Car Simulation for Neurofeedback Research

Bilal Arslan & Patrick Jørgensen

Advisor: Jakob Andreas Baerentzen

Kongens Lyngby 2012
IMM-B.Sc.-2012-28

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-B.Sc.-2012-28

Summary (English)

The goal of the thesis is to create a car-simulation game, with the objective of being the tool used to conduct experiments for a research project in neurology. This thesis describes the steps we have made to transform a relatively simple document, into an accurate environment, and later into a fully working car game. We describe the steps the program takes, first to create the environment. Then we explain how we made a car game, able to drive all the scenes created, and react the way it should be.

When the simulation was ready to be presented, Hvidovre Hospital tested our game. The software lived up to their expectations and the results were very satisfactory. All in all, the researchers at the hospital were pleased with our work, thus we had fulfilled our goal.

Summary (Danish)

Målet ved projektet er at skabe et bil simuleringsspil, med henblik på at være redskabet bag eksperimenter for et forskningsprojekt i neurologi. Denne rapport beskriver de skridt vi har taget for at forvandle et forholdsvis simpelt dokument, til en præcis verden, og senere til et fuldkomment bil spil. Vi beskriver de skridt som programmet tager til at skabe miljøet. Dernæst forklarer vi hvordan vi har lavet et bil spil, som er i stand til at køre alle scenerne igennem, og reagerer som det skal.

Når simulering var klar til at blive præsenteret, testede Hvidovre Hospital vores spil. Softwaret levede op til deres forventninger og resultaterne var meget tilfredsstillende. Alt i alt var forskerne på hospitalet glade for vores arbejde, og derfor har vi opfyldt vores mål.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an B.Sc. in Informatics.

The thesis deals with procedural modelling of a scene in a car-simulation game and how we can generate an environment out from simple input parameters. Furthermore, the thesis also deals with game development and how we can integrate the environment in it.

The thesis consists of different techniques and algorithms used to fulfil such a task. Moreover, the thesis suggests further extensions to improve the quality of the game.

Lyngby, 17-July-2012

Bilal Arslan & Patrick Jørgensen

Acknowledgements

We would like to thank our advisor, Jakob Andreas Bærentzen for presenting this wonderful project to us, as it was our wish to be able to visualize and create an interactive game. Furthermore, we would like to thank him for helping us throughout the project with issues and guiding us through them.

We would like to thank the graphics group (Graphics Spring 2012) for many good suggestions to improve our project and the feedback from the main and other advisor's of the group.

We would like to thank the project group, Konrad Stanek and Steffen Angstmann, at the Research Center in Hvidovre Hospital for their great teamwork and understanding of the project and for the meetings throughout the project period.

Contents

Chapter 1

Introduction

The Danish Research Center for Magnetic Resonance in Hvidovre Hospital is trying to find a correlation between the neurofeedback and actions and decisions taken. To do this, they needed a simple car game, with an easily editable environment to use in their experiments. The game was needed to allow its subjects to take decisions such as whether or not they should turn and when they should turn and change direction of the car; in order to register their brain activity when they these decisions are made. Optimally, they want to predict the decisions of the subject, before he/she actually makes them.

This project involves creating the car game itself, which they will use to carry out their experiments. We have received specific requirements as to what the program should be able to do, and the most important of these requirements, is that everything is to be based on an XML file they generate. Therefore this report explains how we go from a simple XML file, to a full working car game, complete with an environment, with special regard to keeping the design and the program as flexible and extendible as possible.

We have implemented a program that automatically generates a complete environment including roads, tunnels and mountains, all of this, with requirements given by an XML file. As this is a Bachelor Project, limited time was available. We have therefore made our program following software engineering principles, making it as extendible and flexible as possible, so future work is not only

possible but also relatively simple. The most relevant and significant moment of the experiment is the one just before the subject is about to make a decision. At this point the user decides which action to perform, and then the measurements from an EEG (Electroencephalography, which is a measurement 'device' for brain activity) are recorded and a correlation is then searched for.

The report is divided into several chapters. First, we analyse the problem, define the requirements and relate to other work in the analysis section. Thereby, we explain how we designed the environment out from the analysis and the method of solving these different problems. Then we attempt to clarify how we actually did it, by explaining the algorithmic details of our solutions. Throughout the report, we present tests and their results, followed by a discussion of the project, where we among other things, explain many possible extensions of our program. Finally, we will hopefully be able to conclude that the work we have produced can indeed be used for future research experiments in the research of the neurologic field.

Chapter 2

Analysis

2.1 Overview

First and foremost, this project's goal is to assist the research project of IMM Cognitive Systems and Danish Research Center for Magnetic Resonance. Their request is an interactive virtual environment which goal is to help conducting experiments related to BCI (brain-machine interfacing) and real-time neurofeedback.

What is Neurofeedback? Neurofeedback is basically the signals sent from the user's brain scanned by the EEG (Electroencephalography) system. The experiments are done on the user, who is interacting on the virtual environment. The most important data that the experiments should reveal is the decision taking of the user. Their goal is to recognize and capture the phase of action, when the user takes the decision before it is executed. If it is possible to capture such a signal, then the ultimate goal would be to actually control the interactions done on the virtual environment.

The suggested simplest way of capturing such signals to experiment on an virtual environment is when the user interacts on a 3D car game. There are very few decisions to take and it is easy to differentiate between them (turning left, turning right, decelerate, stop etc.). The environment is also highly configurable

in terms of complexity. To increase complexity, one could add objects into the environment, distractions on the side of the road where the car is driving, or adding signs and warnings for the user.

2.2 Requirements

There are specific requirements from the research project, which the game should fulfill. We can separate the requirements into two categories. One being the game logic and control of the car. The other being the graphical part of the game, that is to say the generation of the environment and making it look good.

2.2.1 Game Logic

There are several things that the game should be able to handle. The game should be able to have an external control by an XML file. This includes

- Modifying parameters to set the details of the environment. This could be visibility (fog), shape of the track, light source, speed of car and other extensions.
- Control of the car (this is done through the XML file for our project)
  - input via keyboard.
  - Control of the car input via neurofeedback.
- Data logging. A finite dataset that logs various information such as car position, driving direction, distance to decision point etc.

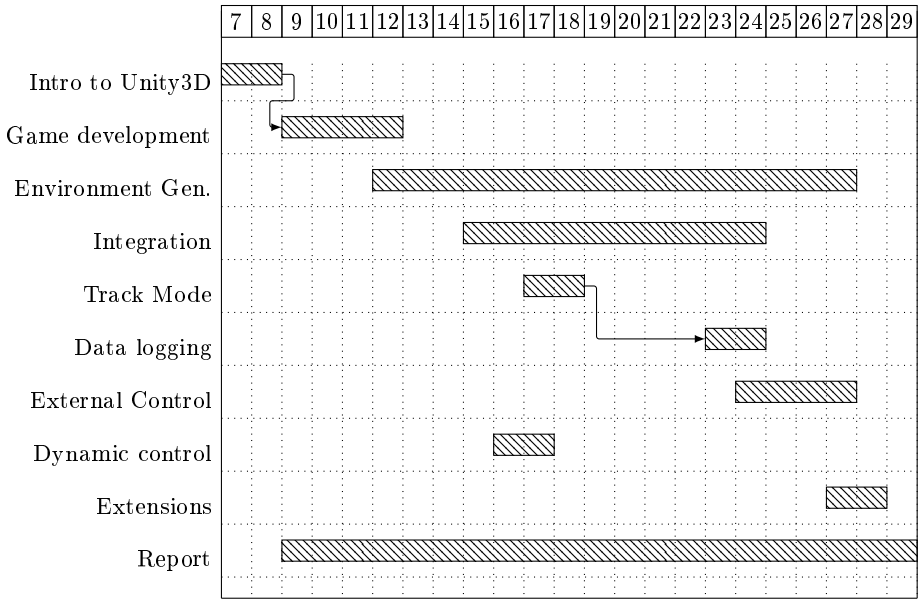
2.2.2 Graphics

The hospital's main interest doesn't lie in the graphics of the game. This doesn't mean that graphics are not important. In fact ignoring them does ruin the experience of the game. Therefore regarding graphics, we had to live up to our own expectations:

- **Textures**. The texture coordinates need to be correctly set, so the objects in the environment resemble the real objects.

- **Simple shaders**. This means that we need appropriate normals on the vertices that shades differently according to the contribution from the light source and other factors as well.
- **Visibility**. Visibility is essential when the user needs to take the decision, and the hospital wants to be in control over it. They want to control how much the subject can see, without affecting the coherence with the rest of the environment.

2.2.3 Project Plan



2.2.4 Related Work

There has been done a lot of other games before and especially car games. Since our project's essence is to create a car game, we can refer to many car games existing for PC, consoles and arcade games as well. To name a few examples: Grand Turismo [Dig97], a very successful car game created by Polyphony Digital to the game platform, Sony Playstation. Another successful car game released on the gaming platform, Microsoft's X-box, namely Forza Motorsport [TS05]. Other popular titles that comes up to mind: Need for Speed, Colin McRally, Formula One etc.

Although, all these titles are made for entertainment purposes and commonly are racing games. Our car simulation has a whole different purpose: to assist in Neurofeedback research. Similar works done with EEG by "Serious" games [QWN10] show that an increasing need of games in medical applications and how we can design them to help medical research. Furthermore, for monitoring pain managements, "serious" games were used to help doctor's understanding the anesthesiology and psychology aspects of patients. [SO11]

There are some involving algorithm's for solving the different problems in our project worth mentioning. One of these is a method for creating roads on the terrain. A project group working with Unity called SixTimesNothing ([Mor12] has created a Road and path tool, which creates a road given a terrain field and some path (spline) coordinates, defined by a few mouse clicks on the scene.

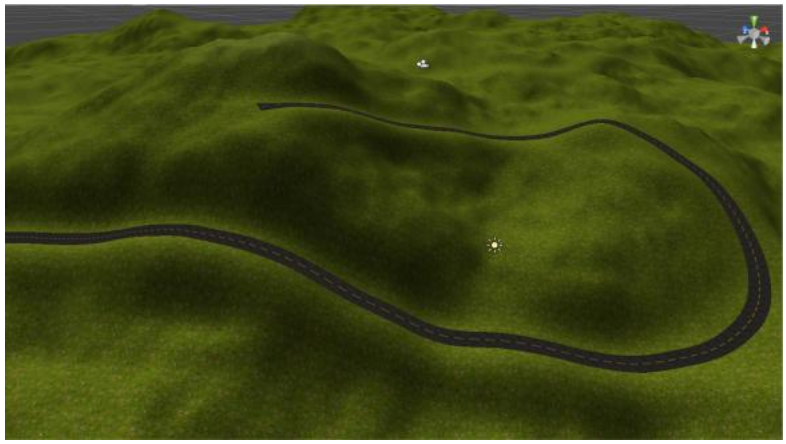


Figure 2.1: Road and Path tool by SixTimesNothing [Mor12]

Unity development team and company has also published a tutorial for a car

game, which helps other Unity game developers to get started with creating car games in unity. It is freely available for download in their homepage. From this one we have taken the 3D car model, and edited the scripts attached to it.

Figure 2.2: Car tutorial for Unity

Unity has also been used to create an artificial virtual environment to people with psychological condition like depression. [MHDB11]. In fact, there is a company working with developing serious games for EEG purposes.

Chapter 3

Method

3.1 Overview

As mentioned in the previous chapter, the game is to be used to conduct experiments at a hospital using an EEG to read the signals sent out from the user's brain. The user interacts with the game by using an input device. To start with, this input device is a simple joystick/keyboard but on a longer perspective, when the experiments have proven successful, the idea is to send the input signal to the game using the user's brain signals. This signal is analysed and converted into an input to the car game (all of this is done by the hospital).

All of the input and output from and to the car game is controlled by XML files and text files. We have three different of these.

Environment.xml This file is used when we are generating the scene from a set of spline coordinates for the road and the tunnels.

Logging.txt This file is used to log specific information about the environment and the location of the car.

Input.txt This file is read from the game to get the input variables to control the car's behaviour.

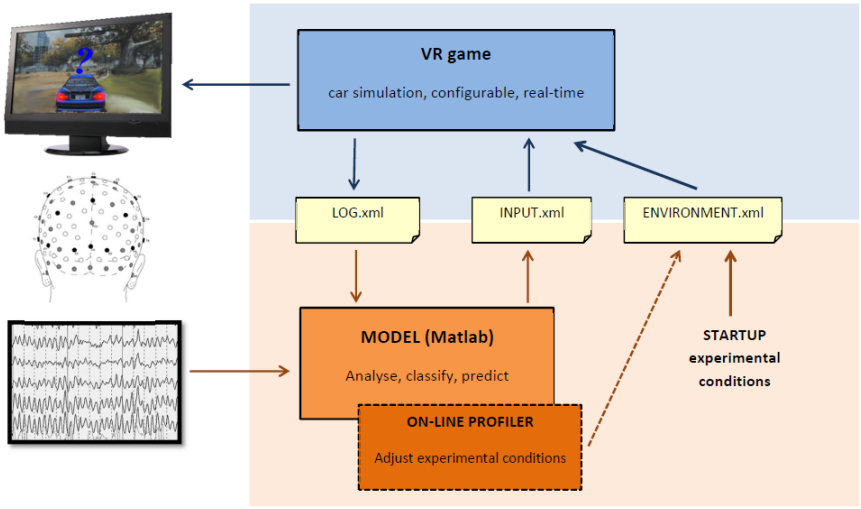


Figure 3.1: Overall design scheme (picture taken from Konrad Stanek's presentation)

3.2 Game development

The initial idea of the project was to generate the environment in OpenGL and C++ code and then, somehow, integrate this environment with the game development engine, Unity. Unity is a wide-known tool used to create decent, sophisticated games and is able export games as playable on PC as well as modern consoles today (Nintendo Wii, Xbox 360 and Playstation 3). [Uni12j]

When we needed to design our environment in a C++ environment, there were several things that needed to be taken care of. First, we had to think about, how we were going to translate the work done in C++ into a format Unity can understand. Secondly, it was not enough to just translate the work from C++ to Unity, it also needed to be user-friendly as the project group from the hospital is going to use our program for their experiments, and is not supposed to to meddle with any code. Thirdly, we needed to think about performance and esthetics.

After some research, we found an interesting file format that fulfilled all of our requirements for the translation of the environment from OpenGL C++ code to Unity, namely the Object (`.obj`) file format. This one suited us, as it was easy to make (a plain text file with a *.obj* file extension), and Unity is able to

read them.

3.2.1 Object file

When different models are designed in computer graphics, they are represented by a set of vertices, normals, faces and other information. The `.obj` file format is exactly used for this reason, to represent the polygonal data in pure ASCII form. [Fil12]

The file is organized in such a way that only pure vertex data can be written and the elements (usually a face) choose their vertices, by designating the numbers of the vertices, defined in the file, they need. For example, say we have 4 vertices defined in the `.obj` file by 4 lines, namely

```
v 0.0 0.0 0.0
v 0.0 1.0 0.0
v 1.0 0.0 0.0
v 1.0 1.0 0.0
```

Then we can specify a face by defining 3 vertices (a triangle) with the numbers of which they appear in the file. Using the example above, a triangle face can be defined by `f 1 2 3`, which means that vertex (0,0,0), (0,1,0) and (1,0,0) is taken to form a triangle. If it was `f 1 4 3`, a triangle would be formed by vertices (0,0,0), (1,1,0) and (1,0,0).

Faces can hold more than just vertex information. They can hold normals and texture coordinates for every vertex as well. There are other interesting elements, groupings and other rendering attributes, which can be found on [Fil12], but in this project, we stick to only using simple vertex data and texture coordinates, and have faces as elements.

