CAR SIMULATOR 3D

Interactive Driving Simulator

Jonny F. Bråten

Table of Contents

Deciding on a Problem: Goal & Thesis

Why a car simulator?

When deciding on what task to tackle for my project, I started by looking at what we had already learned, and which of those things I might want to further expand upon. My original idea was to attempt to make a three-dimensional flight simulator, but after some testing and prototyping I quickly realized I was likely to run into so many issues along the way that I was likely to not finish the project on time with the degree of complexity I wanted. I was, however, still set on the aspect of creating on a user-controlled experience while expanding the physics to all three dimensions. This is how I eventually decided to tackle the challenge of emulating a steerable car.

I liked the car idea, as it ultimately had a rather high degree of complexity, depending on how deep I was willing to dig. At the same time, the emulation of a car seemed modular enough for the task and complexity to grow or shrink depending on my needs. For example, the fact that I might have trouble properly implementing the effects of driving at an angle wouldn't necessarily get in the way of the rest of the physics and computations. Likewise I could probably find a way to exclude realistic effects of rolling friction or drag without having the car going haywire. Having the emulation play out in three dimensions was also an ideal way to further explore—hopefully with some degrees of success—concepts that was mentioned in the standard curriculum, but never expanded upon.

Goal

Although I was prepared from the get-go to keep my final goal flexible throughout the process, my overarching ambition was as follow: Develop a car-model with its core physics based upon real physical laws and have the car be steerable to a large degree by the user in a 3D-space.

When I began working on the project I was already well acquainted with the fact that combining coding, physics, and visual emulation all in interconnection with each other could easily lead to headaches and strange issues, which is why I made a conscious attempt at not make any part too dependent on each other to function properly. That said, I knew I wanted to implement steering and driving around curves, as well as the automatic gearing that was suggested as one of the assignments. Driving around curves was not explored in depth in the curriculum, so I wanted to try to implement it to some degree in my model.

Although I wanted to base as much of the movement on real physical laws, I was ready to tweak each one of them to make the actual interaction feel more entertaining, hopefully without completely eliminating the realism in the model. The idea behind this approach was to create an experience that felt playable while still being based of real life physical equations.

Lastly I wanted to add some sense of purpose to the simulation. Not necessarily anything too complicated, and it could be optional, but I wanted to at the very least add a mechanic that let you drive a given route and have your time be taken from start to finish.

The Physics

General Force Diagrams

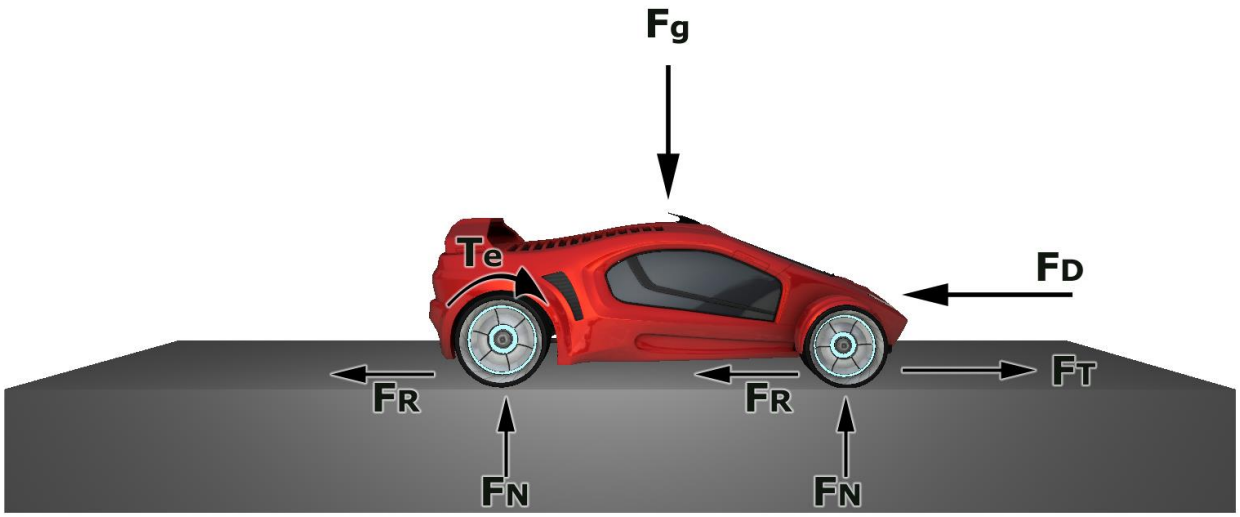

Figure 1 :Force Diagram for the car.

Te: Engine torque on wheels.

FT: The force applied to the wheels.

FR: Friction between rolling tires and the road.

Fg: The force of gravity.

FD: Aerodynamic drag.

FN: Normal forces from the ground.

Despite my simulation happening in three dimensions, the basic force diagram doesn't look that different from what I've used before. After all, force diagrams generally do a good job of explaining the forces acting on the object and how they may affect each other and this case is no different. My assumption in this diagram is that it remains as is regardless of where in the 3D-enviroment is located, assuming the wheels are properly grounded. Also, the diagram above also obviously assumes the car is driving forward. I've included the option to back the car up within the simulation, in which case all the horizontal forces has their direction reversed. The drag will be working onto the backside of the car, the friction will be pointed in the same direction as the front of the car, and the engine torque will rotate in reverse to apply the force to the wheels that moves the car backwards. The gravity obviously pulls the car towards the ground, while the normal force acting on the wheels balance out the force of gravity. While the normal force in reality is divided among the wheels, in my simulation I treat the normal force as a single force acting upwards, but the quantity of the force remains the same. In much the same way, the force of rolling friction would in reality act upon each individual wheel as it rolls on the ground, but I treat it as a single force acting on the car. These simplifications makes for easier computations, but otherwise doesn't have any big implications for my model.

As the car will be moving around in 3D-space, referring to single coordinates in the rather large 3D environment would be horribly ineffective and tedious. Instead I use the local position and orientation of the car to determine where forces should act. I'll expand upon this when explaining the simulation, but as far as the physics go, this means that the forward velocity Vx will always be perpendicular to the

car's front, and will be reversed if the quantity reaches negative values. Computing the velocity is one of the computations that happens last in the program though, so I'll hold off on that for a bit.

