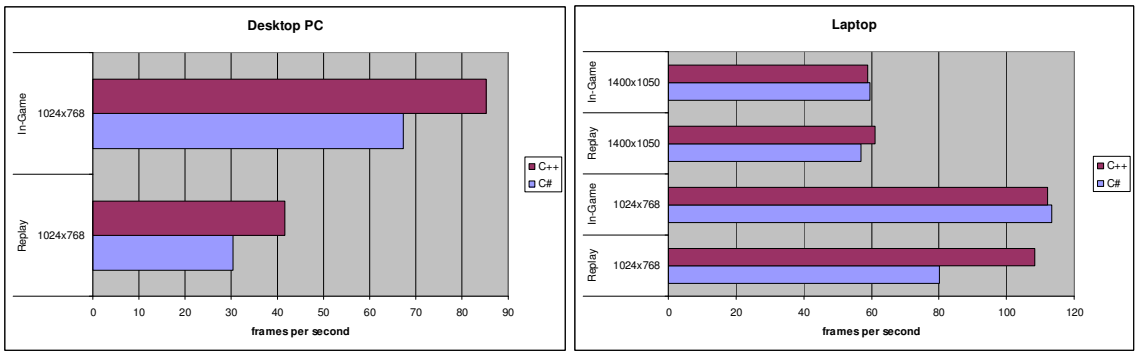


Figure 13 – Results from FPS benchmarking – note the similar results for C# and C++ on the laptop

5.2 Code Profiling

Although the FPS comparisons described in the previous section do provide a high-level view of the speed of the two versions of the game, they do not give much detail. For instance, although it can be seen from the graphs on the previous page that the speed of the two versions converges in higher resolutions, it is not immediately possible to determine the cause. For this reason, another method of benchmarking was used – code profiling. A code profiler is a piece of software that runs alongside the code being tested. It records the length of time spent by the processor inside each method and, if required, the time spent processing each individual line of code. This is often used when optimising code to determine the points where optimisation would be beneficial. For this project, code profiling has been used to compare the length of time spent processing each section of code for the two different versions. It was possible to determine how much of the processor time was spent within DirectX methods and how much was spent within the game code.

A code profiler returns a list of all the methods that have been called during program execution, along with the time spent inside the method and the number of times that the method has been called. Obviously, the timing is altered by the code profiler itself, but since the same profiler is used for both versions, they should both be altered by the same amount.

As described in section 2.5, the code profiler chosen for this project was DevPartner Profiler Community Edition. Since the game execution produces a list of many thousand methods, only the results of most interest are shown below. The timings are in microseconds and are the average amount of time spent in each method.

		C# (µs)	C++ (µs)
Torq.Graphics.ShadowVolume.AddEdge This method is used to construct the silhouette used in shadow volume extrusion	**1024x768**	120.1	120.1
	1400x1050	120.6	121.6
Microsoft.Direct3D.Device.Present This method presents the contents of the buffer to the graphics card	**1024x768**	168.8	212.1
	1400x1050	879.0	864.8

Table 4 – Method timings for two resolutions – 1024x768 and 1400x1050

Unfortunately space does not permit a more detailed breakdown of method calls. However, studies of the results produced by the code profiler indicate that the two methods above are indicative of other methods. The graph below shows a visual representation of the data in Table 4.