3.2.2 Unity

It is not enough to build an entire environment, we need to build a game out of it as well. This is what we use Unity for. It helps, and makes it easier and faster for us to make the game look nice. In Unity, most of the things for rendering the objects are done automatically such as rendering settings, shading, ambient occlusion, light-mapping, keyboard control, camera view, normals calculation and much more. The interactivity and the game mechanics can be controlled

by simple JavaScript and C# scripts. These, we have revised and modified, in order to suit our game best.

3.2.3 Car tutorial and the car

To start with, we had an empty environment and as a starting point, we decided to take some inspiration from other car games. Luckily, the Unity company is sharing tutorials regarding game development and how to get started and even more luckyly, they had a car game tutorial. After throughout investigation of the tutorial and analysing the in-depth code and how the game was working, we noticed that making an object (the car) and making it move accordingly to the laws of physics as it would in real-life requires insight in many things.

Our project is about creating environments and to make a game which is suitable for experiments. Given the time restriction of our project and the amount of details that the game should include, it was decided to take the car model asset from the Unity tutorial and 'borrow' it for our game, as it was also suggested in the tutorial notes by the Unity company.

3.2.4 Terrain and road creation

Having the car, allowed us direct movement towards environment creation. We started by creating a simple environment, using Unity's Terrain function, which was an easy and fast way to create an environment. Next easy step was to create a road, and we used a road creation tool to get inspired by other people's work. [Mor12] This was the first step in leading us to the idea for road creation (see section 3.5 and figure 2.1).

3.2.5 Integration with object files

When we had created the road and the environment using OpenGL and C++, using our algorithm's for road creation and terrain creation (3.4), we integrated this with the object file format. All the vertices we have created and combined in OpenGL, we have added to the object files. Unity has the ability to load object files and it does so dynamically too, which means that if you make changes to the object file outside Unity, it appends the change in Unity as well. This meant great flexibility for the combination of our OpenGL code and Unity environment.

Unity also has the ability to add mesh colliders to the surface as part of its import settings, so that every other object in motion will collide with the environment. On top of that it also calculates the normals, if asked to by the import settings. In our case, this makes the car possible to drive on the surface, where it would fall through the mesh otherwise. All that was needed to be given was the object file.

3.2.6 XML parsing

All the parameters and spline coordinates for the road and terrain are generated through XML, which is why we need XML parsing, in order to read the data. We were advised to use the GEL library by IMM and our advisor [Bæ12] as an easy and simple way to read from XML files. Javascript also contains an XML parser, namely Microsoft's System.xml Namespace. [Mic12] This namespace contains an XmlReader, which we can use to easily read the data in javascript as well, since this is the main scripting language that Unity makes use of. Another alternative is C#.

The hospital's XML file consist first of some parameters for general environment purposes such as fog, size etc, followed by the road map, consisting of roads, side roads and segments. Generally, we have the following structure.

```
<parameters>
   <car_view>outside</car_view><!--inside|back|top-->
   <car_default_speed>50</car_default_speed><!--0-100-->
    ...
</parameters>
<road_map>
   <scene id="1" pid="0">
      <road id="1" pid="0">
         <segment id="1" type="tunnel">
            <x1>52.654</x1>
            <y1>4.6569</y1>
             ...
         </segment>
      </road>
   </scene>
</road_map>
```

Because of the simplistic structure, we can easily read from the XML by just parsing through the DOM elements and read its body.

3.2.7 Light and Shading

Without some light contribution to the scene, everything is dark and nothing can be seen. This is why the game also needed some light sources, which would be to simulate a sun. When making an environment a sun is essential game. This one would typically be modeled as a directional light, where the position of the sun is far away from the scene.

With some light, there needs to be some light contribution for each object on the field as well. Since every object in the environment is separated, one could shade every object differently according to the material. One material could have more ambient contribution and another could have more specular etc. Unity has many built-in simple shaders, which mainly consists of the diffuse term.

3.2.8 Procedural Modelling

The idea of procedural modelling is to model objects by a set of rules, rather than design each object in a game manually. In fact, for standard objects, such as trees, buildings and in our case terrains, it is acceptable, and in many cases, favourable not to design those by hand, but rather have algorithms take care of generation. This not only spares one of a lot of time modelling each object, it also saves memory, as one is not to store huge files of 3D objects, but rather rules that define them, and the algorithm that defines them. From the point of view of a company, it is also very interesting having procedural generated objects in a game, as it saves oneself from paying an artist to model a detail, which is not trivial to the gaming experience, and often simple enough to be looking just as well, if generated procedurally. In fact [PM01] creates entire cities procedurally from statistical and geographical input data. And this project is about creating an environment, and in fact a complete self driving game, from an XML file, so we think it classifies as a procedural modelling project, where a lot of work has previously been done.

However unlike many games, ours has not the goal of being fun, but rather to be able to play itself. So we do not procedurally generate just the environment, but also the driving itself, which has to adapt according to the terrains created.

3.3 Bézier

3.3.1 Theory

A bézier curve is a polynomial curve formed by using few spline coordinates for which it 'aligns' to. These coordinates are more commonly referred to as *control points*.

More specifically, we have used cubic bézier curves. These are only defined by 4 control points and the ends of the curve are joined with the start and the end points, respectively. To give a formal description of how the bézier curve is formed, the formula is given below:

$$P(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3, t \in [0,1]$$

where P_0, P_1, P_2, P_3 are the control points. The coefficients are related to the Bernstein polynomials, which is polynomials in Bernstein's form and is an important mathematical step and the building blocks of a bezier curve. See [Joy00] for further explanation of Bernstein. Below you will find some examples of bézier curves.

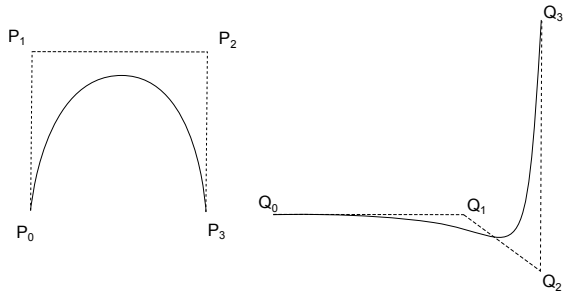


Figure 3.2: Examples of bézier curves.

3.3.2 Practice

We are using the bézier curves in two places, namely the road creation and the tunnel creation. A straight lined road would look dull in a scene and it not common to have only straight roads in real life as well. Therefore, the road

needs to be curved and these can be controlled by the control points, defined by the xml file.

Figure 3.3: Bezier segments. Many road segments are joined together as the control points of the end and start are the same.

A tunnel on the other hand can be straight and in some cases, this is true in real life as well however, the tunnel's functionality is to obscure the user's view so he/she cannot see the end of the tunnel. Therefore, it might be convenient if the tunnel is curved so the end of the tunnel cannot be seen (this is assuming that we have no fog in the scene).

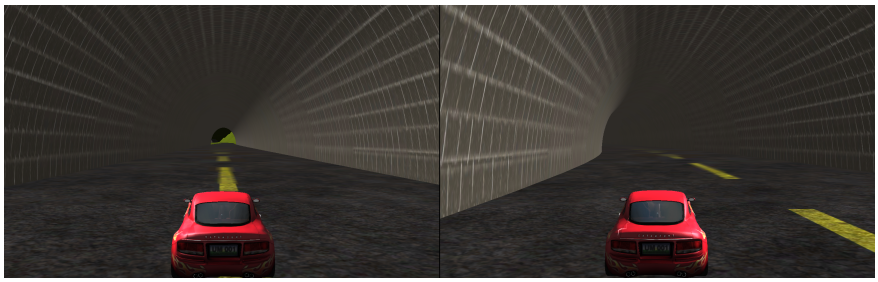


Figure 3.4: Obscured tunnel. To the left, straight tunnel where the end is visible. To the right, we have curved tunnel to hide the end of the tunnel.

Apart from the environment creation, bézier curves are also used to control the car by way points. Way points are explained in 3.8.1.

3.4 Terrain

At the very beginning of the environment creation, we start by creating a terrain which is flat to begin with. The terrain itself is constructed by a set of vertices and where the faces of the triangles are set. A face is basically constructed by a set of 3 vertices and additional information, such as normals and texture coordinates. More formally, the relation between the vertices, edges and faces of triangle meshes, are given by the Euler-Poincaré formula [TAM08a], which is given by

$$v - e + f + 2g = 2$$

where v is the number of vertices on the mesh, e the number of edges, f the number of faces and finally, g the genus (a genus is basically a hole in the mesh).

3.4.1 Height

To make the terrain look more realistic, we need to add heights to it in different fields. For this purpose, we have used a noise-based procedurally generated heights. [TAM08b] [DSEW02] This noise function is a pseudo-random generation of the height, meaning that heights are calculated randomly, but the same random value is produced for every execution of the noise function.

For calculating the noise, we use Sum-of-Sines where we for each randomly generated point in plane, greater and smaller than some defined thresholds, create a vector from the center of the vertex to the points, which is the frequency vectors. With these frequency vectors, we take the sum of the sines of the vector's dot product with the current vertex. Formally, the equation is given as follows.

$$\sum sin(v \cdot f)$$

where v is the vertex we are looking at and f the calculated frequency vector.

A noise texture is sent out from the function and we combine this function with another function, called turbulence, which gives an irregular smoothing out of the terrain, which contributes in making the scene look more realistic. For every vertex, the effects of these two functions are combined and used to give the terrain a nice and realistic look. The turbulence function is given by the following formula.

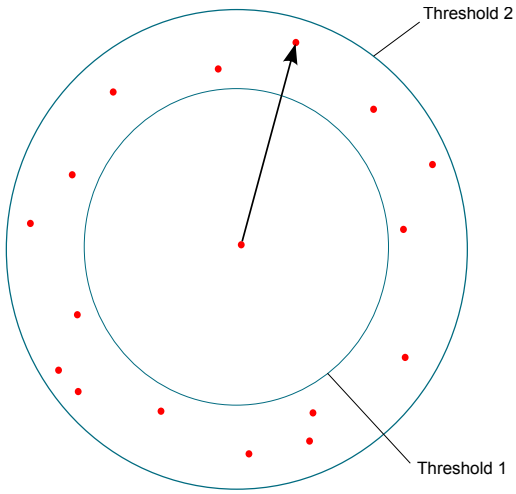


Figure 3.5: Noise frequencies. Black dot is the vertex we are looking at, red dots are the randomly calculated values, which are in between the two defined thresholds. The random values outside the threshold are omitted.

$$\sum_i |\frac{1}{2^i} noise(2^i x)|$$

3.5 Road

Once the terrain is set, we need a road on which the car can drive upon. The road has a different shape and texture than the terrain. Also, the road is not linear but instead is shaped along a curve - a bézier curve to be exact (further explained in 3.3).

Once we have the curve defining the road, we need to actually create the mesh. There were different ideas for the mesh generation. One of the first ideas was simply to translate the curve with the half road width to both sides and produce a triangle strip along these vertices. Although, we noticed quickly that the idea did not hold for several cases. For example, if we had a very curved areas, the width of the road would not maintain the same along the road. Another idea was to introduce half circles along the curved areas and scale them to have

different radius. This way, we would have the same road width along the curved areas, although this required partitioning of the road into smaller segments and seemed rather complicated.

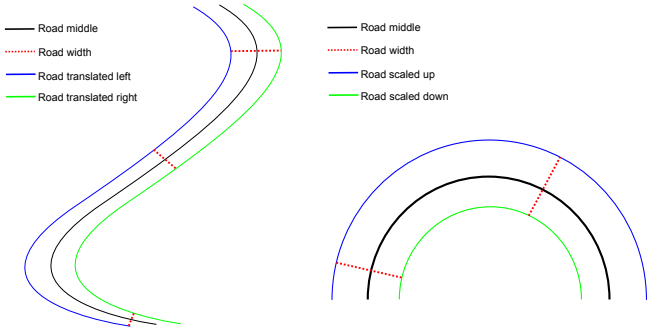


Figure 3.6: To the left, initial idea with translating. As we can see, the road width's are not the same along the road. To the right, scaled half circles to maintain the road width, but too complicated as it requires segment splitting and not so flexible.

The solution we use is quite different. By simply knowing the vertices along the curve, we can create vectors, which is directed toward the next vertex on the curve. By having this information stored, we produce vectors and from each vector start point, we create two new vertices going on the vertexes left and right side, by half the road width. These two vertices, we later use to produce our triangle strip along the road. See illustration in figure 3.7.

3.6 Tunnel

To create tunnels in our program, we had to build on the work we had already done on the roads, seeing as the tunnels are covering the roads, and those are defined by bezier points. The idea of the tunnel, is having some sort of transition between scenes, where the player does not notice that the environment outside changes 3.8.4.

The challenging part of this is to make the arch touch exactly the road, so there is no gap between the road and the arch of the tunnel. Therefore we needed to have a function, which starts and ends in the vertices that define the road.

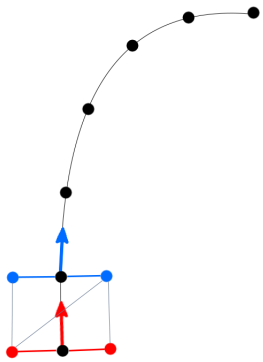


Figure 3.7: Road creation. For every vertex, we find the direction vector and take half the width out from each side, perpendicular to the direction vector.

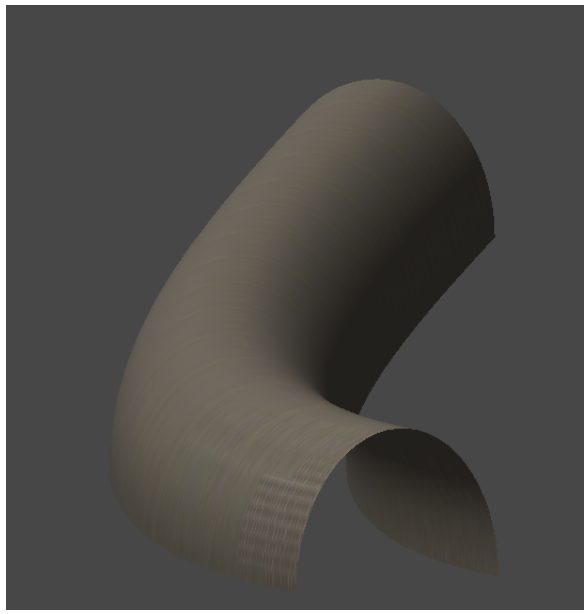


Figure 3.8: Example of tunnel.

3.6.1 The arch

As shown in 3.8 and 3.9, we have decided to make our arch round. However we could have made the tunnel arch be squared, all that would need, would be to raise the two road vertices to a higher height, and then you would have a tunnel. In fact this would simplify the implementation of the tunnel a lot, and it would strongly reduce the amount of vertices used for the tunnel. However we found that, not many tunnels are squared, and to increse realism, we would make them round. We could also combine both solutions, so raising a little, then finishing off with a rounded arch. This might look more realistic, however we found that the round arch looked sufficiently realistic, and in fact, there is not too many extra vertices.

3.6.2 The road and the tunnel

There was discussion, as to whether or not the tunnel should be united with the road, as to whether they should share vertices. We did not think of it as a good idea, as it increased complicity of the implementation, and lowered flexibility in case one was interested in editting the implementation of the tunnel.