Driving on Inclined Surfaces

I've included the physics and computations in my model to take into consideration whether or not the car is driving up or down a slope, and how steep that slope might be. This is an essential addition to my model as the track in my simulation changes elevation along the way, and driving uphill or downhill affects a lot of the factors, and therefor has a large effect on the final computations. It also adds realism and better gameplay in ways like more accurately shifting gears depending on the angle you're driving at. As far as the force in the diagram above goes, the change will mostly be in direction, but the force of gravity will need to be decomposed in order to properly compute its magnitude at an angle. The decomposed vectors are F_{gN}, and F_{gP}, representing the force of gravity normal on the car and the force of gravity parallel to the incline respectively. The angle of the incline, θ, is for my purposes negative when driving uphill and negative when going downhill. For most of the computations I use, the angle of the slope is expressed in radians.

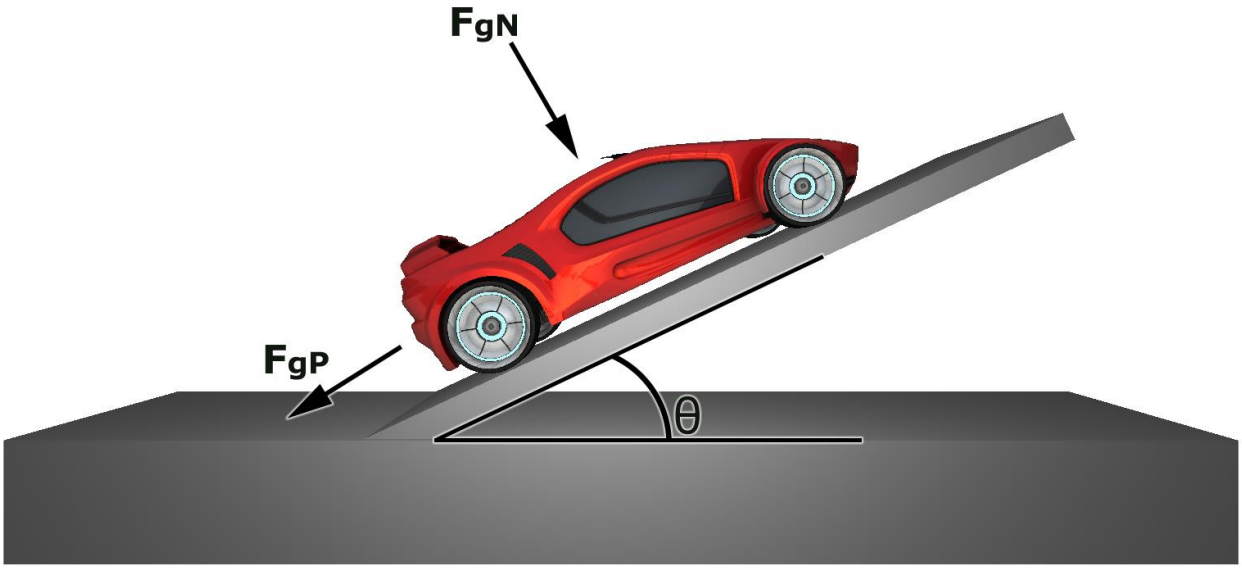

Figure 2 : Forces to consider on inclined planes.

Shifting Gears & Automatic Transmission

To get the actual shifting of the gears out of the way, I'll begin by saying they're not uniquely different to what I have already become acquainted with prior to my project. The act of shifting gears obviously changes the current gear of the car, and then in addition also changes the gear-ratio accordingly and computes the new rpm. The rpm after a gear-shift is computed from formula 8.14 in Physics for Game Programmers, and looks like this:

$$\Omega_e(\text{new}) \;=\; \Omega_e(\text{old}) = \frac{g_k(new)}{g_k(old)}$$

Where Ωe is the engine turnover rate, g_k is the gear ratio, and the new/old terms refer to the values before and after the gear shift has been performed.

One new challenge I knew I would be having from the get-go was to implement automatic gearing, but I didn't find this to affect the actual physics so much as it affects the general behavior of the car. The car in my simulation has six gear which it changes between automatically. In the physical world, the key internal difference between manual and automatic transmission is the way that manual transmission unlocks different sets of gear to the output shaft to achieve different gear ratios, while in an automatic the same set of gears can produce all of the different gear ratios. While this is interesting to make note of, it doesn't directly affect my simulation, but the conditions for when the transmission takes place does.

After reading up on automatic transmissions I've come to discover that they can be amazingly complex, to the point where a computer in the car can utilize algorithms to get an idea of your driving habits, then use the computed data over time to change gears in a way that fits your style of driving. As a lot of my work and time has gone into getting the car to act properly in a 3D-enviroment, I've considered something like this to be far outside the bounds of my project. In hindsight I do see now how one could probably make a model entirely focused on achieving a realistic automatic transmission, but alas that was not my goal. Instead I decided to take some factors that I found was used in reality, and tweak them into a formula that would fit my needs. The formula is my way to compute a value used by the car to know when to transition gears without relying entirely on the current *rpm*(expressing the engine turnover rate) of the engine.

The formula, although based on the same values a real car with automatic transmission might utilize to determine when to shift gears, is something I made by mostly trial and error. I used the pure rpm-value as an outset for the value, and looked at how I could influence this value with quantities such as the current speed of the car, its acceleration at the time, as well as the vertical angle the car might be driving at. In lack of a better name I'll simply refer to the value here as the shift reference value.

$$ShiftReferenceValue = \boldsymbol{rpm} + \boldsymbol{V_x} * 1.5 + \boldsymbol{\alpha} * 3000 - \boldsymbol{F_{gP}} * 0.35$$

The *rpm* is the current rpm of the car, *Vx* is the forward acceleration, and *α* is the current acceleration of the car. F_{gP} is the gravity affecting the car parallel to the slope it is driving on given that the car is driving at an inclined angle. In my computations this value is positive when acting in the same direction as the back of the car. In this context that means that if the car is driving upwards an inclined angle, a higher rpm, velocity, and acceleration will be required for the car to shift up. Likewise, if the car is going downhill then car will be more likely to shift up, and less likely to shift into lower gears.

The inclusion of acceleration is somewhat self-explanatory when you consider the context in which a car would ideally shift gears. If the quantity of the car's acceleration is high, then it is far more likely that shifting into a higher gear will be optimal for the car's performance. If the car is slowing down, that makes for a negative acceleration, meaning the car will more easily shift to a lower gear to make up for the decrease in speed.