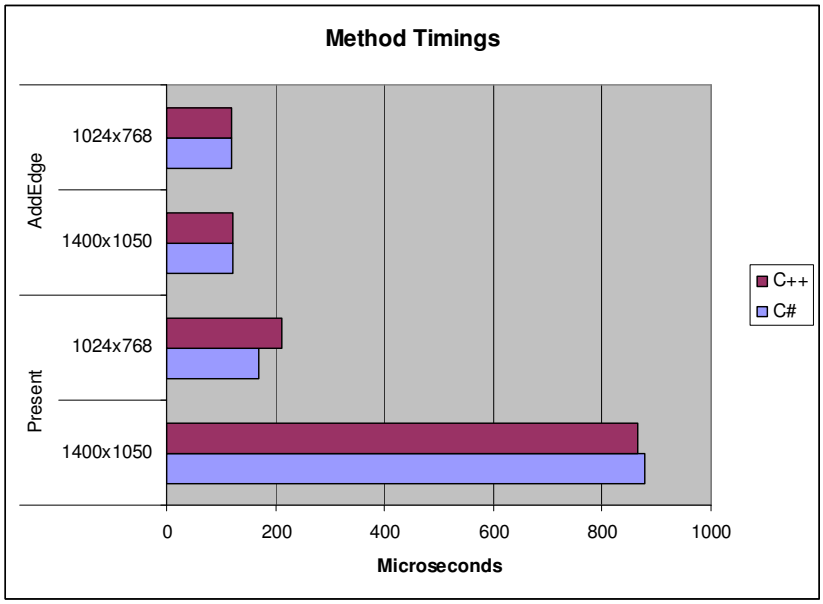


Figure 14 – Graph of the data in Table 4

The first method shown in Table 4, `AddEdge`, is the slowest CPU-based method in the game. CPU-based, in this context, means that it runs entirely on the CPU without any reference to the graphics card. As can be seen from the graph above, the time it takes to execute varies very little in different resolutions.

The second method in Table 4, `Present`, is the slowest of the non-CPU-exclusive methods. These methods execute on the CPU, but cause the operating system to send or retrieve data from the graphics card. As can be seen, the speed of this method is heavily resolution-dependent.

It can be concluded from these results (combined with the fact that these two methods are indicative of other CPU-based and non-CPU-exclusive methods) that the speed of non-CPU-exclusive methods varies with different resolutions much more than the speed of CPU-based

methods. The code profiler data showed that as the resolution increases, the execution time for CPU-based methods becomes a small percentage (around 5%) of the total program time.

Given that C# and C++ are both calling the same underlying DirectX methods, it follows that the speed of these methods would be the same. The differences in speed between C# and C++ occur in the CPU-based methods, and since the execution time for these becomes a small percentage of total time at higher resolutions, the difference between the two versions tends to zero, as is evident from Figure 13.

Therefore, whenever the program spends the majority of its execution time inside non-CPU-exclusive methods, it can be seen that C# and C++ will have very similar performance. As average screen resolutions increase, it is more likely that this will be the case.

5.3 Testing

A questionnaire (see Appendix 3) was given to eight external testers in order to acquire feedback about the game. This was necessary to verify that the game was playable, and therefore that the benchmarks were meaningful. The following charts show a summary of the feedback received for questions 1 and 2 on the questionnaire.

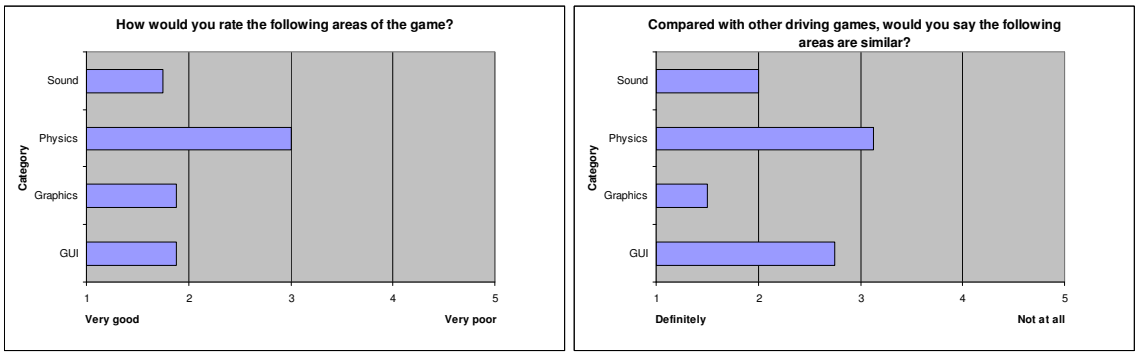


Figure 15 – Summary of questionnaire feedback

When asked whether they would want to play the game again (question 3 on the questionnaire), the average answer from the eight testers was 2.5. A score of 1 represented "definitely" and a score of 5 represented "not at all", and so 2.5 represented a positive score.