Figure 3.9: Example of tunnel with road in game

3.7 Mountain

After having created the tunnels, we needed them to make sense. Therefore we needed to make mountains over each tunnel. We have defined it, so the mountain has a certain width. However this one remains small, as we want it to have relation with the road width, so we can multiply this value with the width of the road, in order to keep it bigger than the wider than the road at all times.

3.7.1 Integration with terrain

There was discussion, seeing as we enter a tunnel, and the road reaches the border of the tunnel, whether or not the mountains should just be the terrain with a raised height. We would then have to make sure, that at places where there is a tunnel, we would add new vertices.

We would like to add mountain vertices at the entrance and exit of the tunnel, to make sure there is no gap between the tunnel and the mountain, so it looks as if the tunnel has been created by drilling the mountain. At all other places, we would just like to add additional height to the terrain so it shoots up in the air.

However we found that adding extra vertices in the terrain, will complicate the creation of the terrain. Therefore, decided to make mountains independable of the terrain, but based on the implementation of the tunnel.

The idea is making a carpet over the tunnel, that connects to both the tunnel and the terrain. This means, we would like to make the same as described above, only as an independent object.

3.8 Car control

In this section, we will talk about how the various controls and behaviours of the car are designed. First, we will talk about way points, which plays an essential role in follow-track mode of the car and many other things as well. Then we will talk about the follow-track mode in detail and what it actually is. Furthermore, we will talk about how fog is involved in our scene and how it is handled. Moreover, we will talk about how the car is translated between scenes and after, we will discuss how the input and output from the car is handled and how we control the car, when a decision needs to be taken at a certain point. Lastly, logging is explained.

3.8.1 Way points

We have made way points, to guide the car through the scenes. However they are not necessary, as a lot of calculations could have made the car drive by itself. However making way points spares a lot of calculations, that are only needed to be done once, upon startup, and therefore sparing the car for many calculations, and thereby speeding up the game.

3.8.1.1 Basic

A way point in general can represent many things in many different concepts, sometimes used in GPS systems to indicate crossroads and turns, but in our case, a way point is simply a position/location on the road, which helps us to indicate and decide different events when such a way point is reached. The way point does not have to be represented in any polygonal mesh, but a simple set of coordinates or a point (dot).

The way points are placed on the road and generated through the same bézier curve as the road. Thereby, we ensure that the way points lie in the same place as the road (see Figure 3.10).

The general usefulness of way points in our project is to control the car's behaviour according to its speed and steering. What we mainly want to achieve is to make the car follow the aligned way points on the road and thereby guide the car throughout the scene. This is the method we use to implement follow-track mode, which is the mode where the car follows the way points and thereby drives automatically.

The other great usefulness of way points is to trigger special events when their location is reached. This can help us to take a decision regarding the car movement and it's destination. This can further be explained in the section for decision taking. 3.8.5.

Figure 3.10: Way points on road. In this scene, they are simply formed as spheres, but no mesh is drawn to hide their existence to the user.

3.8.1.2 Extended use of way points

Now that we have fixed points in the environment, we have an easy way to send any information to the car. As the way points are used to make the car drive, each way point can have a big amount of information, that the car doesn't need to calculate. It can hold a lot of information that the environment has, and the car wishes to retrieve. The way points can hold information about where they are, which scene has created them, how far there is to the next segment, whether or not the car should turn, which mode the car should be on (follow-track or free-ride) and a lot more. Mostly the way points hold the information mentioned above, which the car uses for its logging. So it contains information about which road we are on, which scene it is and where the car is in local scene space.

All in all, the way points is a good way to pass information from the world, the scene, and the way point to the car. In fact it is the direct connection between the car and the xml file. The way points help in many ways, and according to new needs in the game, one could use the way points to solve a lot of problems.

3.8.2 Fog

We also use the way points to control the fog. When the car reaches specific way points, information is passed to the car, that the fog can change. The hospital needs to blur the exit of the tunnel, so the player cannot see outside of the tunnel, when being inside. By testing we found that having greyish whitish fog, gave the "light inside the tunnel" sensation 3.11. So we decided to use the built in Unity `fog` [Uni12c], and simply control the density, and the color. However, the idea 3.8.4, is that the player does not notice changes of scenes. Therefore we have a demand that the tunnel, have a minimum length (around 200 meters), as if it is shorter, the fog would not cover the environment outside the tunnel.

Figure 3.11: Fog inside the tunnel

Figure 3.12: Fog on the environment

When on the environment, we wanted a feint, blueish fog 3.12, that one barely notices (the ozone's scattering of light). Seeing as we are editing the density and the color, we need a transition in between, for it not to be too noticeable. To control this color and density, we could have had the car decide how far it is from the tunnels, and from exiting the tunnels (when inside). However seeing as we already decided to control the car's follow-track from the way points, we decided to make them give the message for the car to make a transition. In fact there was question as to when the message should be passed. As one could decide to do it according to distance, segment or time until the car reaches the tunnel (likewise when exiting).

However seeing as each segment knows and reaches the last spline coordinate 3.3, we decided just to use the way points on the last segment to send messages to the car about changing the fog. One could just let the way points change the fog density and color, however as the way points are not equally distanced, nor do the segments the same length, fog transitions would never be the same. Also if the way points defined the fog settings directly, the fog would jump form one value to another, and it would not be smooth. One could also have fixed small values, that one adds, for every update. This would indeed make it smooth, however we wish the fog to finish settings upon reaching the tunnel at a certain point. Also if one adds a fixed value to the fog density, one also depends on the frame rate of the computer, as the settings are set in Unity's overridable

function `Update()` which is called for every frame [Uni12f]. One could instead use Unity's overridable function `FixedUpdate()`. As this one is called at a fixed time, the fog would be the same for every computer, however one would have a lot of unnecessary calculations (or not enough, depending on how performant the computer is).

We decided though to use `Update()`. We make the fog come increasingly according to how far the car is from the last way point of the segment before entering the tunnel. Thereby we can easily control when the fog starts to change, and how fast the transition happens. This also make most of our fog transitions look the same. However not all, as if the straight line distance from the car to the last way point on the last segment, is shorter than the distance set, we have the fog suddenly appearing.

On the way out of the tunnel however, we are not specifically interested in having a distance in which the fog changes, but we still wish it to be the same, no matter the computer we are working on. Therefore the car looks for when it is reaching the last way point of the tunnel, and we define a fixed time, in which the car should have changed from tunnel settings to environment settings.

3.8.3 Follow-track Mode

A very special feature of the program is the follow track mode. The main focus of the experiment performed by the hospital is the decision taking (whether or not I should turn or not), and not so much the driving. Therefore the car should be able to drive by itself and follow the road created by the spline coordinates given by the XML file. Therefore we use the XML once more to generate the way points, and those are used to make the car drive, using the information stored by the way points to guide the car.

The car could also not use the waypoints to drive, in fact it could just read the XML file itself, and calculate its track from that. However we decided that the way points should hold the information about the path, because it saves a lot of calculations (as it is only made once on startup), and allows us to partition each segment into several points, that the car has to reach, allowing more or less smooth driving.

3.8.4 Transition between scenes

In order to maximize flexibility of the program, and to maximize use of the spline coordinates, the hospital have created small scenes, that the car should traverse. This enquire new design questions, as to how make the transition in the best possible way.

The problem has been divided into several steps:

1. How should we import the scenes, created by our C++ program?
2. Upon start-up, when should each scene be loaded, and where, and thereby how should the car go from one scene to another?

The first problem is one we have invested a lot time into. We tried to find out how Unity can load an entire folder independently of how many files are inside, and also load them into the game. However we found that Unity loads the folder correctly, but it wasn't obvious how to make dynamical import from a script. Therefore we ended up deciding to make a fixed number of 150 scenes from our C++ program, and load all of them into Unity, and into the game every time, which is done by Unity's import settings. However in the case (most of the cases), where the XML file contains less scenes than 150, the C++ program create all the scenes necessary, and the rest are just empty. Making it this way, the empty scenes do not eat up computation nor memory, so we found it an easy and good solution.

The second problem is less trivial. It all depends on the way one wishes to interpret the problem. In order to make the transition more smooth, the hospital have made it so, that the first segment of every scene, and the last segment (no matter the decision) is a tunnel. So the player will not notice a shock when changing from one scene to another, especially seeing as we have created fog to blur the exit of each tunnel 4.6.1.2.

A more or less easy, however not very efficient solution (in terms of calculations and memory), could be to duplicate the next scenes into each exit of every previous scene. However very fast, one would get a lot of unnecessary scenes, seeing as one would have:

$$u_n = k^{n-1} + u_{n-1} \text{ scenes}$$

where $u_0 = 0$, where n is the number of scenes and k is the fixed number of exits per scene. On the other hand one could also dynamically delete the scenes

that are unreachable. However this would mean that we should have another way of checking each decision the car takes, and delete all the scenes that are unreachable, or have the car do it, thereby adding extra computation to the car upon decision making.

Another solution to this problem could be just to keep one prefab[1], of each scene, and just move it, according to the decision taken by the car. This would help the unnecessary calculations on start-up, however the game would still have to move the scene to the exact right location, so a script would have to run and control this. Also a problem could be, as we have squared terrains (for simplicity), if a tunnel is not at the border of a terrain, part of the previous terrain, could affect the next scene, so one should also delete the previous scene, when entering the next.

Both solutions are dependent on having the exit tunnels of one scene, being identical to the entrance tunnel of the next scene. In fact the solution we chose 3.13, is also dependent on this. We decided to simply translate each scene (on start-up) to be at the width of the mountain[2] plus 50 away from the end of the previous scene in the z-coordinate. However this means, that we need to 'teleport' the car from one scene to another.

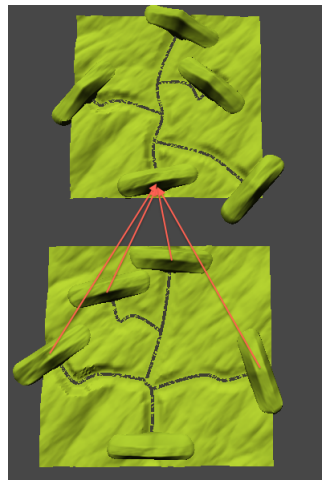


Figure 3.13: Illustration of the car's teleportation

[1] Prefab are an instance of an object inside the project [Uni12g]
[2] width of the mountain is defined by `mountain_width` multiplied by the road width 3.7

3.8.5 Input and decision

In the beginning, we need to control the car by some input from the keyboard and later on extend this input to be from reading a simple text file. To start with, the Car asset[3] has got predefined keyboard control for steering the car and for acceleration and braking of the car. This, we need to edit to our own way of controlling the car.

The simple design is that the hospital has got a program for writing input to a text file. This input is done by a scales of milliseconds, so it is important that our program reads this input fast as well. We simply need to read this input and use it to control the car's behaviour.

The problem here arises when we need to be careful with the sharing violations with the file. One process, which is the hospitals program, needs to *only* write to the file and our process in Unity script *only* reads from the file. More detailed, our program keeps this file open at all time and when input is received, this input is read immediately and removed from the file and await for the next input.

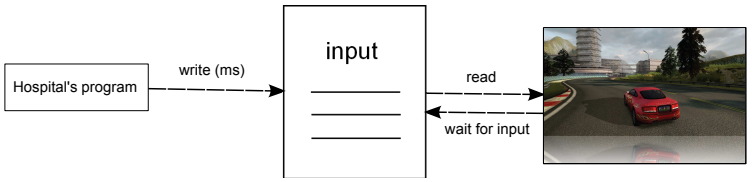


Figure 3.14: Input and read/write scheme. Hospital's program sends input to a text-file and the car game waits for the input and then reads the input from the text file.

We can have five different input

- **start** to start the game.
- **left** to turn left.
- **right** to turn right.
- **up** to accelerate.
- **down** to brake.

[3]Car and the car control is developed by Unity

When the input is read, we have to possibilities depending on the mode we are driving.

1. If we are driving in free-mode, we take any input directly from the text file and the action is made.
2. If we are driving in follow-track mode, we only need to take the decision at the cross roads or where there is a possibility of turning in a direction. In this case, we simply accept input from only a certain time-interval from the actual turning point. This can for example be to accept input only from the start of the segment road and 5 seconds onward. If the input is received after 5 seconds has passed from the start of segment, we reject the input and use the last input received before 5 seconds, if any.

Furthermore, in the follow-track mode, the decisions are controlled by the way points. At every cross-road, we create a special way point which triggers the event of checking the input. If no input has been given, we simply continue.

3.9 Logging

When doing such time sensitive experimenting, logging of the game's state, car position and other information is quite important when the hospital backtracks the events at certain times. This is also very important in the decision taking phase, to see a correlation between the brain signals and the log data. Therefore, careful and precise logging is essential to the experiments.

Unity engine contains a very nice solution for this time sensitivity. It is called `fixedUpdate()`. [Uni12b] What `fixedUpdate()` does is that it runs a certain portion of the script in a fixed time step interval, meaning that we can log at any time frequency as the user wants. For the hospital, every milliseconds counts, and therefore this functionality prooves to be very useful.

The format of the log is provided to be as the following:

id	gt	st	sid	rid	spid	posx	posy	orien	speed
0	497	0	1	0	1	51.77	6.72	(−1.0, 0.0, −0.1)	0
1	768	15	1	0	1	51.77	6.72	(−0.2, 0.0, 1.0)	0.0588622
2	1008	30	1	0	1	51.77	6.72	(−0.2, 0.0, 1.0)	0.1632052
3	1008	44	1	0	1	51.77	6.73	(−0.2, 0.0, 1.0)	0.1632052
4	1008	60	1	0	1	51.77	6.73	(−0.2, 0.0, 1.0)	0.1632052
5	1008	74	1	0	1	51.77	6.73	(−0.2, 0.0, 1.0)	0.1632052
6	1009	89	1	0	1	51.77	6.73	(−0.2, 0.0, 1.0)	0.1632052

To describe the columns, `id` is log id, `gt` is the global time in milliseconds(ms), `st` is the scene time in ms (reset every teleport), `sid` the scene id, `rid` the road id, `spid` the spline or segment id, `posx` and `posy` is the position of the car in x and y coordinates, respectively, `orien` the orientation in normalized vector form and lastly `speed` the speed of the car.

Chapter 4

Implementation

In the implementation chapter, we will thoroughly explain how we actually implemented and created our environment with additional information about the game development in Unity game engine. We will go though how we applied the bézier algorithm to create the road, tunnels, mountain and we will explain how we generated the terrain. Furthermore, we will explain how we implemented artificial intelligence to our car (follow-track mode) and how we made logging along with input and output to take decision of the car's behaviour.

4.1 Terrain

The implementation of the terrain is pretty straightforward, with the exception of the height. Without the height, it is just a matter of running a double for-loop with the width and height of the terrain, respectively and creating vertices with these coordinates. The number of steps you take can be adjusted with how detailed the terrain is drawn.

For the height, as mentioned in the design section 3.4, we use a combined noise and turbulence function. In the noise function, we create an array of frequencies within a threshold, by using a random function in the GEL library called

`gel_rand` [1] [Bæ12] and we create a vector out from random values. Then, if the length of the vector is greater than the threshold (0.5 in our case), we add this vector to the frequency vectors, else we discard it and try again. When the frequencies are generated, we simply sum up the results, using the Sum-of-Sines formula. For turbulence, we simply call this noise function a number of times to generate turbulence.