The forward-speed is a very simple inclusion, and it doesn't have that big of an effect on the formula. I simply found trough testing that it would generally be ideal to shift at lower rpms for the top gears. For

example, my car would have a much harder time reaching a high rpm-value when in fifth gear as opposed to third gear. This meant that the car could potentially have a much harder time to shift into sixth gear than it should have, hence the inclusion of speed in the formula.

Lastly, the factors multiplied with each value have no significant value side from making the formula function properly within the simulation and emphasize the different values as I've deemed fit. The car looks at the shift reference value for every time increment that passes, and when the values gets above or below a set minimal value and a set maximum value, the car automatically performs the required transmission accordingly.

Drag & Friction

A car in motion will be subject to aerodynamic drag, which I have accounted for in my simulation. As illustrated in my force diagram, the drag works in the opposite direction of whatever direction the car is traveling in. In my case that means the drag will act on the front of the car if the car is moving forward with a positive velocity, and on the back of the car if the car is backing up. Either way the drag will function as stalling effect on the car and reduce the rate at which the car can accelerate. Aerodynamic drag has come up in a lot of situations while studying this subject and as a consequence the formula with which I compute it is one I've become very familiar with over the course of the semester.

$$F_D = \frac{1}{2} C_D \rho v^2 A$$

C_D is the drag coefficient, ρ is the air density, v the current velocity, and A is the characteristic area of the car. For the purpose of my simulation I've assumed the drag coefficient to be constant, and the same goes from the air density and the characteristic area. Assuming a constant drag coefficient doesn't make that big of a difference in this context, and the elevation changes in my project are so small that the inclusion of varying air density would have miniscule impact. For my computations I use an air density of 1.2 kg/m3, and the drag coefficient is set to 0.31. The visual car in my simulation is not modelled after a real car, and as such I felt at liberty to not model my car properties after any specific car either, but seeing as the car has the general shape of a sports car, I'm using a drag coefficient that is common for that type of car. Similarly I've computed the frontal area of the car by using the frontal area of the Porsche Boxter S sports car in order to get an estimated value for what might be the area of my virtual car.

The rolling friction, as the term suggests, is used when referring to the force of friction that resists the rolling motion between the tires and the road. Referring to this force as friction can be somewhat misleading though as what is happening is not really what we would generally associate with friction. Rather than being pushed along a surface, the tires simply roll across it, and therefore there's not really any motion between the wheel and the surface, as any given point on the wheel only touches the road one single time per revolution. What does happen however, is the deformation of the tire, and potentially the surface it rolls across. A regular sports car isn't likely to cause any notable deformation on a solid asphalted road, but it's something worth taking note of nonetheless. The rolling friction works in the opposite direction as the car's velocity and is computed by the following equation:

$$Fr = \mu_r * m * g$$

The mg simply makes up for the horizontal forces acting upon the car by multiplying the mass of the car by the acceleration of gravity. μ_r is the factor worth taking note of here, as it is a coefficient used exclusively for rolling friction, simply referred to as the coefficient of rolling friction. The coefficient is constant and evidently ranges from 0.01 to 0.02 for tires. In my simulation it is set to 0.015.

Engine Torque and the Power applied to the Wheels

In order to compute the engine torque, I first need an idea of how the torque line looks, and at what point my car is located in relation to the curve line at any given time. For my simulation I have simply based my torque data on the one used for the Porsche Boxter S sports car. For simplicity's sake the torque curve is made up of three straight lines. The program looks at the car's rpm, then compares that to the torque curve to determine the constants b and d, which is then used to compute the engine torque.

$$T_e \;=\; b * \Omega_e \;+\; d$$

Computing the Acceleration and Intermediate Velocity

In the end, computing all of the forces acting on the car pays off in the manner that the results allows me to compute the acceleration. In the end what I really want is to know the velocity at any given time of the simulation, but the obvious way of going about this in terms of physics is obviously to utilize my knowledge of the acceleration, coupled with a time increment for measuring the passage of time. My general equation for the acceleration looks like this:

$$a = \frac{dv}{dt} = c_1 v^2 + c_2 v + \; c_3 + c_4$$

The key here are the four c's in the equation. Each one is computed on their own, and then plugged into the equation for the current acceleration. Every constant is computed by using the forces and gear-values computed prior to this point.

$$c_1 = -\frac{1}{2} * \frac{C_d \rho A}{m}$$

$$c_2 = \frac{60 g_k^2 G^2 b}{2\pi m r_w^2}$$

$$c_3 = \frac{g_k G d}{m r_w}$$

$$c_4 = \; -\mu_r g \cos\theta - g \sin\theta$$

Most of these terms have already been explained. The only new one would be the current gear ratio *G*, which changes depending on what gear the car is in. The 60-factor of c_2 is there only to convert the rpm from revolutions per minute to revolutions per second.

The c_4 term includes the factors that are affected by the angle of the terrain the car is driving at. It uses the slope-angle in radians to compute how the force of gravity will affect the car in the given situation. I found it useful and thought my computations became both easier to read and easier to handle within my simulation by isolating these within their own constant. If I have a concern with how anything related to the incline of a plane is affecting the acceleration I need only to look at this constant.

Note that this computed acceleration is the maximum acceleration can reach at a given time, and assumes the driver is pushing the throttle all the way down, which is obviously not always the case.

The Lateral Force around Curves

I've set up computations for computing the *lateral force* on my car when doing a turn. This is the force that pushes the car outwards when driving in a circle, eventually making the car slide outwards. I looked around trying to find a good way to compute this, but eventually just settled on having the lateral force equal the difference in centripetal force acting on the car and the friction on the tires. Aside from the desire to explore new aspects of the physics of car driving, my reason for implementing this was mainly to try to add more fun and playability to my simulation. Realistically, the slip wouldn't be nonexistent, but you would need pretty sharp turns at high speeds in order to achieve some truly significant skidding on dry asphalt.

$$F_{lateral} = \frac{mv^2}{r_c} - \mu_k mg\cos\theta$$

The r_c-term here is the car's turn radius, which is the radius of the circle the car would drive in if there was no slipping and the angle of the wheels remained constant.