Chapter 6 – Conclusion

6.1 Summary

The purpose of this project was to create a 3D car racing game in both C# and C++, such that the performance of the two versions could be compared. Several DirectX APIs were used to implement the graphics, sounds, and input.

In order to create a car racing game, a physics engine was written that simulates the movement of a vehicle across a rough terrain, including realistic collision response. This physics engine was based on mathematical principles and has been shown in testing to be a believable simulation.

To add further to the feeling of realism that is necessary in a successful game, sound was implemented in the form of car engine, road surface, and impact sounds. Along with these sounds, music tracks are played that vary depending on where the player is within the game.

After the game had been finished, it was tested and benchmarked. The testing was done to ensure that the game was of sufficient quality for the benchmarks to be meaningful. The benchmarks were the ultimate aim of the project, and they produced some very interesting results.

According to the benchmarks, the C++ version of the game performed up to 30% faster than the C# version at a medium resolution (1024x768). However, as the resolution was increased (1400x1050), this difference of performance lessened to the extent that in one test, the C# version was even slightly faster than the C++ version.

This is an interesting result because screen resolutions in games are gradually, but continuously, increasing, thanks to the combined effect of ever more powerful graphics cards and better quality monitors [23]. Within a short space of time, therefore, it is envisaged that screen resolutions will be at a point where the difference in performance between games written in C++ and C# is negligible. At this time a commercial game written in C# and Managed DirectX will become a very real possibility.

6.2 Directions for Future Work

Although the project can be considered a success since it met the main objective of comparing the performance of the same game written in C# and C++, there is further work that could be carried out. For example, since the .Net platform allows code to be written in many languages, any code that is identified as a performance bottleneck in C# could be written in Managed C++. This may have the result of bringing the performance of C# closer to that of native C++ – further benchmarks would be necessary to verify this.

Other work could involve testing the two versions of the game on many different computers, with many different configurations of processors, memory and graphics cards. This could allow a more complete picture to be built up of when and why C# and C++ performance differs.

References

[1] J. Moran, *Game Development Costs Spiraling Upward*, 2004, Viewed 4 May 2004 on http://www.gamemarketwatch.com/news/item.asp?nid=2887.

[2] D. Jones, "Interview with Peter Moore, Corporate Vice President of Microsoft's Home Entertainment Division," *GamesTM*, Issue 18, p20, 2004.

[3] D. S. Platt, *Introducing Microsoft.NET*, Microsoft Press, 2001.

[4] T. Miller, B. Benincasa, *Managed DirectX*, Oct. 2003, Viewed 4 May 2004 on http://msdn.microsoft.com/theshow/Episode037/Transcript.html.

[5] Vertigo Software, Inc., *Quake II .NET*, July 2003, Viewed 4 May 2004 on http://www.vertigosoftware.com/Quake2.htm.

[6] IEEE Software, *C# and the .NET Framework: Ready for Real Time?*, 2003, Viewed 4 May 2004 on http://www.computer.org/software/homepage/2003/s1lap_1.htm.

[7] R. van Gaal, *Racer*, 2004, Viewed 4 May 2004 on http://www.racer.nl/.

[8] E. Espié, *TORCS*, 2004, Viewed 4 May 2004 on http://torcs.sourceforge.net/.

[9] R. van Gaal, *Pacejka's Magic Formula*, May 2003, Viewed 4 May 2004 on http://www.racer.nl/reference/pacejka.htm.

[10] C. Hecker, "Physics, Part 1: The Final Frontier" *Game Developer Magazine*, Oct. / Nov. 1996.

[11] Microsoft Corporation, *DirectX 9.0 SDK Update (Summer 2003) C++*, 2003, Viewed 4 May 2004 on http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/directx9cpp.asp.