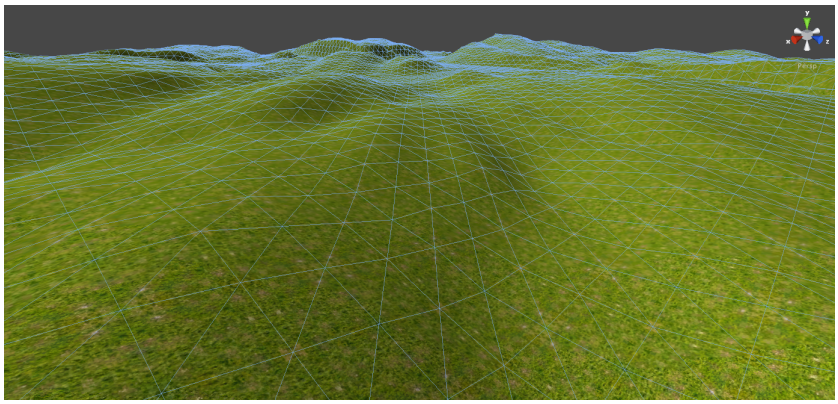


Figure 4.1: Example of the generated terrain using noise and turbulence.

However, as shown on figures 4.3 and 4.2, we want the road to be on level $y = 0$. Therefore, upon making the random height of the terrain, we need to check whether there is a road or not. In fact instead of calling the random function directly, we have made a function which not only sets the height, but also checks if there is a road on the terrain. The way it does that, is as follows: While calling our bezier function, we set values to an array, initialized (with 0 in all fields) to have the terrain width multiplied by the terrain height divided by the terrain step as size.

$$init = \frac{terrain_w \cdot terrain_h}{terran_{step}}$$

In that way, we have a field in the array for each of the vertices in the terrain. When creating the roads, the bezier function sets the accurate fields in the array to having value -1. Also on the sides of the road, we want a transition (as shown on 4.2), meaning we want the terrain to reduce the amplification final height of the terrain on the sides of the road. Therefore, in our array we set the surrounding fields of the array to have values $\in [0:1]$. Eventually, when calling the terrain height function, it will retrieve values from the array. If the value is

[1] The function is not developed by us, but by our advisor Jakob Andreas Bærentzen. The reason we used this function in this particular noise function is in connection with the course Real-Time Graphics, where we also used this particular random generator to generate heights in the terrain.

−1 it will not edit the height, setting it to 0. Else it will use the array's values (other than 0) to multiply on the height returned by the turbulence function, thereby making it smaller.

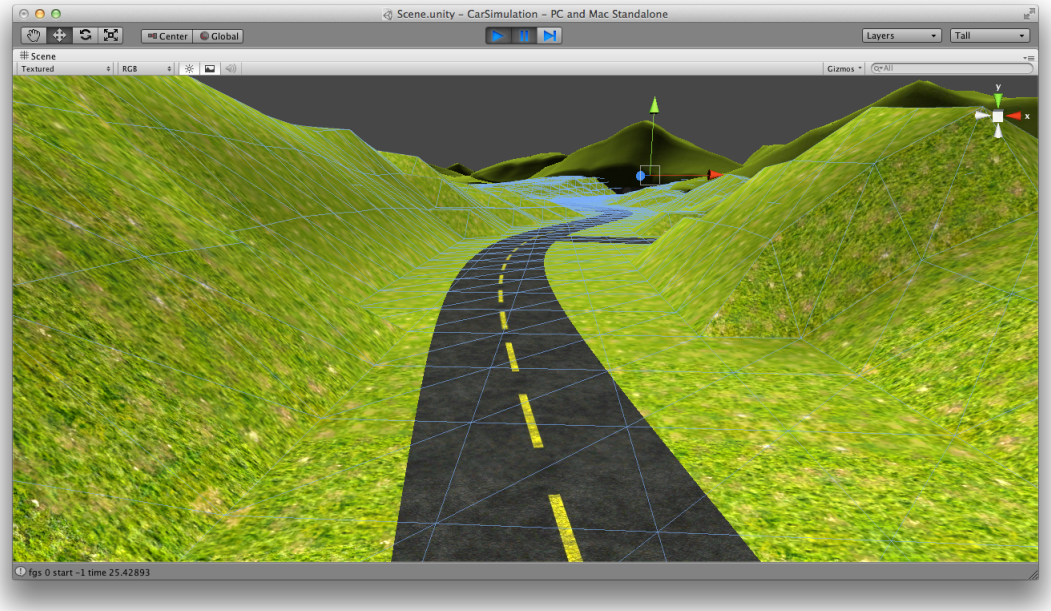


Figure 4.2: Example of the generated terrain using noise and turbulence.

4.2 Road

As mentioned in the design section, we use the last mentioned idea for implementing the road. [3.7] The road is created by the global function we have called *bezier()*. As the bezier formula given to us by A, the curve is created by using a variable $t \in [0; 1]$. Therefore only approximations can be used in our program. In fact we cut up our bezier curve in a certain amount (named `bezier_steps`) of pieces. For each step we create two new coordinates `nP0x` and `nP0y`. Next, we create next set of coordinates by taking the next step. Those we name `nP1x` and `nP1y`. Now, these two set of coordinates are used to create a directing vector from `P0` to `P1` called `vec`.

What we want to do is to create triangle strips, where we combine vertices translated with the road width, so to have the road width contribution to the mesh we want to create, we create a new vector with the road width contribution by the following (where rw = road width)

$$vec_{rw} = \frac{vec}{\|vec\|} * rw$$

4.2.1 Vertices

When we create the vertex for the road, we simply translate with `vec_rw` from x and z position (in our case, z because Unity uses xz-plane, instead of the xy-plane) respectively. So the new vertices are given as

$$v_1 = \begin{pmatrix} nP0x - vec_{rw}.x \\ 0 \\ nP0y + vec_{rw}.y \end{pmatrix} \qquad v_2 = \begin{pmatrix} nP0x + vec_{rw}.x \\ 0 \\ nP0y - vec_{rw}.y \end{pmatrix}$$

The y vertex is 0 because the hospital wants the car to drive at the plane where $z = 0$, which is $y = 0$ in our case.

4.2.2 Texture

Mapping the texture on the road was not trivial, as simply taking constants was not sufficient. In fact as all segments are not the same length, we would not have the same size of texture as we move along the road. Therefore we have defined a

`tex_index` for each x-coordinate, on which we add the distance between `P0` and `P1` divided by a constant. The y-coordinate of the texture mapping however was a lot simpler, as we just needed used the index 0 on one side of `P0` and 1 on the other side.

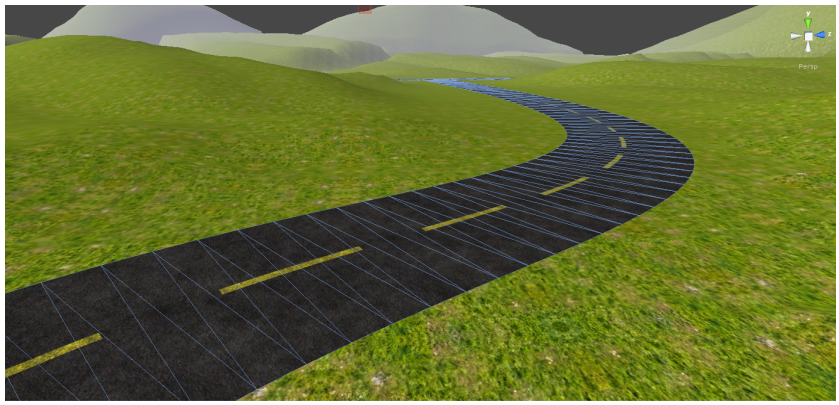


Figure 4.3: Road vertices and texture mapping.

4.3 Tunnel

We have implemented the tunnels by using the road vertices implementation *nP0x* and *nP0y*, along with the cross vector, to give it the same width as the road. Therefore we added the implementation of the tunnel to the *bezier()* function. So for each pair of road vertices, we have an arch on top.

4.3.1 Vertices

As the roads are always on the xz-plane we just have to change the y coordinate of the tunnel to give it correct height over the road. We have decided to use a simple sine function to make the arch of the tunnel. However we should use a cosine function on the x and z coordinate of the cross vector, of the road vertex we are looking on, then substracting this one from *nP0x* and adding it to *nP0y*, as we do in the road implementation. This will allow us to progressively go from one side of the road, till the other. So we get a formula looking like this, with $\theta \in [0; \pi]$, and X and Y being the x and y coordinate of the cross vector at position $(nP0x, nP0y)$:

$$x = nP0x - Xcos(\theta)$$

$$y = sin(x)$$

$$z = nP0y + Ycos(\theta)$$

Seeing as we manually have to decide, how many vertices we will use to define the tunnel we created a for-loop wrapping around this. This one starts on 0 and ends on the density chosen (here 20). Therefore we now identify θ as: $\theta = \phi/density \cdot \pi$ where $\phi \in [0; density]$

4.3.2 Texture

Now that the vertices are created correctly, we are looking into mapping the texture correctly onto the vertices. On the x-coordinate of the mapping, we will simply use the same variable as used for the roads, seeing as we traverse the same distance. On the y-coordinate we just use $\phi/density$, so $\phi/density \in [0; 1]$.

4.3.3 Integration

Now having the vertices, and their respective texture coordinates, we have to, once more, add this to the object file. The way it is done, is we add all the vertices, and the textures created from the *bezier()* function. When that is done, we have to create the faces, as shown on figure 4.4. However we have complications, seeing as we have both the roads and the tunnels in our array. However we know that for two road vertices, we have 21 (seeing as we have a density of 20, and our loop goes from 0 to 20 included) tunnel vertices. Moreover we need 3 vertices to make a face, so to make the road, we need the three road vertices. Also, to better differentiate, we make a group every time we create a pair of faces, thereby differentiating between a road and a tunnel. However in our data structure, we have 2 road vertices at a time, so we have tunnel vertices in between, so we have to skip them (using a variable adding 2 with the density of the tunnel plus 1). Regarding the tunnel we do the same, however we need to skip the two first vertices (seeing as they are road vertices), and also the last one, as we connect with the next vertices (and if we didn't we would have tunnel vertices connecting to road vertices).

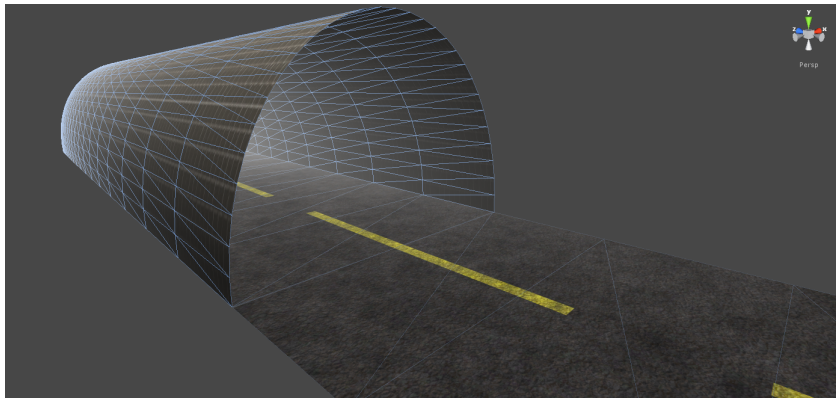


Figure 4.4: Final product of tunnel

4.4 Mountain

We based our implementation of the mountain, on the one of the tunnels. We integrate this one with the for-loop of the tunnel. This means that we have as many vertices on the mountains as the tunnels. However we found that the texture had problems working correctly. In fact, around the entrance of the tunnel, the texture got stretched, and made it look bad. So we had to partition the implementation of the mountain into an entrance and the actual mountain.

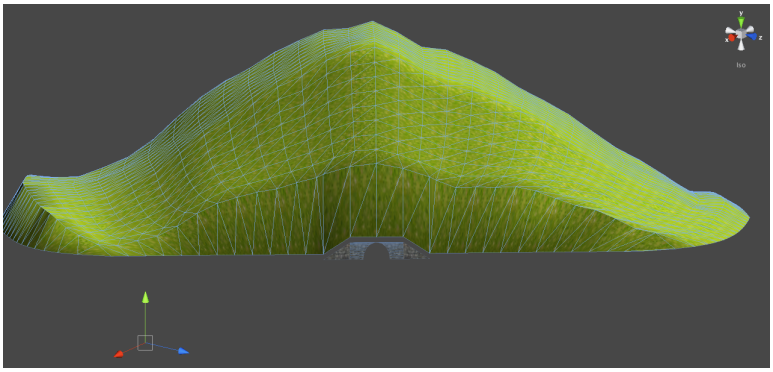


Figure 4.5: Final product of mountain

4.4.1 Vertices

Just like the tunnel, we have to define the height of the mountain. We use almost the same formula as the tunnel to make the mountain. We use $(nP0x, nP0y)$ again, however now we want a linear function defining the mountain, as shown on figure section 4.6.

Therefore we will find formulas for the x and z coordinates. We will use the variable ϕ, `density` and vec_{rw} as the ones described in 4.3.1, and m_w being the mountain width.

$$x = nP0x - (vec_{rw} m_w (1 - (\phi \cdot 2)/density))$$

$$z = nP0y - (vec_{rw} m_w (1 - (\phi \cdot 2)/density))$$

We see that $(1-(\phi\cdot 2)/density) \in [-1;1]$, as $\phi \in [0; density]$. Thereby we we will have $m_w(1-(\phi\cdot 2)/density) \in [-m_w; m_w]$, and by multiplying on the cross vector (which is normalized), we get the wanted vertices. Next we will look into the y coordinate. However in order to make the mountains a little more realistic we have a varying width. To do this we subtract a small decreasing fraction of m_w from m_w given by:

$$m_w - \frac{m_w}{(i+3)}$$

Where $i \in [0; steps/2]$, where *steps* is the total amount of steps we take for the bezier function.

As shown on figure 4.6, we have made a linear height for our mountain. Therefore to make it look real, we need to control the max height of the mountain on the segment that contains a mountain, as well as the height on the side.

To determine the max height we have made two functions, depending on how far on the bezier segment we are. We have $i \in [0; bs]$, `bs` being the amount of steps in our bezier function, m_h being the height of the mountain, and t_h being the height of the tunnel.

$$peak1 = \frac{3i\cdot m_h}{bs} + t_h$$
$$peak2 = m_h - \frac{3(i-2bs/3)m_h}{bs} + t_h$$

Looking at the first function, we have $3i\cdot m_h/bs \in [0; 3m_h]$, and the second function $m_h - 3(i-2bs/3)m_h/bs \in [0; 3m_h]$, however the first function is increasing, and the second function is decreasing (the first one starting at 0 finishing in 3*mh, the second one starting in 3*mh and finishing in 0). Moreover we add t_h, so the mountain does not cover the tunnel at any time. In the end, we have our actual `peak` which is found by taking the minimum of: `peak1`, `peak2` and m_h, thereby ensuring the height stays maximally by the value of m_h, and having the first third of the mountain growing, the second third stable, and the last third decreasing (as shown on figure 4.6).

Now that we have the `peak`, being the maximal height when advancing in the bezier, we will now make the actual height on the sides of the mountain. We identify two functions for this:

$$y1 = peak - \phi \cdot peak/density$$

$$y2 = \phi \cdot peak/density$$

We can see the first function $peak - \phi \cdot peak/density \in [0; peak]$ and the second function $\phi \cdot peak/density \in [0; peak]$, as $\phi \in [0; density]$. One more time, we made the first function decrease, and the second increase, and determine the actual height by taking the minimum of both. All this gives us a mountain looking like figure 4.6.