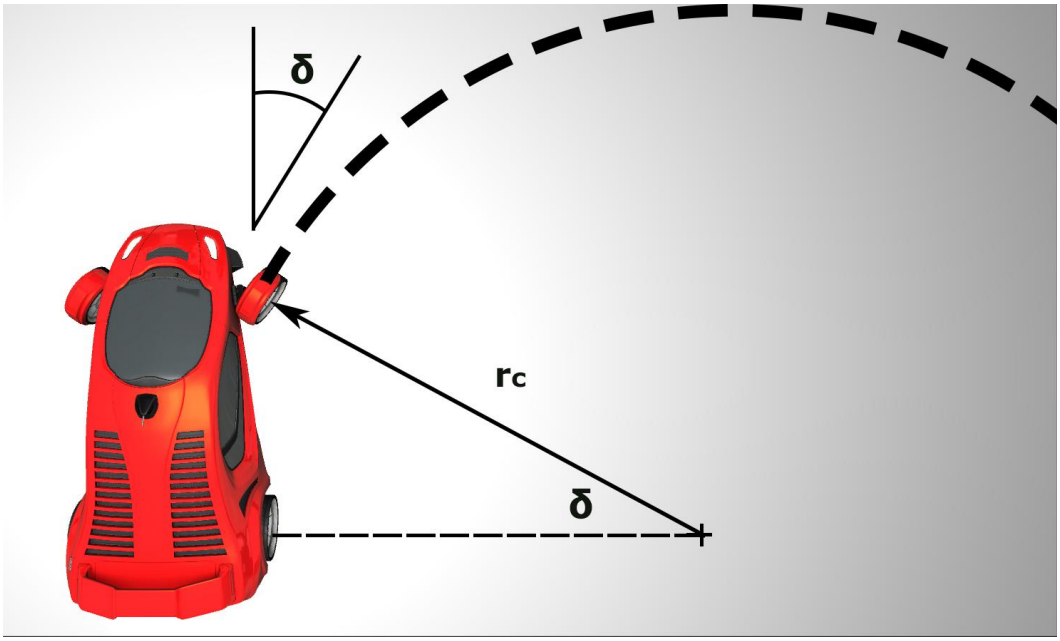

Figure 3 : Driving around curves.

Its magnitude is computed by the equation:

$$r_c = \frac{l}{\sin \delta}$$

Although not entirely happy with the result, I wanted to include skidding to the point I was able to satisfactory replicate it. It doesn't act quite like I want it to, but I think it has some merit to it, and with all the time I've spent trying—and mostly failing—to make it work based on real physical models, it seemed like a shame to include it at all. In the end, including it also made going around curves easier for the player, and in my opinion it makes the driving more comfortable despite its flaws.

The Simulation

Figure 4: A screenshot from the simulation, showing the car driving on the track.

To visually present my model I've used the Unity Engine, as well as a series of standard assets included with the software. Unity has its own physics engine, and a lot of its tools have been made with this in mind, but if you're willing to go through some hoops you can manipulate the visuals just fine and have it be completely independent from Unity's own physics. In addition to representing my visuals, Unity also helps me easily detect collisions and contact between the 3D objects that represent my physical objects. This could have been done manually of course, but creating my very own collision-detection for a three-dimensional world like this would have required a lot of effort, and I don't find that the question of whether two objects are within touching-distance of each other is a question worth putting too much weight on in the first place considering the curriculum. I will however do my best to properly detail how Unity helps me and most importantly how it affects the physics and behavior of my model.

Collision & Contact

Unity detects collisions and contact between physical objects by comparing the coordinates of *colliders*. Colliders can be attached to any object and shaped to fit its edges if desirable. Each collider has their own 3D-coordinates in the world, and if the coordinates of two colliders were to overlap, a collision will occur. Not that colliders can also be used simply as triggers, in which case an collision occurring doesn't necessarily hinder the movement of the objects. Triggers can be used to detect when object when objects enter the collider of an objects without actually making a physical collision occur, but the detection still allows you to call functions within your code when the overlap happens.

For my purposes, the most important thing the colliders do for me is to ensure that my car does not simply fall through the terrain. Each wheel on the car has its own collider that runs along its edges, and the road the car is driving on has a collider on its own. The force of gravity will be default drag the car downwards, reducing its coordinates along the Y-axis in the world space. When an object such as the

tire collides with the ground however, the engine knows that the object has hit the terrain and as such will no longer continue to fall.

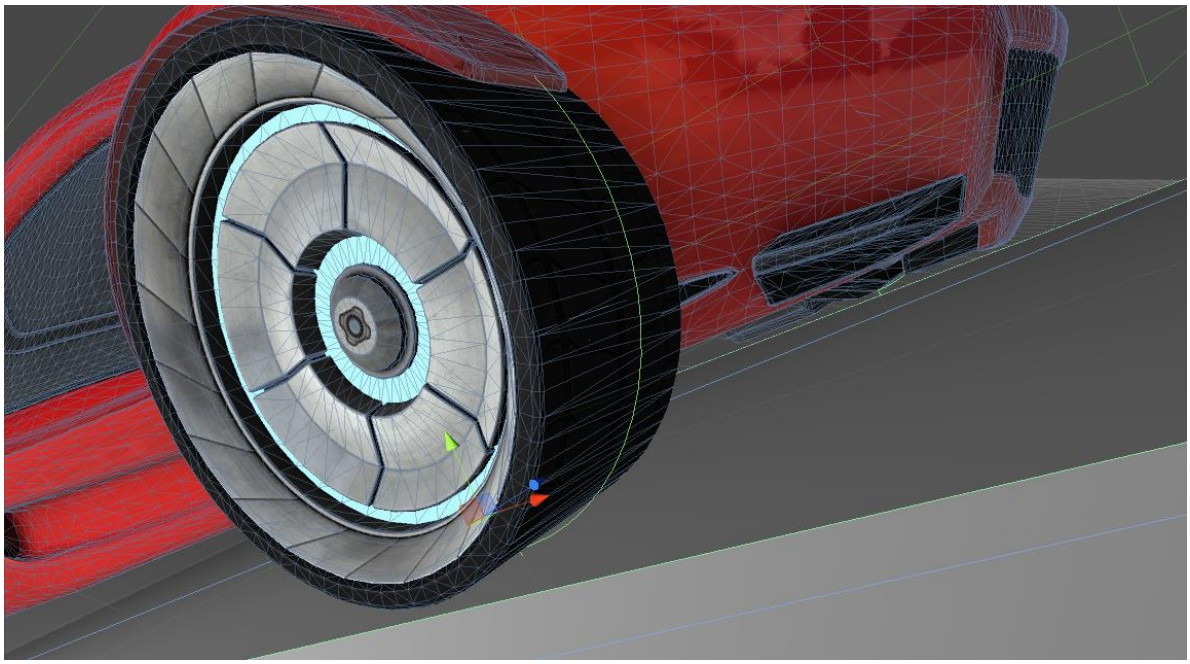

Figure 5 :Colliders on the wheels detects contact with the ground.