[12] R. Dunlop, *Writing the Game Loop*, Jan. 2002, Viewed 4 May 2004 on http://www.mvps.org/directx/articles/writing_the_game_loop.htm.

[13] D. Krahenbuhl, *30 Frames per Second vs. 60 Frames per Second*, Viewed 4 May 2004 on http://www.daniele.ch/school/30vs60/30vs60_2.html.

[14] M. Marco, *Car Physics for Games*, Nov. 2003, Viewed 4 May 2004 on http://home.planet.nl/~monstrous/tutcar.html.

[15] K. Murphy, *Car Physics*, Viewed 4 May 2004 on http://66.98.192.31/~maskedc/carphysics/carphysics.htm.

[16] G. Washington, *System Modeling Basics*, Web-published course notes for ME481 System Dynamics, Dept. of Mechanical Engineering, Ohio State University, Viewed 4 May 2004 on http://rclsgi.eng.ohio-state.edu/~gnwashin/me481/mech_sys.html.

[17] P. Goode, *ProjectM*, Viewed 1 Oct. 2003 on http://wave.prohosting.com/projectm/.

[18] D. Eberly, *3D Game Engine Design*, Morgan Kaufman, 2001.

[19] D. Badouel, *Graphics Gems I*, Morgan Kaufman, 1990.

[20] G. Gribb, K. Hartmann, *Fast Extraction of Viewing Frustum Planes from the World-View-Projection Matrix*, 2003, Viewed 4 May 2004 on http://www2.ravensoft.com/users/ggribb/plane%20extraction.pdf.

[21] W. F. Engel, *ShaderX2 Introduction and Tutorials with DirectX9*, Wordware Publishing, 2003.

[22] L. Haendel, *Functors to encapsulate C and C++ Function Pointers*, Jan. 2004, Viewed 4 May 2004 on http://www.function-pointer.org/functor.html.

[23] *Browser Statistics*, 2004, Viewed 4 May 2004 on http://www.w3schools.com/browsers/browsers_stats.asp.

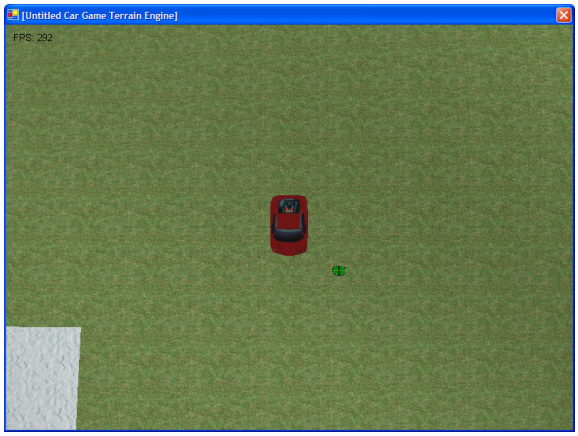
Appendix 1 – Gantt Charts

The Gantt charts on the following two pages show, firstly, the projected work schedule, and secondly, the actual work schedule. Although these schedules are fairly similar, the actual work schedule needed to be adjusted in light of certain requirements that changed as the project progressed. These changes were to do with specific features that were found to be impractical or overly complex for what was necessary for the game. For instance, although it had been planned to use an RK4 integrator to increase accuracy during numerical integration, this was not implemented because the Euler integrator was found to be sufficiently accurate. This allowed more time to be spent implementing more visible features.