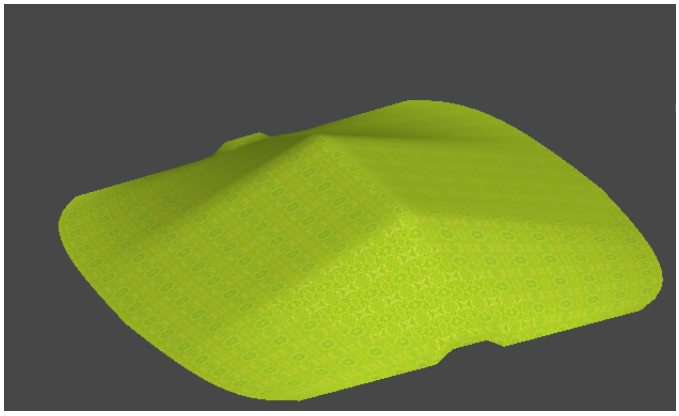


Figure 4.6: Basic mountain

However, figure 4.6 is just the basis of the mountain. For making it look more natural, and not mathematically generated, we applied the same random function as the one used for the terrains. Then we end up having a mountain that looks a lot nicer, shown on figure 4.5.

4.4.2 Textures

We decided to make the texture coordinate for the mountain very simple, and depending on the amount of vertices that define it. Therefore in the x-coordinate we use a modulo 2, for each vertex, and likewise in the y-coordinate. Thereby we will have either 1 or 0, and the texture will look correct. However we notice that the mountain will have more detailed textures the more vertices there is to define it.

4.4.3 Entrance

Implementing the entrance is about combining the tunnel and the mountain. We have made the mountain touch the floor, in all sides, however leaving a hole for the tunnel to run through. So what we want is to take the vertices where the mountain raises itself above the tunnel, and store them so we can combine them with the vertices of the tunnel. Therefore we are only interested in the first and last set of vertices of the tunnel and the mountain. The idea is that the vertices on the left and right side of the tunnel, should connect to a vertex having the same height, but fixed x and z coordinate being on the left and right of the tunnel respectively.

When that is done we give those points texture coordinates. In order to make it work, we use the cosine values for the tunnel vertices multiplied by the road width divided by the distance between two mountain points. We chose those values, as this gives us a direct relation between the distance to the vertex on the side, and how far the bottom point is from it. Therefore as long as the vertices on the side are further away from the tunnel's center, we will have a texture coordinate for the vertex on the side of 1, and $[0; 1]$ for the texture coordinates of the tunnel.

4.4.4 Intergration

Now that all vertices and texture coordinates have been implemented, we now make sure that they are written correctly to the *.obj* file. This time we have a separate data structure for both the entrance, and the mountain. To start with we write the entrance to the file as triangle strips. This means, that when we do not want to connect two consecutive vertices, we take both these vertices, and add them again in between. So for writing the faces, we have no special cases, we just write them as a triangle strip, and when we have two times the same vertex, it will not make a face. In the end we make sure we increase the offset for both the vertices and the textures.

After having made the entrance, we focus on the actual mountain. This one is written the same way as the tunnel 4.3.3. However we do not have a road interfering with the array, so we just have to be careful not to make a face, when reaching the last vertex of each row.

4.5 Simple Checking Program

As a little extension to our C++ code, we implemented a small OpenGL program (see figure 4.7), so the hospital can check each scene. The idea is that by arrow presses one can check every scene, making sure that tunnels and roads are placed correctly, and in agreement to what the hospital expected.

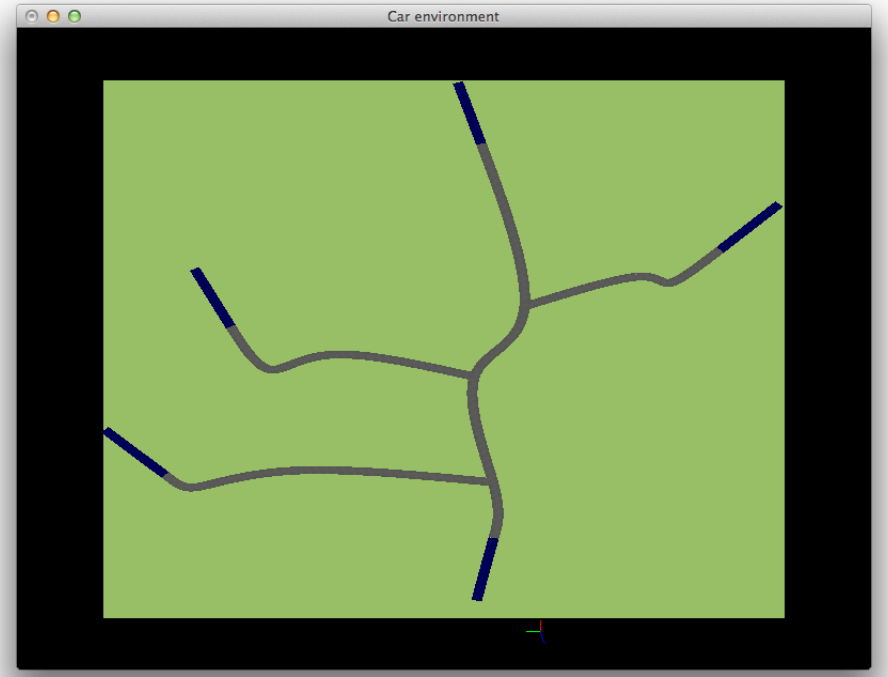


Figure 4.7: Our OpenGL output to check the scenes

After having generated all the scenes, we compute one scene once more. By having an index that we increment and decrement for every key press, we know which scene to compute. Each of our functions: `road`, `tunnel` and `mountain`, receive a file as input. When we want to write to the screen, we give a `null` file as input. In fact what we do, is we make the same calculations as we have done before, except that we do not write to a file, but to the screen.

We therefore make a check in each method whether or not the file is `null`. If it

is, in the `road` and the `tunnel` functions, we draw a triangle strip. In fact what both the `tunnel` and the `road` functions do, is make vertices that draw a road. Therefore in our `bezier()` function we make vertices and texture coordinates every time. Only when `gl_begin(GL_TRIANGLE_STRIP)` is called, then those are used to draw to the screen. However to tell the difference between a road and a tunnel on the OpenGL program, we have coloured the tunnels blue (see figure 4.7).

4.6 Car control

In this section, we will talk about how we implemented the way points, the follow-track mode using these way points, teleportation, how input is handled through file sharing and logging.

4.6.1 Way points

4.6.1.1 Basic

Way points are generated on every segment of road and follows the same bezier curve as the road. Way points are not visible, which is why they are modelled in Unity instead of the environment in OpenGL (although it may have been possible, but that would give us less flexibility in terms of control in Unity).

A way point is modelled as a whole new class with its own script in Unity. Most basic, it contains most of the information about the environment. Most importantly for the follow-track mode, it contains the position of the next and last way point in the segment. Because way point is modelled as a class, it has many get and set functions.

Way points are modelled as spheres in the game, but the form does not matter much, because it is invisible and used trigger events and hold information. The reason to model them as spheres is simply to avoid rotating them throughout the scene, because the road curves, and we wish them to cover the entire width of the road. When running through the bézier function (this time in Unity), we create the way point at the current position of the function. Each segment of bézier points, the same number of way points are generated.

4.6.1.2 Extended use of way points

Now that we have implemented way points, we use it to pass a lot more information to the car. As mentioned in section 3.8.1.2, we use the way points to pass fog and logging information to the car.

Upon creation, the way points get information from the scene, as which type of road the way point is on (tunnel or regular road), which scene it is on, and what the local coordinate (relative to the scenes) of the car is. The car script then processes the information and sets the different values that need to be set.

As the road identifier is not changed until the car is turning, and seeing as the turn is decided on a way point, the road identifier can be passed from every way point to the car, and the logging information will remain correct. Also when teleporting from one scene to another, it is decided by the way point, so there is no problem having the scene identifier designated by the way points as well.

4.6.1.3 Fog

We implemented fog[2] in our game, and we have the way points to control it. The way points check if the segment is a regular road, and the next one is a tunnel, and vice versa. If it is, it sends information to the car, as to, how many way points until reaching the end of the segment, and thereby the new fog settings.

Entering the Tunnel

As mentioned in 3.8.2, upon entering the tunnel, we use a fixed distance (set to 250m in our case) to decide our transition. In fact, when the car gets told by the way points that it is the last segment before a tunnel, the car starts checking, if the straight line distance to the last way point (found by taking the length of the vector between the car and the last way point) is less than the fixed distance. If it is, we compare the two, and use this percentage for our fog density. We use the difference between the environment fog density and the tunnel fog density, to know how much we should add to the environment density in order to eventually get the tunnel fog density. Also seeing as we have the percentage, as to how far we are from the last way point, we multiply this number to the difference, and we get a smooth transition, which gets updated by how far the car is from the tunnel.

Exiting the Tunnel

When exiting the tunnel, we weren't too interested in having a fixed distance by which the fog should have changed. But instead we defined a fixed time, by which the fog should change. We decided that the transition should start, when reaching the last way point, when still completely in the tunnel, and one second later (value which we have tested) the density should have changed to the new environment fog.

[2] The one that can be found under Render Settings in Unity

Figure 4.8: Fog transition from environment to tunnel

The way we make sure this happens, is by taking the time when reaching the last way point in the tunnel. Then we take the subtraction between the recorded time plus one second and the actual time. Then by using Unity's function `Lerp()`[3] [Uni12e] on the tunnel fog density and the environment fog density, and using the previously calculated difference to interpolate, we get the transition we wanted.

[3] Function which interpolates between two values

4.6.2 Follow-track mode

As mentioned we use the way points to orient the car, and tell it at which speed it should drive. However all the way points just hold information about its own location, and the location of the next way point. The next thing to do though, is to process this information according to the car's own location and orientation, which is varying. We have made a function, that calculates the car's orientation and speed. This one of course replaces the function that receives input and drives the car from that.

The function starts by calculating the direction of the car, which is the vector from the car position to the next way point position. Therefore the direction vector of the car is found by taking the next way point's position minus the car's position[4]. After that we calculate the car's orientation. As the car always drives on 0 height (y = 0 refvertexRoad), the orientation is found by taking the angle of rotation by the y axis (the one going up). This one is found by using Unity's variable `eulerAngles` [Uni12i], where the y coordinate hands out this value directly. Afterwards we can find the orientation of the car with the following vector:

$$orientation = \begin{pmatrix} sin(\theta) \\ 0 \\ cos(\theta) \end{pmatrix}$$

Now having the orientation of the car, and the vector that should orient it, we need the angle between them to know how much the car should turn. To find this angle we use the Unity function `AngleBetween` [Uni12k]. However, Unity's function returns a value between $[0; \pi]$, no matter if the angle is positive or negative.

Therefore we made function to get an angle with a sign. The way we do that, is that we take the perpendicular of one of the two vectors, and take the dot product with the other vector, using Unity's `Dot` [Uni12k]. Using this we now get values between $[-\pi; \pi]$.

Now having the angle between the orientation of the car, and the direction to the next way point, we want to define the angle by which the car should turn. If we made the car turn it all at once, it wouldn't look natural or flowing, so we need to take it bit by bit. We found that the turning angle, per frame, should not only depend on the total angle needed, but also by how far there is to reach it. In fact when meeting the way point, we calculate the distance, from the car's position to the next way point, and use this value, until it is updated, when meeting the next way point. We then divided the degrees the

[4] Using Unity's transform component which has an up to date vector as a variable [Uni12i]

distance. However by testing, we found that dividing the degrees by the half of the distance, makes the car reach the wanted way point in time.

Now that we have the degrees per update, we want to regulate the speed. For that we use a value, called **maximal speed**, that we use to compare with the actual speed the car has. If the actual car speed is more than 5% greater than maximal speed, the car should brake (throttle[5] = -1), likewise if the actual speed is less than 5% than the maximal speed, the car should accelerate (throttle = 1). We wish to regulate the speed according to the degrees that still remain to be turned. We have calculated a percentage of 45 degrees and the degrees we have. We chose 45 degrees, as we estimated that, when exceeding that, the car would need to slow down in order to turn correctly. When we have that percentage, we define the maximal speed as the multiplication of that and the default speed set by the XML file.

[5] variable taken from the tutorial 3.2.3

4.6.3 Teleport

As we have decided to make the car teleport from one scene to another, we had to find a way to make it look smooth. Once more we had to use the way points to control this, and to tell the car where to go. As mentioned in section 3.8.4 we need to have the segment of one exit tunnel of a scene, be the same segment as the first tunnel of the next scene.

Therefore we need to identify which way points are way points that are teleporter way points, and where they are teleporting to. As we asked the hospital to make the identical tunnels (as explained above) but also that the tunnels should have a minimum length of 250m[6], we defined from the third way point in that tunnel, the car should teleport to the third way point of the entrance tunnel in the next scene. We decided to take the third way point, as we did not want the car to see any part of the mountain when teleporting. And as we have a minimum length, we are ensured that the third way point (of 10), is more or less one third of the segment of the tunnel.

4.6.3.1 Translate

In order to make the teleportation work correctly we had to identify where the car touches the way point (which is round), and place it on the equivalent location on the new way point. In fact each way point has a direction vector, being the vector pointing to the next way point.

To start with, we want to find vector from the center of the way point to the location where the car hits the way point:

$$location = Carxyz - WayPointxyz$$

Afterwards we take the angle between this one, and the vector from the center of the way point, to the next way point (using the angle function created for the follow-track 4.6.2). The next step is to put the car in the right location, using a rotation matrix [Wei12]. Seeing as we only drive in the xz plane, then we only need to perform a rotation on the x and z coordinates.

[6] We need them to have a certain length as the driver should not be able to see out from the tunnel 3.6.2

$$P_0 = newWayPoint_{xyz}$$

$$P_1 = newWayPoint_{xyz}.norm \cdot r_w + newWayPoint_{xyz}$$
$$x = cos(\theta)(P_{1_x} - P_{0_x}) - sin(\theta)(P_{1_z} - P_{0_z}) + P_{0_x}$$
$$z = sin(\theta)(P_{1_x} - P_{0_x}) + cos(\theta)(P_{1_z} - P_{0_z}) + P_{0_z}$$

What we do is we take the orientation vector of the new way point, and use it to place a reference point P_1 on the border of the new way point (as the way points have the radius of the road's width). Then we use P_0 to define the center of the rotation of P_1. In fact if we have the rotation matrix given by:

$$R = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}$$

Now we apply the rotation matrix to the point that is placed on the border of the way point. However if we want to make a correct rotation, we have to translate this one back to the origin, which we can find by substracting P_0 to P_1. So we will rotate the point P given by:

$$P = \begin{pmatrix} P_{1_x} - P_{0_x} \\ 0 \\ P_{1_z} - P_{0_z} \end{pmatrix}$$

Applying R on P we now have:

$$x = cos(\theta)P_x - sin(\theta)P_z$$
$$z = sin(\theta)P_x + cos(\theta)P_z$$

Now that the rotation is done, we want to translate it back to where it was before in the world space. So we add the coordinates from P_0 back to the x and z coordinates calculated and we get the previously mentioned equation:

$$x = cos(\theta)P_x - sin(\theta)P_z + P_{0_x}$$
$$z = sin(\theta)P_x + cos(\theta)P_z + P_{0_z}$$

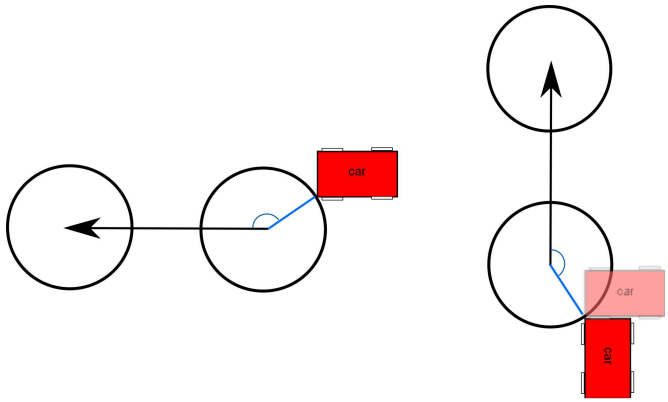


Figure 4.9: Translation in teleportation. Note that the car has not yet been rotated to have the right orientation, which is explained in the next subsection.