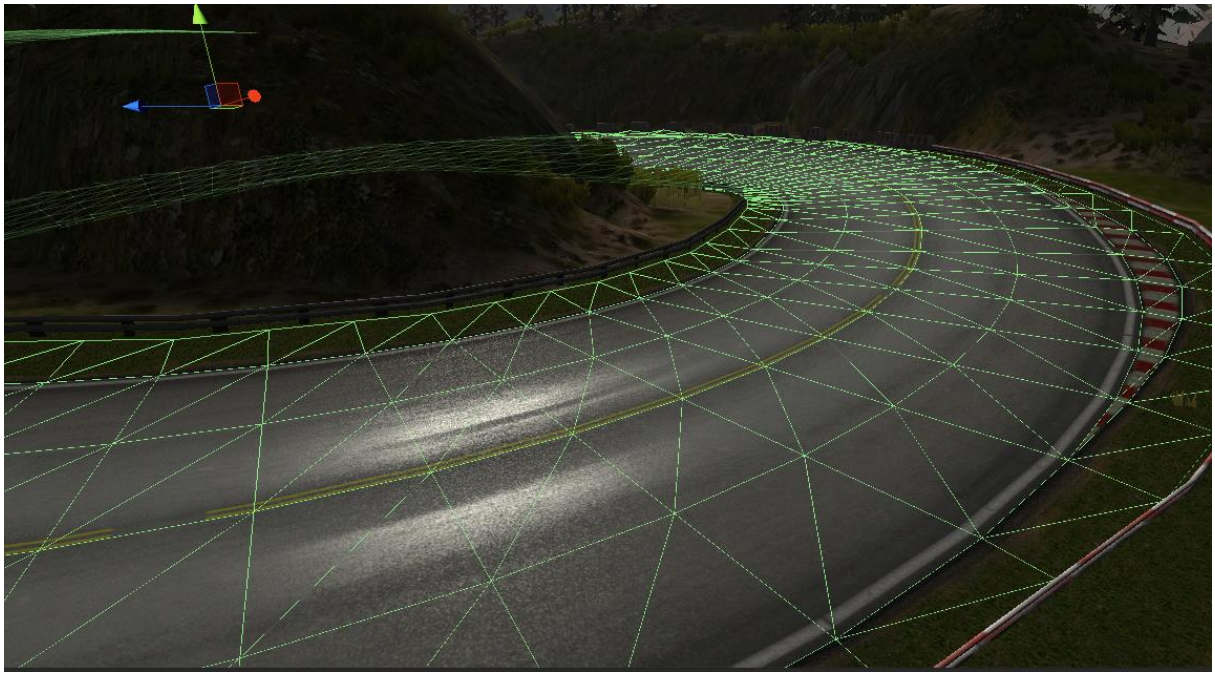

Figure 6: Colliders along the road lets the car know when it has contact with the ground.

In my simulation, it is also not possible to accelerate or apply the brakes if the car for some reason is not grounded. This makes sense, as a driver obviously wouldn't be able to be in control of their car's acceleration if the wheels can't make contact with the ground. The only thing affecting the car's forward

acceleration when the wheels have nothing to make traction on, is the aerodynamic drag that affects the car as it passes through the air. In the end this is simply a small touch, and not very essential for my simulation as the car should stay grounded for most of the time assuming nothing breaks. It is achieved trough “raycasting”, in which my car ejects a short ray/line, invisible to the player. The line is directed from the bottom of the car and a short distance downwards. The ray is of such a length that it can be used to determine whether the car is properly grounded. If the ray touches the ground, the script allows the player to accelerate, if the ray looses contact with the ground, control is temporarily lost.

The other use I have for colliders is to detect whether the car physically collides with something ahead or behind it. Colliders are placed at the edges of the road to make sure the user cannot simply drive off the road and trough the surroundings. This is where what we would in the context of car-driving would usually refer to as actual collisions come in. Implementing the physical implications of a physical collision was on my to-do list of things to implement in my simulation, but it wasn’t high on my list of priorities and in the end I never implemented it in the final model. Proper collisions would not make that big of a difference in my opinion, as creating some sort of real-time deformation would be far too time-consuming, and the only thing to collide with is solid terrain. Were I to have implemented realistic collisions not using deformation, every collision would simply be an elastic collision between the car and an immovable object. This would cause the car to bounce in the opposite direction of the collision.

Although my simulation does not contain realistic collisions, I have added a function to initiate a sort of inelastic collision. If the car drives into something or back up into something, a function is called that nullifies the speed and acceleration of the car, as well as making it unable to keep driving in the direction in which the car’s path is blocked. This ensures that the car cannot simply drive trough terrain and makes the driver unable to keep accelerating the car into an obstacle. If the car were to collide in something in front of it, the reverse-function allows the driver to back up and get back on course.

Gravity

When struggling to make my own physical model utilize all of Unity’s tools I was able to root out most problems, but implementing my own force of gravity proved difficult. My own gravity and the colliders didn’t play well together and after several attempts to stop my car from falling through the terrain, I decided to simply use Unity’s gravity to the point I needed to.

This is the only physical property Unity handles in my model, and it is limited to the Y-plane. That is to say, Unity’s gravity makes sure the car falls downwards if it does not have contact with the ground, and ensures it stops falling as it hits the ground. Any other effects of gravity, such as the additional force required to drive up an incline, or the potential for the car to start rolling when on a steep slope, is all handled by my own model. I want to emphasize this as the goal of this assignment is to make my own physical models, and as such I have done my best to not have Unity interfere with any physical properties in my model, and aside from the Y-plane of gravity I’d claim I have succeeded in this.