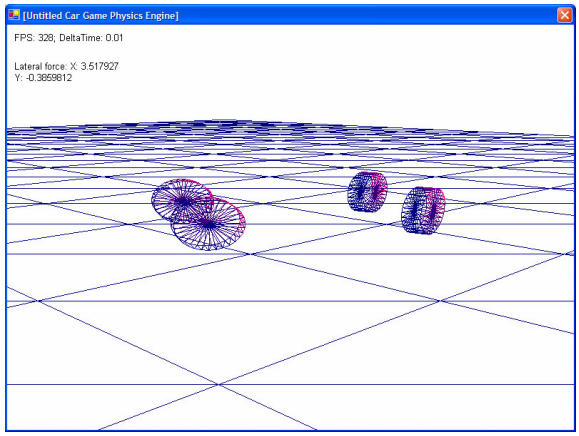
Figure 16 – Initial Gantt chart

Figure 17 – Final Gantt chart

Appendix 2 – Development History Screenshots

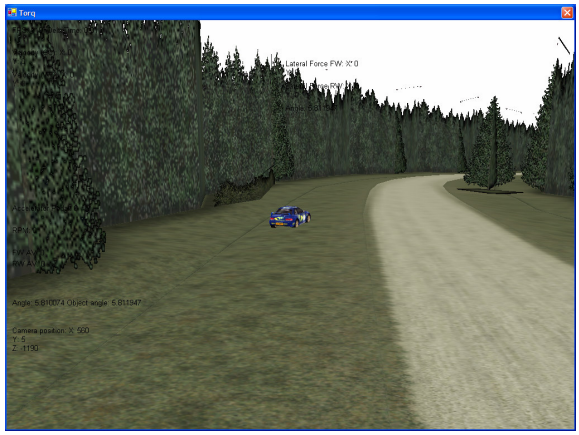


20th October 2003

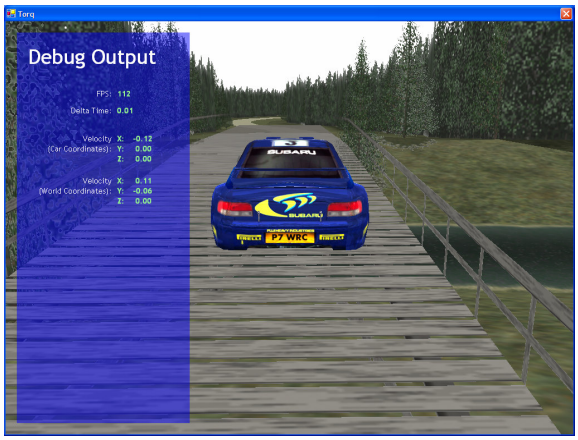


28th October 2003

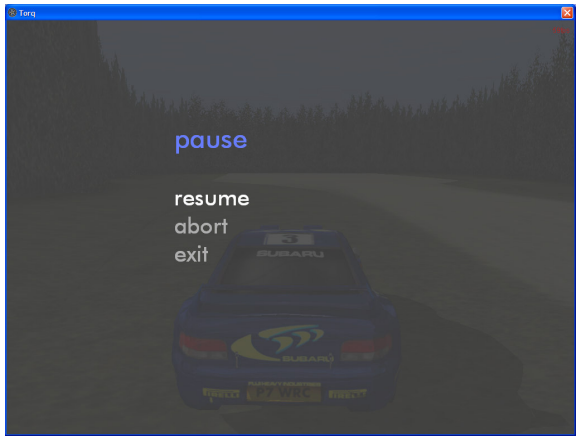
4th November 2003



18th November 2003



25th November 2003



9th December 2003

2nd February 2004

17th February 2004

Final version

Appendix 3 – Questionnaire

Please circle your answers clearly.

1. How would you rate the following areas of the game?

	Very good				Very poor
a. GUI	1	2	3	4	5
b. Graphics	1	2	3	4	5
c. Physics	1	2	3	4	5
d. Sound	1	2	3	4	5

2. Compared with other driving games, would you say the following areas are similar?

	Definitely				Not at all
a. GUI	1	2	3	4	5
b. Graphics	1	2	3	4	5
c. Physics	1	2	3	4	5
d. Sound	1	2	3	4	5

3. Would you play this game again?

Definitely				Not at all
1	2	3	4	5

4. Please briefly explain your answer to the previous question.

_____

_____

_____

_____

_____

Appendix 4 – Program Code

The CD-ROM attached to this page contains the complete source code for this project. This includes the source code for both the C# and C++ versions. It also includes the executable files for both versions, as well as all the data necessary to run the game (track and car meshes, textures and all configuration files). Please see the readme.txt file in the root of the CD for more information.