4.6.3.2 Rotate

Now that the car is placed correctly, we want it to have the correct orientation after being translated. To do that we simply find the angle between orientation vector of the teleporting way point, and the orientation vector of the way point the car gets teleported to. Applying turning the car by that angle, will be oriented correctly. The way we do that is we use once more the `eulerAngles` from the Unity's transform component [Uni12i], on the y coordinate seeing as we only rotate around this one. Also the car's velocity should also have the correct orientation. This one is retrieved by using taking the `velocity` variable from Unity's `rigidbody` component [Uni12h]. However this one is given as a vector, so we need to transform the angle into a vector.

$$orientation = \begin{pmatrix} sin(\theta) \\ 0 \\ cos(\theta) \end{pmatrix}$$
$$rigidbody_{velocity} = orientation \cdot v$$

where v is the velocity of the car. The way we do that is by taking the sine of the angle on the x coordinate, and the cosine of the angle on the z coordinate, and normalizing the vector. We also take take the length of the `velocity` vector, which gives us the speed of the car as a number. Eventually we just multiply the speed on the vector found before, and the car's speed will have the right orientation.

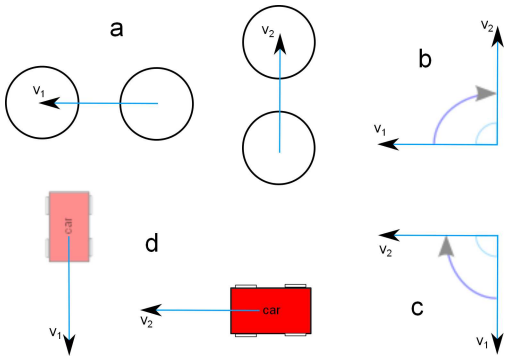


Figure 4.10: Orientation of the car. Waypoints (a) and their direction vectors, and the angle between v_1 and v_2 (b). The rotation is applied to the car (d) with the rotation shown (c).

4.6.4 Camera

Now that we have moved and rotated the car, we want the car to be oriented correctly. In fact, when the camera view is outside of the car, it has smooth movements between old and new positions. However, as we do not want the player to notice that the car has moved, we want the camera's location to be updated just as the car teleports. Therefore we made a new function in the camera class[7] that we called `reset()`. All this function does, is set the velocity of the camera to be exactly the same as the car, and updates its previous velocity, which it uses to make smooth transitions in other cases.

4.6.5 Logging and Follow Track

The last thing we need to do, is to update the logging variables and the follow track mode. In fact, the hospital are interested in the time since start-up, and

[7] Taken from the car tutorial [Uni12a]

the time since the car enters the current scene. Also the hospital is interested in the local coordinates on the scene (seeing as the world coordinates are subjective 3.8.4), and seeing as we are moving each terrain in the z direction, we just need to update one variable with how much we have translated the scene. Thereby subtracting this from the car's z coordinate we get the local coordinate of the scene.

Also, we found that we needed to disable the follow-track mode when teleporting, so the car does not try to drive somewhere where it shouldn't, as it might try to drive to a way point on the old scene when teleported. Therefore we decided to stop all driving actions when we teleport, and enable them again, when we meet a new way point which is not teleporting.

4.6.6 Input and decision

Unity has its own way of dealing with the control of the car. They simply make use of the left and right arrows of the keyboard, which in Unity changes the values of horizontal and vertical axis of a keyboard value I, where $I \in [-1, 1]$. In the horizontal case, when the left key stroke is held down, it reaches the value -1 whereas 1 means that the right key stroke is held down.

As we were not able to use Unity's function `getAxis()` [Uni12d], seeing as it reads directly from the keyboard, and interpolates values between -1 and 1, we had to make it our selves. We therefore based our implementation on the one, Unity has documented on [Uni12d], both for throttling and steering, where inputs *left*, *right*, *up* and *down* from the text file, are transformed into -1, 1, -1 and 1 respectively.

But to set these values, we have to read from the input file and make changes at the same time. In Unity, reading and popping a line was not as straightforward as we hoped and therefore, we came up with another solution. We read the first line of the file with `File.ReadLine()` and then we read the rest using `File.ReadToEnd()`. After we write the rest to the file, replacing what is already there. However, this is not the optimal solution and it may raise some problems with the sharing violations of the file.

The input we receive is pushed into an array. In follow-track mode, if the way point happens to be a decision way point (checked by calling `getIsDecision()` contained in the way point class), we simply check the last contained value in the array. If the last input is left, the car turns left, otherwise right. To take the *decision*, we define an angle *theta*, which is the angle between the overlying way points' direction vector. However the way we choose if it is the reference way point (the one that directs the car on the same road as the one it was previously driving on), is by checking if it has the same `road_id` as the car. If $\theta < 0$, then the way point is on the left side of the car where as $\theta > 0$ means the way point is on the right side of the car. See figure 4.11.

If the car is in free-mode, the horizontal and vertical axis of the keyboard entries remain their appertaining values.

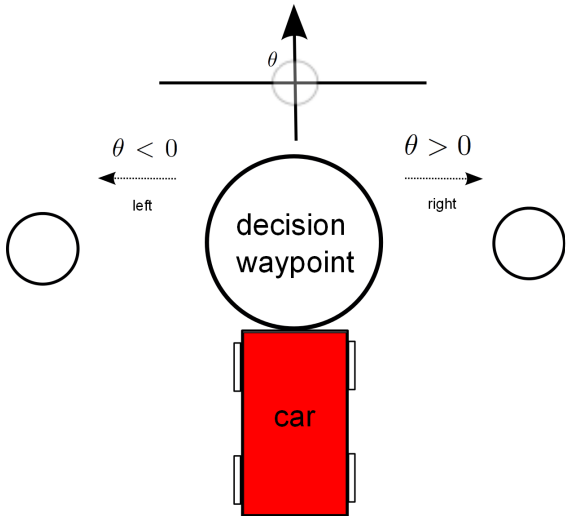


Figure 4.11: Decision taking.

4.7 Logging

Logging depends much on the File handling in Javascript, where logging takes place, and we use System.IO for this purpose. This namespace has most of the file handling functions needed and the ones mostly used in the program are `CreateText()` and `AppendText()`, whose names are self explanatory.

Basically, we keep the logging file open, and we append text to the file dynamically. The data needed is shown in 3.9. They want the global time and the scene time. The global time, is the time since the game started, whereas the scene time is reseted for every scene, which means that at every teleporting point (see Follow-track 4.6.2) we reset the scene time to 0.

We also need to update the z-coordinate of the car. As explained in section 4.6.3.1, we move each scene in the z-axis. Therefore, we need to know, by how much the current scene has been translated. This one is updated as explained in section 4.6.1.2. Removing this from the car's z-coordinate, we get the local coordinate of the car on the scene.

All the other variables are accurate, and can directly be appended to the logging file.

Chapter 5

Results

In this section, we will present the various tests conducted and the results of these tests. Most of the tests for the environment and the game creation, has mostly been done throughout development. Some tests were done by simple debugging, by using the built-in debug tools in the IDE's.

5.1 C++ and OpenGL

The initial tests were done in the C++ environment, where we simply rendered the environment in OpenGL. These tests grew rapidly into tests that were Object file (.obj) related and we needed to integrate with unity, meaning that we could no longer take a glimpse at the environment without looking at the contents of the object file.

In the integration process, we used Unity to test the environment as we loaded the object files. Even to test the object file and how it was working, Unity was used. The process was simply a matter of drag and drop of the object file from the C++ environment to the Unity import folders and thanks to Unity's auto detect and import feature, this made testing much easier.

The whole environment was created this way. Vertices, texture coordinates and

normals are calculated in OpenGL and the object file are generated and tested in Unity.

5.2 Unity

Apart from the object file generation, we use Unity to tests the gaming behaviours itself, meaning the car control, way points generation and other render settings in the Unity environment. To be able to test our environment on the car, we add mesh colliders to prevent it from falling through the environment. Way points normally have their mesh renderer turned off, because we want the way points to be invisible in the final product, but for testing and debugging, it was convenient to have the way points rendered so we could actually test when the car was hitting the way point and thereby triggering a certain effect that we wanted triggered. See figure 5.1.

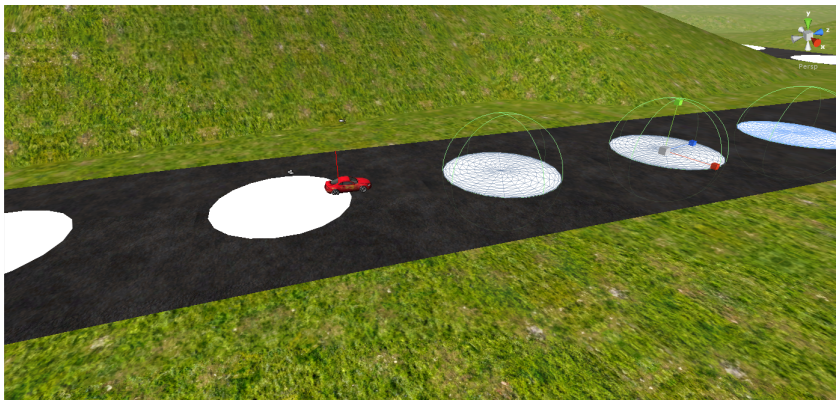


Figure 5.1: Testing of way points. The mesh renderer of the way points are turned on to make it easier to debug, when the car reaches the way point to trigger a certain event.

When working with scripts in Unity, it is very easy to debug certain variables by use of the debug tool in Unity. One can simply write `Debug.Log(var)` and the variable is shown in the console log. Also, some global variables can be set to public, so when running the program, the Unity's inspector window shows many variables which update frequently as the game is running.

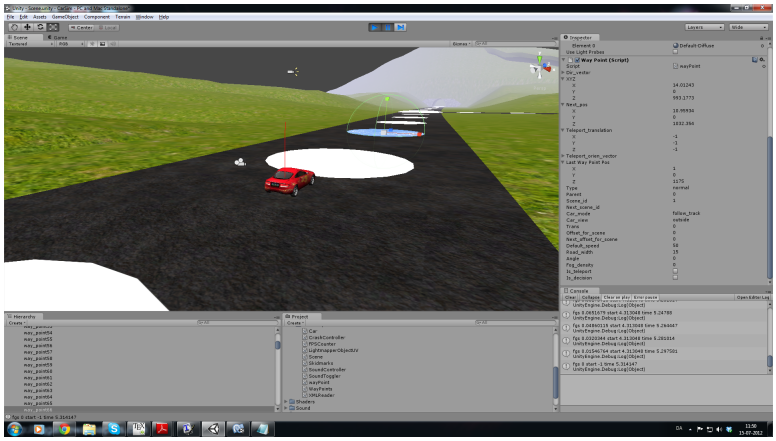


Figure 5.2: Unity development interface. On the right, you can see the inspector window, showing all the variables such as directing vector, type of way point, fog density etc. The game is paused, but when it is not, the variables are changed at every frame. Below the inspector view, console can be used to print out additional information in the code.

5.3 Hvidovre hospital - Experiments

At the closing of our project and the thesis, we departed to Hvidovre hospital to start with the testing of our car-simulation game.

The day started off with discussing what the computer in their department required to run our game. The pre-installation requirements are listed below.

1. OpenGL, glut and glu dll files.
2. Microsoft Visual C++ 2010 Redistributable package
3. Included files such as texture in the folder
4. Unity game engine

The first three mentioned is to run the C++ program. To build the game as a whole, unity game engine is required. It took some time to install, import the scenes and build it all, so in the meantime, we discovered some errors in their spline coordinates such as the side roads being too close to one another, which

made the mountain coat the other side road. After some minor bug fixes to the XML file, we generated the whole scene.

We had prepared a simple guide, C for them to follow when compiling and building it up. They were quite happy with the simplicity of the process. Konrad Stanek had prepared a python program to run simultaneously with our program in the background to input to and output from the text file. We tested his and our program together. It worked fluently in follow-track mode. However, in free-mode, it seemed to yield some problems to the control because of sharing violations of writing to the same file.

But this did not matter too much for their experiments, because they, to begin with, are mostly interested in the follow-track mode. Therefore, we decided to conduct a real experiment with the EEG set up.

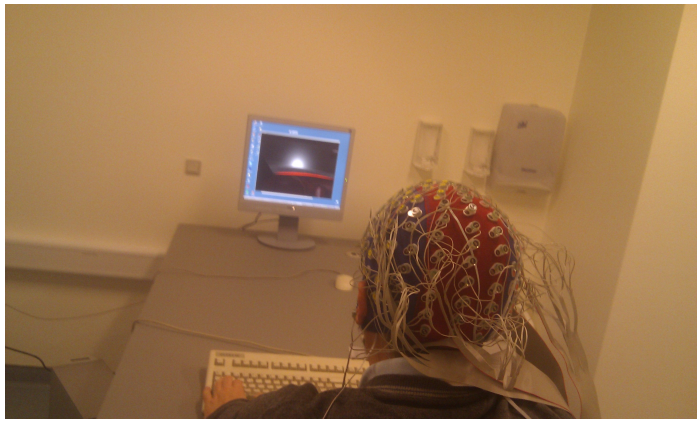


Figure 5.3: EEG on the subject.

As shown on the picture, the subject is equipped with the EEG. The set-up was done in an isolated room, where no magnetic noise was interfering with the signals from the EEG. Therefore, all the electric devices and cables were turned off.

When the subject started playing with the game, the decisions at the turning points seemed to work naturally for the subject. He had no problems in doing so, and the track was completed successfully with no major errors. The minor problems encountered was the car, trying to stabilize its orientation after teleportation. Other than that, every scenes was run, and the experiment was successful.

The research group recorded the data sent from the EEG while the subject was playing the game. Later, they will try to investigate the brain signals and the logging file and see if they can find any correlation, between the subject's decision taking and his brain signals.

Overall, the tests were successful and they were quite happy with our product. More information and screen shots about the experiments can be found in their brief in Appendix B.

Chapter 6

Discussion

6.1 Project discussion

Throughout the project, we have tried to keep track of our progression, by looking at our project plan, which we have created in the start of this project. We have been more or less on track the whole time, and we were too much behind on schedule.

From the start of our development of the environment, we tried to be as flexible as possible. This meant that we had variables for almost everything and not much was hard-coded. Also we made the algorithms in such a way, that they are extendible. At the beginning, we did not have any XML to read from so it was a little hard to keep the values from being hard coded. When we finally had the XML, we set it all up and read its values and it was working as intended. From our experience, this rarely happens and it is thanks to our flexible coding style, that the transition from hard coded variables to those read from the XML was smooth.

What we can be proud to say is that we have a final game up and running, which is therefore ready to be used in experiments. The hospital is very pleased with our car simulation program and the tests were conducted successfully, as explained in the results chapter.

6.2 Limitations

When the hospital is running the experiments, there are certain things that need attention when modeling the environment. The first thing is the mountains. Mountains have a certain width, that is configurable in the XML. However, if the width of the mountain is bigger than the distance between two roads (meaning two side roads are too close to each other), the one mountain may cover some areas of the other side road. This can be fixed by either reducing the mountain's width, or by separating the side roads furtherer away from each other.