User Input: Throttle & Turning

My simulation allows for user-input to steer the car using set key bindings. Using the vertical arrow keys or the W/S-keys on the keyboard, the throttle can be increased or decreased, and the brakes can be applied. Meanwhile, the horizontal arrow keys or the A/D-keys can be used to turn the wheels on the car. Input is basically divided among the vertical axis and the horizontal axis, one axis controlling their own property. The inputs along these axis's also have some 'weight' to them, meaning that you first of all need to hold down the forward-acceleration key for a small amount of time until you actually reach the maximum amount of throttle-input. Granted, in reality you have no problem simply pushing the gass-pedal to the floor with all your might, but in terms of controls in this simulation I think it feels better and more natural to have this be different. The faster you would have the input reach full throttle, the more sensitive it would have to be, and high sensitivity makes driving at somewhat stable velocities a lot harder. This 'weight' also works the other way, so if you're going on full throttle and let go off the key, it will take some time for the throttle to return to its natural section, and will gradually decrease in order to reach it. The exception for this is if you go from throttling to braking or vice versa. In that case, in order to achieve the appropriate responsiveness, the input axis will 'flip', causing the input to jump instantly to the neutral input and then begin to progress in the direction of the new input. This means that even if you're driving at high speeds and you have the throttle all the way down and you decide to initiate the breaks, you will instantaneously begin to brake, and the throttle will be let up.

The steering, or the turning of the wheels, has some additional logic attached to it that is altered within the code. This is because the way that the steering input it processed actually changes depending on the state of the simulation and the car. The way the steering works is that it changes the direction of the velocity by a factor decided by the angle of the wheels. The angle of the wheels in return is decided by the input from the horizontal input axis. For starters, the input-axis for the wheels does not flip. That's to say that even though you reverse the input while the wheels are turned 20 degrees to one direction, the wheels will not instantly jump to the neutral position like the throttle, as this wouldn't make a lot of sense. Instead you will need to wait as the wheels turn gradually back to the neutral position and then past it. The wheels will also try to return to their neutral position on their own if the user makes no input to change or maintain the angle, not unlike the throttle.

The biggest difference in the actual turning of the car is that the rate of turning is affected by the speed of the car. In reality, you will feel more resistance in the steering wheel the faster you're going, making it harder to make sudden large turns as you reach higher velocities. In an attempt to re-create this affect, I've set my code to decrease the effect of the input has on the turning of the car at higher velocities. This has the effect of allowing the user to easily make the car turn at the maximum turning angle of its wheels relatively quickly, but turning at the same angle at higher velocities will take quite a bit longer. This also makes steering more comfortable, as having the car turn as quickly at high speeds as low speeds would make the steering feel overly sensitive and unrealistic, and it would take a great deal of precision in order to make high-speed turns without driving the car straight off the road. To accomplish this in the code, I simply decrease the responsiveness of the car's turning by a factor which increases with the current velocity.

Much in the same way I've made the velocity of the car limit the speed of turning at high speed, I've also made the angle of turning limit the velocity, or the acceleration more specifically, when the car is driving at lower speeds. If I were to let the car accelerate while driving in circles as if it was driving

straightforward, the car would be able to keep acceleration while driving in circles until it reached its maximum velocity. As this is not the case in reality, I've altered the total acceleration to decrease based on the angle the car is turning at to avoid this.

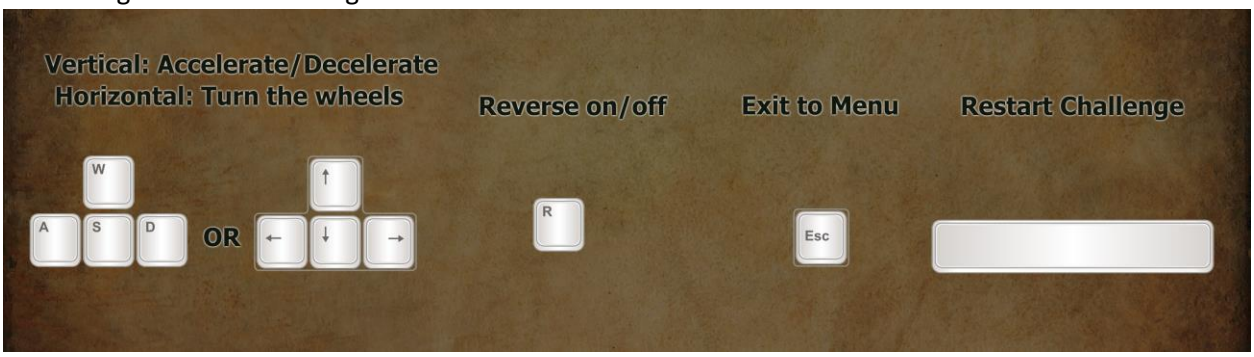

Figure 7: User Controls.

Moving the Models

All movement that takes place within the simulation (aside from gravity in the vertical direction, as specified earlier) is caused, in one way or another, through my own physical model. This includes user input, as all input also goes trough my code and is processed and applied accordingly.

The most obvious thing and important thing that needs to be moving is of course the car itself. In an attempt to re-use code I've already used and maintain modularity, the code for my physical model is already set up to alter the horizontal coordinates of the car based on the computed velocity and the time increment used. There's a slight disconnect here as manually inputting each specific coordinate in this setting would be very complicated and tedious, and so I've utilized the physical model slightly different than what I originally did. To determine the amount the car moves, my simulation looks at the value of the x-coordinate before a time-increment, then compares that to the x-coordinate in the model after the next time increment. The difference in between the two coordinates is translated into forward movement of the actual 3D-model of the car. The forward movement of the car is relative to the car's rotation in the world, so the change in x-coordinates within my model will always correspond purely to forward movement of the car in the 3D environment. This works well, and the only alteration truly worth mentioning is the balancing of making the change in location in the code fit with the change in location of the car-model and look natural compared to the size of the world. I therefore slightly alter the amount of change in coordinates before applying it to the car in order to make the amount of movement believable and manageable. Note that this also works regardless of inclines or whatever angles the car might be turned at.

Other manipulations of the models in my simulation is the rotation of the car's tires, as well as the turned of the car's front wheels. The angle of the wheels are in direct correspondence to the user input described in the input section, and rotates to match the angle decided by the input. Of course, the location of the front wheels (or all the wheels for that matter) are also linked to the car's location, so although they can rotate independent if the car, their actual location follows the car.

The rotation of the wheels due to engine torque should be in direct correspondence with the forward or backward movement of the car, as it is of course the rotation of the tires that drives the car forward

(assuming the tires are rolling without slipping). To simulate this, the speed of the wheel's rotation is the same as the velocity of the car. This creates the visual effect of making it look like the car is in fact moving forward due to the force applied to the wheels from the engine.