The car is meant to drive on way points, so driving outside them won't trigger certain events such as fog. If the car is driving outside of the road in free-mode, the fog won't adjust correctly, because the way points control the fog. One possible way of fixing this problem could be to check the position of the car from each tunnel. Also, if the roads in the XML are too short, way points might be overlapping, which may cause problems when driving in follow-track mode.

Many variables such as speed of the car, steering, fog and mode (follow-track or free) are preset in the script. Most of these variables are made public, so it is easy to change them in Unity's inspector. Some of these variables may not exceed limits, as that may cause trouble. The maximum speed of the car can be set to over 100, but this does not have any effect on the car, seeing as it can only reach a maximum speed of around 70-80, because of the physics done by Unity.

6.3 Extensions

As this is a Bachelor project, and therefore limited time was available to make the program, there is many areas one could improve.

6.3.1 The Environment

There are countless details one could make, or improve in our environment to increase reality.

Bezier

1. **Unnecessary vertices**
   If one decided to make vertices according to distance instead of splines, one could spare a lot of unnecessary vertices (and way points) 6.1.

Figure 6.1: Example of areas with more vertices than needed.

Terrain

The terrain has countless areas in which it could be made better, as it the main thing the player sees when playing.

1. **Hills**
   Even though the hills on the terrain, which helps a lot against uniformity, the terrain remains very uniform. One could have made a shader, which changes the color of the terrain, according to what the altitude is.
2. **Shadows**
   Unfortunately we did not have Unity Pro, so if we wanted shadows in our game, we had to make them ourselves, using shaders.
3. **Vegetation**
   One could add trees and other kinds of vegetation, or boulders to the terrain, making it look a lot better.
4. **Human created objects**
   One could also add Human created objects, like road signs, traffic lights, or buildings to the environment.
5. **Obstacles**
   One could make obstacles like the ones mentioned above, on the road.

6. **Water**
   Water on low heights of the terrain, would look nice.

7. **Levels of Detail**
   It could be interesting to have varying levels of detail, according to variation on the terrain, for example one is more interested in having more vertices on the hills, and less on the flat terrains.

8. **Hilled roads**
   Even though it was a design specification, that the car roads should always be on level $y = 0$, one could make the roads hilly.

9. **Anomalies**
   We notice on places 6.2 (also related to the level of detail of the terrain) that we have steps on the terrain, because of the way we define the terrain height.

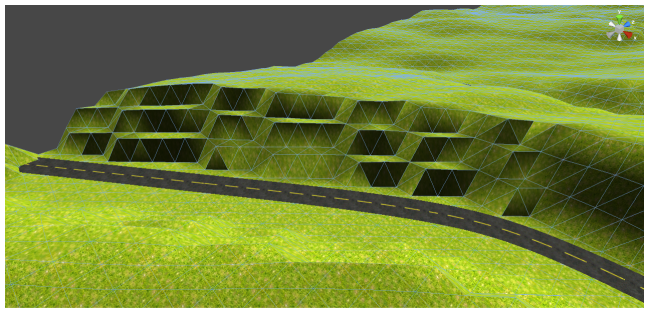


Figure 6.2: Example of terrain anomaly.

Mountains

We have found some small issues with the mountain, so one could also improve there.

1. **Mountains check road**
   We have made the mountains all with fixed size, and therefore at some points 6.3, the mountain could end up covering the road. One could check if a mountain vertex is covering the road at a certain location on the terrain, and edit its height accordingly.

2. **Varying heights and width**
   One could have varying widths and heights on the mountains, that would help credibility.

3. **Closing gaps between mountain and terrain**
   We have made our mountain, on the borders, have height $y = 0$, this means, if the terrain doesn't have height $y = 0$, there will be a gap between the mountain and the terrain, and one could fix it, by giving it the same height as the terrain.

4. **Level of detail**
   Right now, the mountain depends on the amount of vertices of the tunnel. However even though sharp and rough edges look nice on a mountain, the mountain need more vertices than the tunnels, and one could do with a lot less vertices for the tunnels.

5. **Shading of mountain**
   It would be nice to shade mountains, as they would have different color than the terrain, and one could make give them a white peak, making them more real.

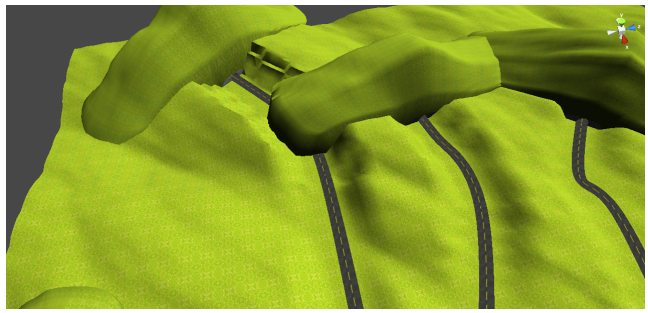


Figure 6.3: Example of a mountain covering a road.

Tunnels

There are also a few things one could improve in the tunnels.

1. **Lights on the ceiling**
   By adding lights on the ceiling of the tunnels, one would make the game more credible, as it is not completely dark, and the player might long for a source.

2. **Flash when exiting the tunnel**
   One could make a flashing effect, when exiting the tunnel rather than use only the fog.

6.3.2 The game

In the actual game, there are a lot of things that can be improved.

The Car

1. **Physics**
   The car is taken from a tutorial. This one has made some rough estimations about how the car should drive. If one were to improve the game, one should revisit most of those functions, and find new values, or solely change them.

2. **Different cars**
   If the game is to be interesting, one could make several different cars that the player could use.

3. **Advanced follow-track**
   In the case where the game would have hilly roads, the car should also adjust its speed according to slope.

6.4 Credits

In this section, we will structurally describe in which parts of the Car Game we take credit for and which parts is not developed by us. We feel that it is a great important to distinguish our work and from other's and with respect to the DTU's rules for the Bachelor thesis.

6.4.1 Distribution of work

All of the work and decisions have been discussed and implemented jointly. In the report, there were parts in which we delegated the responsibility of work.

- **Group work**
  Summary, Introduction and Conclusion was done in a group.
- **Bilal Arslan (s093268)**
  - Analysis
  - Method - Overview, Game development, Bézier, Terrain, Road, Way points (Basic), Input and decision, Logging.
  - Implementation - Terrain, Road, Way points (Basic), Input and decision, Logging.
  - Results
  - Discussion - Project discussion, Limitations, Credits.
- **Patrick Jørgensen (s093298)**
  - Method - Tunnel, Mountain, Way points (Extended), Fog, Follow-track Mode, Transition between scenes.
  - Implementation - Tunnel, Mountain, Simple Checking Program, Way points (Extended), Follow-track Mode, Teleport, Camera, Logging and Follow-track.
  - Discussion - Extensions.

Everything but what is mentioned below, has been done by us. In the case where something is found, that has not been mentioned below and not been made by us, we strongly apologize, as it has not been our intention to take credit for it.

6.4.2 What is NOT done by us

1. **Car**
   The car itself is taken from Unity's Car Tutorial. [Uni12a] This includes the car chassis, body, wheels, lights etc.
2. **General car behaviour and control**
   Steering, throttling and many other physics simulation of the car is done by the Car script, developed by Unity. This could be done by us, but unfortunately, this was not the primary goal of the project and therefore, we made use of it and then decided to go back to this and try to make it on our own, if time enabled us.
3. **Car camera**
   The camera following the car is again done by Unity. Although, we have made small changes to make the camera follow the car from an inside view, to simulate a real driver driving the car.

4. **Unity**
   Many in-built functions and run-time rendering is done by Unity and many things are given free. If our program based entirely on OpenGL, then it would not be possible for us to make it this far into the project. It is thanks to Unity that we have made it this far.

5. **Shading**
   The shading is done by using Unity's standard shaders, which are available and is set as default when important certain objects. Even though, we could have done this by using GLSL (because we are experienced in that shading language) in Unity, which is possible.

6. **Fog**
   The rendering of fog is done by the Unity language and the only thing that we take credit for is adjusting of it, depending on the car's position on the road and the distance from the tunnel

7. **Skybox**
   The skybox is a built-in function in Unity, which is rendered to the camera. Thus, we do not take credit for having skybox in our game.

8. **Random generator,** `gel_rand()` This random generator is taken from the GEL library, developed by our advisor, Jakob Andreas Bærentzen.

Chapter 7

Conclusion

What can we conclude from all this? Most of our work has been based on the hospitals demands and what they prioritised. They started by not expecting too much from our game: a simple car-simulation game. They were not disappointed, in fact it was a little more advanced than they expected.

All of their requirements are fulfilled. Our game is able to successfully read the environment data from an XML file, create a whole environment with multiple scenes, integrate this environment into the game environment in Unity, and finally we can simulate the game using the entire environment. We have made the car able to drive in two different modes, namely the free-ride mode and follow-track mode, both of which are controlled by a simple text file. How this file is created, whether it is by neurofeedback, or just by keyboard presses, is not our task.

To keep track of their experiments, the hospital also asked us to create simple data logging within a time scale (a matter of milliseconds), which has also been made configurable.

As for the graphics of the game, the hospital people were more than satisfied. They did not expect much. The complications were to meet our own expectations, taking into consideration the tools and the time available, which we have.

The programs have been delivered to the hospital and it seems they do, what they are supposed to. They have acknowledged and complimented our work, meaning our work is very likely to be used for future in future neurological experiments, which pleases us. Even so, there are a lot of improvements and extensions, we would have enjoyed making 6.3, however this is but a Bachelor project, and if those are to be brought to life, is in the hands of another project. The hospital is now ready to get started with their experiments, they just need subjects. This concludes that our work has been accepted and our final goal has been reached.

Appendix A

Cubic Bézier curves

This appendix includes the documentation of the cubic bezier points used in our project. The paper is from Steffen Angstmann, one of the Ph.d. students at the hospital who is going to use our program for experiments.

Cubic Bézier curves

1. Overview

Bézier curves are a method of designing polynomial curve segments when you want to control their shape in an easy way. Bézier curves make sense for any degree, but we'll concentrate on cubic ones, the most important case. ("Bézier" = "Bay zee ay".)

To specify a cubic Bézier curve, you give four points, called *control points*. The first and last are on the curve; the middle two may not be. When you change the control points, the shape of the curve changes. It is helpful to indicate the control points by connecting them with line segments to form the "control polygon" (although this is not a polygon in the usual sense, as it is not closed). Some examples are shown in Figure 1.

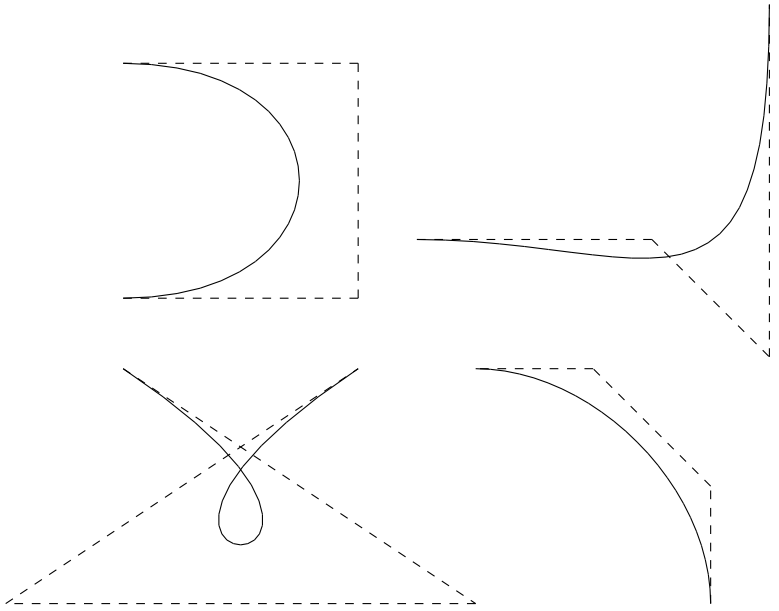


Figure 1: Some Bézier curves

It does not matter which end you consider to be the first and which the last; you get the same points for the curve either way. Observe that the curve is tangent to the first and last "legs" of the control polygon.

2. Details

The Bézier curve is just a particular linear combination of the control points with time-varying coefficients. If the control points are P_0, P_1, P_2, P_3, then the curve is given by

$P(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3$, for $0 \le t \le 1$.

Why these coefficients? They arise in a way related to binomial expansions. Recall that $(s+t)^3 = s^3 + 3s^2t + 3st^2 + t^3$. Now consider these terms individually rather than added together, and put $(1-t)$ for s. You get four functions of t, called the *Bernstein polynomials*. These are

$$\begin{array}{rcccl} J_{3,0}(t) & = & (1-t)^3 & = & 1(1-t)^3t^0 \\ J_{3,1}(t) & = & 3(1-t)^2t & = & 3(1-t)^2t^1 \\ J_{3,2}(t) & = & 3(1-t)t^2 & = & 3(1-t)^1t^2 \\ J_{3,3}(t) & = & t^3 & = & 1(1-t)^0t^3 \end{array}$$

Thus for a Bézier curve,

$P(t) = J_{3,0}(t)P_0 + J_{3,1}(t)P_1 + J_{3,2}(t)P_2 + J_{3,3}(t)P_3$.

As you will see, the Bernstein polynomials have nice properties that are reflected in the properties of Bézier curves. Bernstein polynomials and Bézier curves can be defined for any degree n by using the expansion of $(s+t)^n$, but let's continue to concentrate on the case of degree 3, since that case is most frequently used.

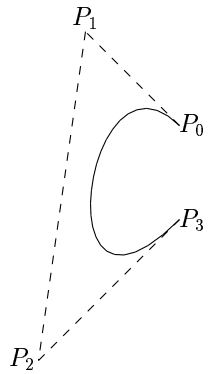


Figure 2: Example

Example. Suppose the control polygon has
$P_0 = (2,3)$, $P_1 = (0,5)$, $P_2 = (-1,-2)$, and $P_3 = (2,1)$, as in Figure 2. Then
$P(t) = J_{3,0}(t)P_0 + J_{3,1}(t)P_1 + J_{3,2}(t)P_2 + J_{3,3}(t)P_3$
$= (1-t)^3(2,3) + 3(1-t)^2t(0,5) + 3(1-t)t^2(-1,-2) + t^3(2,1)$
$= (2 - 6t + 3t^2 + 3t^3, 3 + 6t - 27t^2 + 19t^3)$

Appendix B

VRE with EEG

Here we present the brief done by Konrad Stanek and Steffen Angstmann at Hvidovre Hospital.

Application of Virtual Environment to cognitive science research at DTU/DRCMR.

Background

The virtual environment (VRE) will be actively used in the series of experiments aiming at elucidating the neuroanatomical and physiological correlates of human decisions. During simulated car driving, the fast electroencephalographic data (EEG) will be acquired to map neural activity of participants with their actions, with purpose of inferring decisions before the actual performance. The goal of the research is to contribute to the state-of-art technology in the field of brain-computer interfacing (BCI) and to better understanding of the decision processes in humans.

Paradigms used so far for investigating voluntary actions mostly employed very abstract stimuli as letters or pictograms (e.g. (Krieghoff, Brass et al.2009), prompting one specific decision. Inherent in doing so is a lack in external validity (Haggard 2008) and, moreover, they are prone to random sequence generation behavior (Jahanshahi and Dirnberger 1999; Lau, Rogers et al. 2006), which is a strong confounding factor in brain signals. A virtual reality environment (VRE), on the other hand, will provide a more natural platform similar to real life condition. At the same time, however, it will enable the experimenters to carefully control and manipulate the environmental parameters, participants' degree of freedom, inclusion of various types of decisions; hence making it possible to analyze different aspects of voluntary action. In particular, decisions can be investigated with regards to three different dimensions (tasks, according to (Brass and Haggard 2008)): "whether" to perform an action (whether to take a side road or not), "what" type of action to perform (left or right side road) and "when" to perform an action (which side road to take).