Audio

An audio script handles the audio effects, which is simply a modified version of what Unity's car assets already used to handle sound. I also utilize the Unity audio-affects, all in order to give sound to the engine, and have it be reflective of how the car is currently driving. To determine the pitch and sound the engine should make, the audio script accesses the current engine turnover rate, as well as the forward speed of the car, then uses those to make a sound that in turn sounds far more dynamic than a binary on/off engine sound on loop. I've also added a couple of audio effects for the occasion that the car crashes. The audio effects wary in intensity, and the script chooses what one to use by referencing the speed of the car at the time of collision.

For the music I created a separate script that can change the type of music depending on what mode the player picks, and make sure it starts playing when it's supposed to do so. Music also presents the issue of repeating the same track over and over again, and in a mid-race setting this should preferably be seamless. I accomplished this by editing the music with audio-editing software, where I made a looped version of the main mid-game tracks, and then separated their intro into its own track. This way, the script can begin by playing the intro-track, and then transition into the rest of the track when it reaches the right spot in the music track, and then simply put that track on loop until given further notice. Music might seem inessential to some, but try driving with music and then in complete silence and I'm sure you'll agree that the effect it has is quite considerable.

Adding Challenge & Competition

I knew I wanted to add some sort of goal to my simulation from early on. Not necessarily something big and complicated with several degrees of scripted success/failure scenarios, but something that would give the player a sense of purpose. Simply driving around a track alone would probably be sufficient for showing of the car, but I wanted something more, and the most time-efficient thing I could think of was to have the game track the player's lap time and allow them to race against their own record. I accomplished this with logic and code that tracks when the player passes the finish line and the time in between the last time the car passed it and the current one. I've added a tracker on the car itself that the goal-post looks for when something reaches it, and if the tracker sees that it's the player passing by, the timer and time records are updated accordingly. In addition some more GUI element were added to display the player's current time, as well as their fastest time as of yet.

The other choice of objective for the player is a race. The race lets the player race around the track against three other cars. Note that these three cars are ***not*** *of the same kind as I have made and explained here*, although their models look the same. Programming AI to work with my script would be very time consuming and would add a lot of complexity mostly irrelevant to my physical model, and as such I utilized Unity's AI car-scripts in order to add some more exciting competition, even though the cars do not all act the same. These are also "ghost cars", meaning they will ignore collisions, and are simply there to give the player a clear and visible goal to aim for. The race lasts for three laps around the

track, and once a car finishes the third lap they are ranked depending on their placing. When the player reaches goal the UI changes to let the play know the race is finished, and shows them how they did.

I've made one small change to the Unity-cars, which adjusts their maximum speed slightly depending on the current speed of the player. This is an attempt to make the race a bit more even and adjust the difficulty based on the player's skill. The AI-script gets the current speed from the player and dynamically adjusts the maximum speed they will attempt to drive at. With this, the faster the player drives, the more the competitors will try to speed up, and the slower the player is, the slower the competitors will drive, giving the player a greater chance at catching up. The settings for each car have been set beforehand. The blue car being the slowest, the green being faster, and the yellow one being about as fast as the green one, only that it has a greater reaction to the player's behavior, and will therefore be the fastest assuming the player doesn't perform like absolute rubbish.

Figure 8: Three AI-controlled cars are used for the race, although they do not utilize the player-physics.

Menu & GUI

The one piece of GUI you will always see when driving is the display showing the current status of your car; this includes the current speed, rpm-value, the gear the car is currently in, amount of throttle applied, as well as an on/off value telling you whether the car is in reverse or not. These values are displayed by its own class, but the values are taken directly from my Car class, which contains the physics. Up in the opposite corner of the screen, information will be displayed depending on what mode you choose. For time trial it will display your current time and your best record for one lap, while in a race you will see the number of laps which lap you are currently on.

I also added a main menu of sorts to make the application more user-friendly. This lets the user easily choose the mode they want, quit the application, and adjust the music volume if they wish to lower it or turn it off completely. This way the user may also play a race until finish, or play a time trial until they are satisfied, then simply exit back onto the main menu and choose another mode if they wish to do so. Each mode may also simply be restarted with the push of a button (space) without going back to the menu.

Code & Unity

I've commented my code thoroughly, and I believe my comments alone will hopefully be enough to explain everything that goes on in my code, but there are a few things I feel like I should address. These are mostly things that specific to Unity, and might look odd in an otherwise plain script that doesn't utilize any graphical-extension. All code is written in ***C#.***

Coding in C# for Unity

First there are the publicly declared variables that often shows up at the beginning of my scripts, and some of which are never visibly set in the code. These are usually references to Unity's "game objects", such as specific model-parts of the car in the CarSimulator-class, or most of the audio tracks in the MusicHandler-class. Declaring variables such as these in Unity makes those variables appear in the Unity engine's editor. This allows you to set variables directly, without altering code. In most cases, it is more efficient to directly reference game elements in this way, and so if it seems some public variables are not set within the script, it means they are set by default to reference a game element. The alternative is to search the game-world for the object you're looking for using *GameObject.Find()* and sometimes GetComponent<>(). I do use these at some points out of necessity. The more efficient method of directly referencing public variables within the editor, can only be used if the game object (or the particular instance of that game object you want) exists in the content of the script in the context you're working on in the editor. For example, I cannot use the editor to directly reference the player car to any script while I'm on the main menu, as the car does not yet exist in that current context.

The other thing I'd like to quickly clarify are some Unity-functions that keeps on repeating themselves, and as they are the fundamental building blocks of Unity-code, I feel like they are worth mentioning here.

- **Start():** This function is called at the start of a *scene*. In my case, my game consists of two scenes: The first of which is the main menu, the second being the racing track. This means that any element present on the racing track, such as the car and its script, will have its start-function run at the time that scene is loaded.
- **Update():** Update is simply the loop of the Unity-engine. It is called one per frame and is the place to check for input, update variables, and check for conditions of interest.
- **FixedUpdate():** This works largely like the update-function, with the only difference being that it is frame-independent. This comes in handy, and makes it a good fit for physics, which is why all the physics I apply is within this function. Having physics within the regular update-function would make the physics frame-dependent, which means the quality of the frame rate would affect the physics. We obviously don't want this, which is why it is used for all physics, and is called every 0.02 seconds regardless of frame rate and performance.

The Runge-Kutta solver

A few notes on runge-kutta. I use the runge-kutta solver taught in the book, which takes the form of a class capable of integrating 4th order ordinary differential equations contained within class-instances of the ODE-class (the car in my simulation is an instance of a car-class that inherits the properties of the ODE-class). This method does its integration by using a trial step at the midpoint of an interval to cancel out lower-order error terms. Essentially, it estimates a series of locations for the new time increment by looking at the car's velocity for different values of a variable x, and the time, t. First one estimate is done using Euler's method, then a new one is made by looking at the velocity halfway between the known position and the first estimate, and then it does so two more times until we're left with four estimates, which is then added together linearly to get the final answer/estimate.

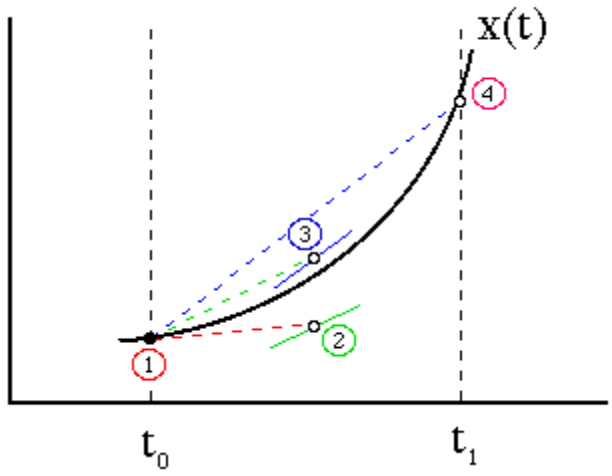

Figure 98: A visual portrayal of the fourth order runge-kutta method.

Solving the equations is what allows the program to compute the updated position of the car per physics-update, based on the speed and acceleration. An array in the car-object is filled with the equations for the car's acceleration for the different coordinates, and calling on the ODE-solver solves them based on the values the arrays have been filled/updated with.

Known Issues & Potential Improvements

Collisions

Collisions are only included in the most simple form possible. While I still believe including deformation would be prove to be quite a large task, there is at least potential for improvement by adding simplified collisions that ignore deformation, and decided how elastic the collision is by other means.

Wheel Traction

My code actually contains computations that are meant to determine whether the wheel traction of the car is good enough not to spin, but although I implemented this in the Car-class, I do actually utilize it in my simulation. This would have made the physics more complete and realistic, but in my case I don't really see any contexts in which having your wheels spin against the ground would improve the experience of playing it in any ways, so I wouldn't necessarily say the application suffers from the lack of spinning.

Slipping in Curves

As mentioned I'm not entirely happy with the way slipping in curves acts. I don't mind it not functioning as in reality, as my ultimate goal was to base the movement of real physics and then adjust that data through my simulation to make a more fun experience, but the way it acts doesn't feel quite like I'd want it to feel either, and with more time I'd like to try and re-implement it in a way that makes driving around sharp curves at high speed more fun.

Cheating

I have not implemented a system that can detect whether the player has truly finished a round. The system simply detects whether the player-car passes the finish-line, and as such the player could potentially just drive in circles at the finish-line to easily win the race or get ridiculously low time records.

Robustness

I've experienced more odd issues during the making of the simulation that I care to count. Car's falling trough the terrain, bursting trough mountains, inexplicably taking off into the air, and the list goes on. While a lot of time in making this has gone into rooting out these issues, I doubt I've found and fixed all of them. Some bugs happens irregularly and are hard to replicate, while others make so little sense I'm not entirely sure what might be causing it in the first place.

As of now, the main issue that I'm actually aware of is the potential for the car to crash slightly sideways into the side of the road, and then get stuck. If the car travels far enough it might also clip into the terrain, and at the very worst the player might be able to force the car into the terrain to the point where the physics goes bananas, to use the technical term. I have no implemented solution for this, but at the very least the player may restart the stage to get unstuck and try again.

I've also encountered an issue where the AI messes up and drives off the course or otherwise disappears, but this has only happened once or twice, and I do not what cause it or how to replicate it.

Controller Support & Motion Blur

A couple of pure gameplay-related features I experimented with adding but has not yet been able to successfully implement as this hand-in, is controller support and motion blur. With controller support I'd like to be able to control the game using an Xbox 360 controller in order to make the steering of the car more fun and less clunky, while my purpose with the motion blur would have been to give the player a greater sense of speed as they're driving.

Reference and Resources

The parenthesis behind each URL-adress is the last date I visited the site and confirmed it was still up and running.

Unity Sample Assets:

This includes all the models and textures that make up the car. It also contains every script used in the emulation that was not made by me, plus the original versions of those that I used and modified for my purposes.

https://www.assetstore.unity3d.com/en/#!/content/14474 (14/11/14)

The car track was extracted from a car-project Unity put out on their Asset-Store:

https://www.assetstore.unity3d.com/en/#!/content/10 (14/11/14)

Other Graphics:

Tutorial for stopwatch icon: http://www.vectordiary.com/illustrator/how-to-create-a-vector-stopwatch-tutorial/ (20/11/2014)

Racing flags icon: http://pixgood.com/racing-flag-png.html (20/11/2014)

Runge-Kutta Visual: http://www.physics.drexel.edu/students/courses/Comp_Phys/Integrators/rk4.html (2/12/14)

Editing, drawings, and creation of graphics such as the menu-background was done with Photoshop CS6.

Music:

Mario Kart 8 Menu Music Variation – Ryo Nagamatsu

Mario Kart 8 Stadium – Ryo Nagamatsu

Main Menu Screen for USFIV - Hideyuki Fukasawa

Mario Kart 64 Countdown SFX – Kenta Nagata

Information & Theory:

Rolling Friction: http://physics.tutorvista.com/forces/rolling-friction.html (29/10/14)

Car Physics for Games: http://www.asawicki.info/Mirror/Car%20Physics%20for%20Games/Car%20Physics%20for%20Games.html (11/20/2014)

Motion on a Curve: http://www.ux1.eiu.edu/~cfadd/1150/05UCMGrav/Curve.html (15/10/14)

The Book: *Physics for Game Programmers* by *Grant Palmer* was used as a basis for a lot of the physics, as was its most general classes and its implementation of an ODE-solver. The full source code that accompanies the book can be downloaded here:

http://www.apress.com/downloadable/download/sample/sample_id/705/ (20/11/14)