Requirements

Several specific requirements must have been imposed on a virtual environment to make it applicable to the aforementioned research. First, it must be flexible and highly configurable, allowing the researcher to define exact road coordinates, visibility, terrain shape and size, etc. Second, it must implement a follow-track mode of steering, where the car follows the direction of the road and participant's degree of freedom is reduced to selection of the side roads. Third, it must provide a sensitive logging mechanism (millisecond precision), in order to synchronize acquired brain data with in-simulation events. Finally, it must implement a flexible input mechanism, so that any EEG/fMRI-specific input devices may be used in place of standard keyboard. None of the existing of-the-shelf games meets the stated requirements, and hence dedicated virtual environment, being subject of this thesis, has been developed.

Deployment and test experiment

The final version of the virtual environment has been deployed and thoroughly tested in DRCMR (Danish Research Center for Magnetic Resonance, Hvidovre Hospital). Three blocks of twelve trials were performed

with simultaneous EEG data acquisition from one healthy subject. One trial corresponds here to one tunnel-to-tunnel road stretch, with decision being freely taken by the participant at each crossroad. The events corresponding to keyboard press are forwarded through parallel port to mark the time points on continues EEG data stream. The hardware setup consisted of BioSemi ActiveTwo system (128 active scalp electrodes, amplifier, optical box); VRE computer running the game and sending synchronization triggers to optical box; recording computer running ActiView acquisition software. Figures 1, 2, 3 and 4 illustrate the experimental setup.

Fig.5 shows the brain signals recorded while the participants navigated through the environment. The signals were acquired from 128 active electrodes covering entire scalp, digitized with 2kHz sampling rate, high-pass filtered at 0.16Hz, and low-pass filtered at 100Hz.

The acquired EEG data is stored for subsequent analysis. The type and extent of the analysis will depend on the paradigm and experimental design, and data from multiple participants will be usually combined to account for statistically significant population effects. In case of this particular experiment, our purpose was to validate the VRE/EEG setup rather than to test any specific hypothesis. The number of recorded trials would need to be significantly higher to observe cognitive effects. We restrain therefore here to only few exemplary views on the acquired data, all related to right-turn decision trials. Scalp distributions of electrical activity are shown by Fig.6, example of event related potential by Fig.7, and distribution of spectral power is illustrated by Fig.8.

References

Brass, M., P. Haggard (2008). "The what, when, whether model of intentional action.", Neuroscientist 14(4): 319-325.

Haggard, P. (2008). "Human volition: towards a neuroscience of will." Nat Rev Neurosci 9(12): 934-946.

Jahanshahi, M., G. Dirnberger (1999). "The left dorsolateral prefrontal cortex and random generation of responses: studies with transcranial magnetic stimulation." Neuropsychologia 37(2): 181-190.

Krieghoff, V., M. Brass, et al. (2009). "Dissociating what and when of intentional actions." Front Hum Neurosci 3: 3.

Lau, H., R. D. Rogers, et al. (2006). "Dissociating response selection and conflict in the medial frontal surface.", Neuroimage 29(2): 446-451.

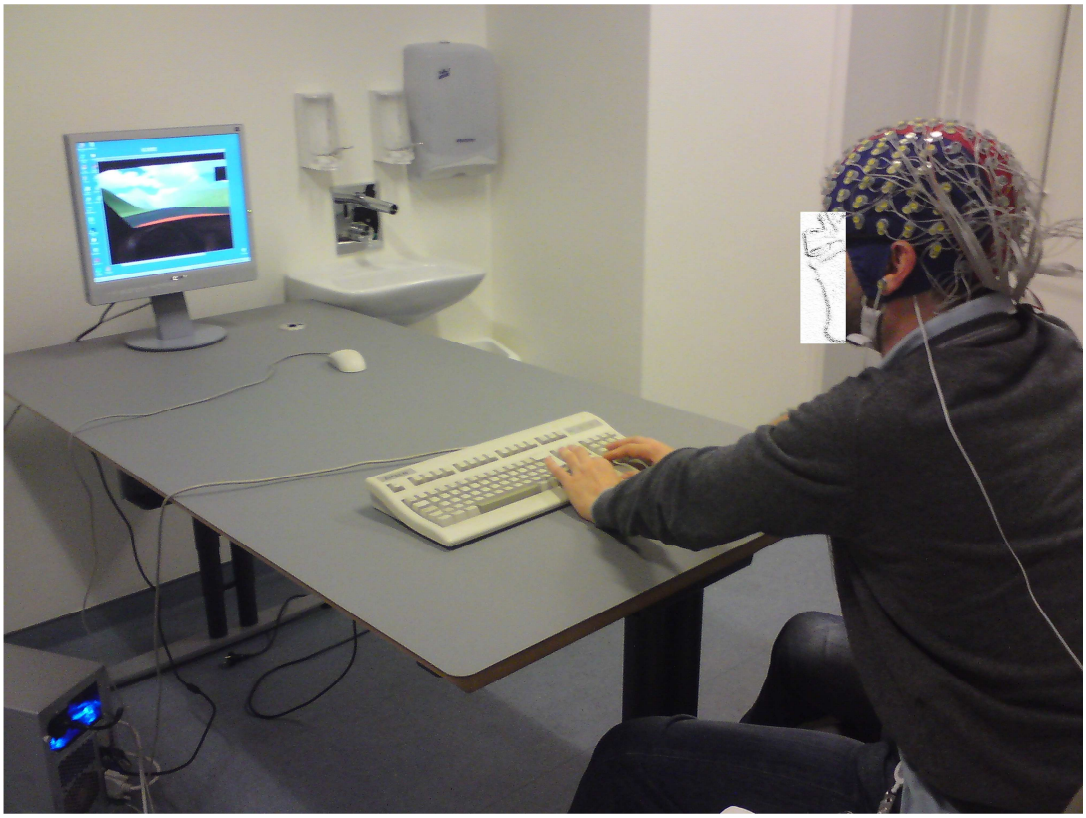


Fig 1: Experimental setup – subject prompted to press left/right arrow keys with left/right index finger to decide on driving direction. Computer and LCD screen positioned in relative distance to reduce 50/60Hz noise.

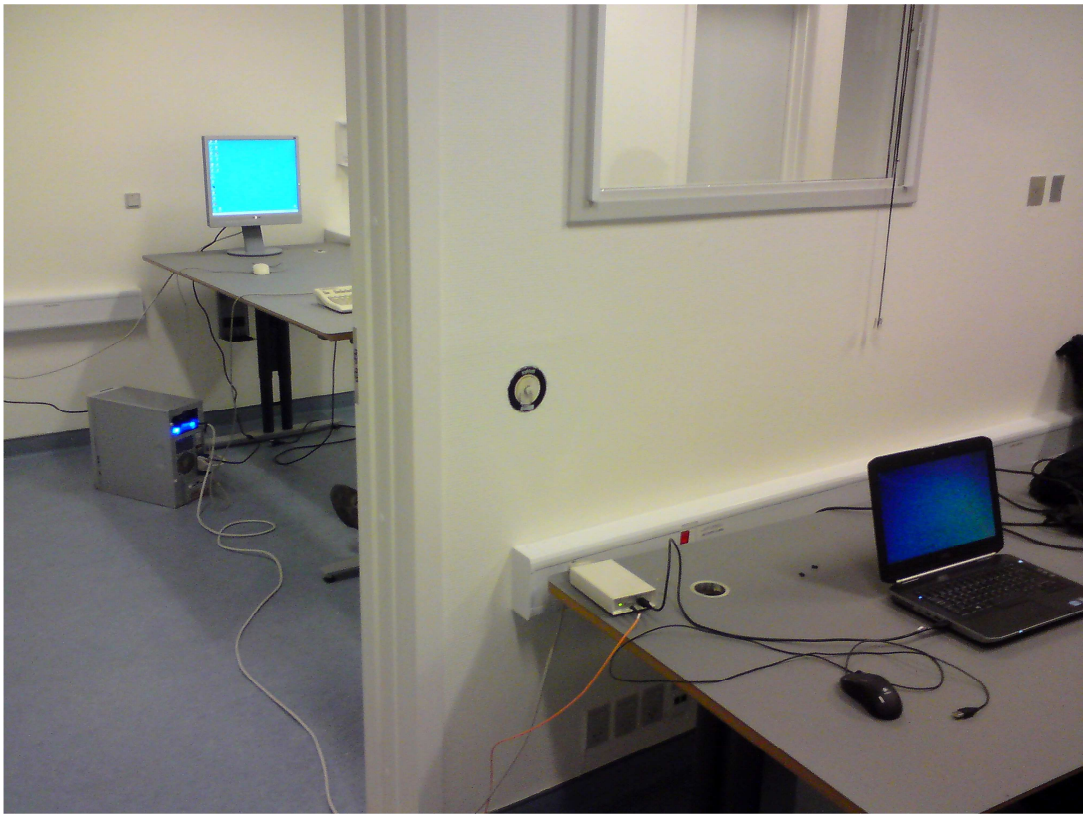


Fig 2: Experimental setup – recording computer placed at adjacent room. The optical box receives EEG data through optical fiber cable (orange cord) and synchronizing trigger signals through parallel port (white cord) from the computer running VRE.

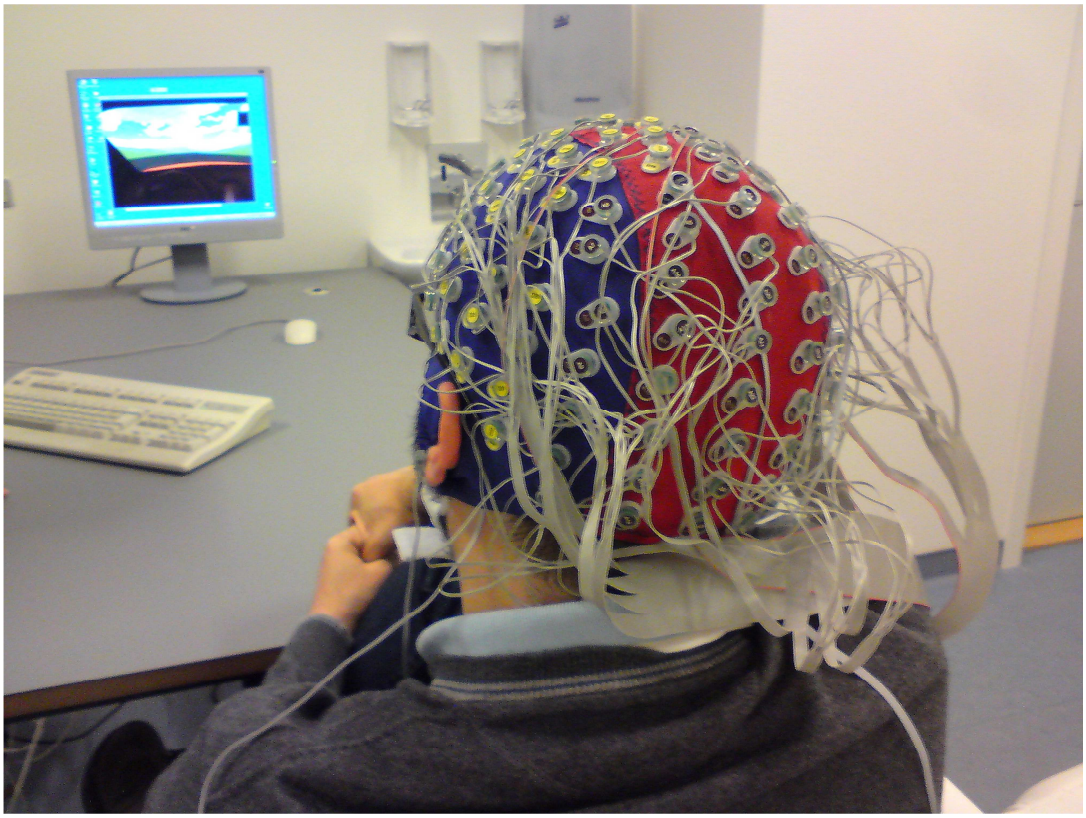


Fig 3: BioSemi ActiveTwo system used for EEG recordings. 128 active electrodes, distributed evenly across subject's scalp, capture miniature potential differences corresponding to cognitive and motor neural activity (microvolts scale).

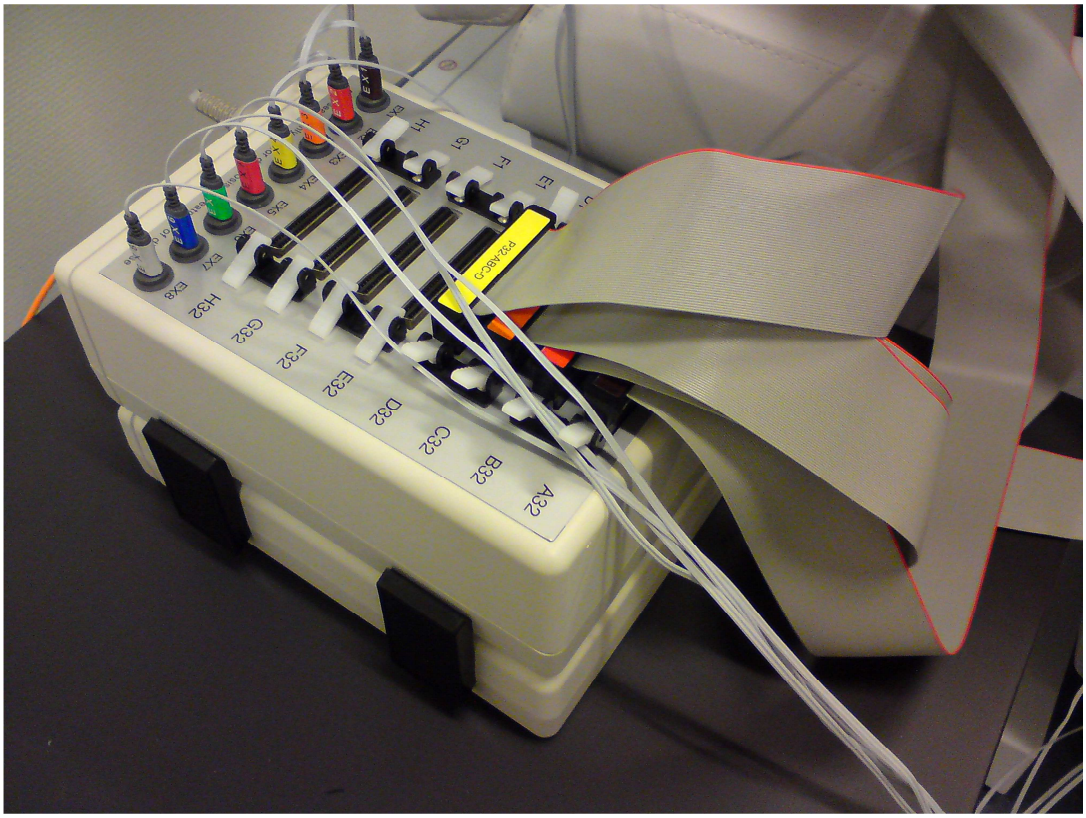


Fig 4: BioSemi amplifier and AD converter box. The amplified, digitized signal from here is sent via optical fiber cable to the optical box and further to the recording computer.

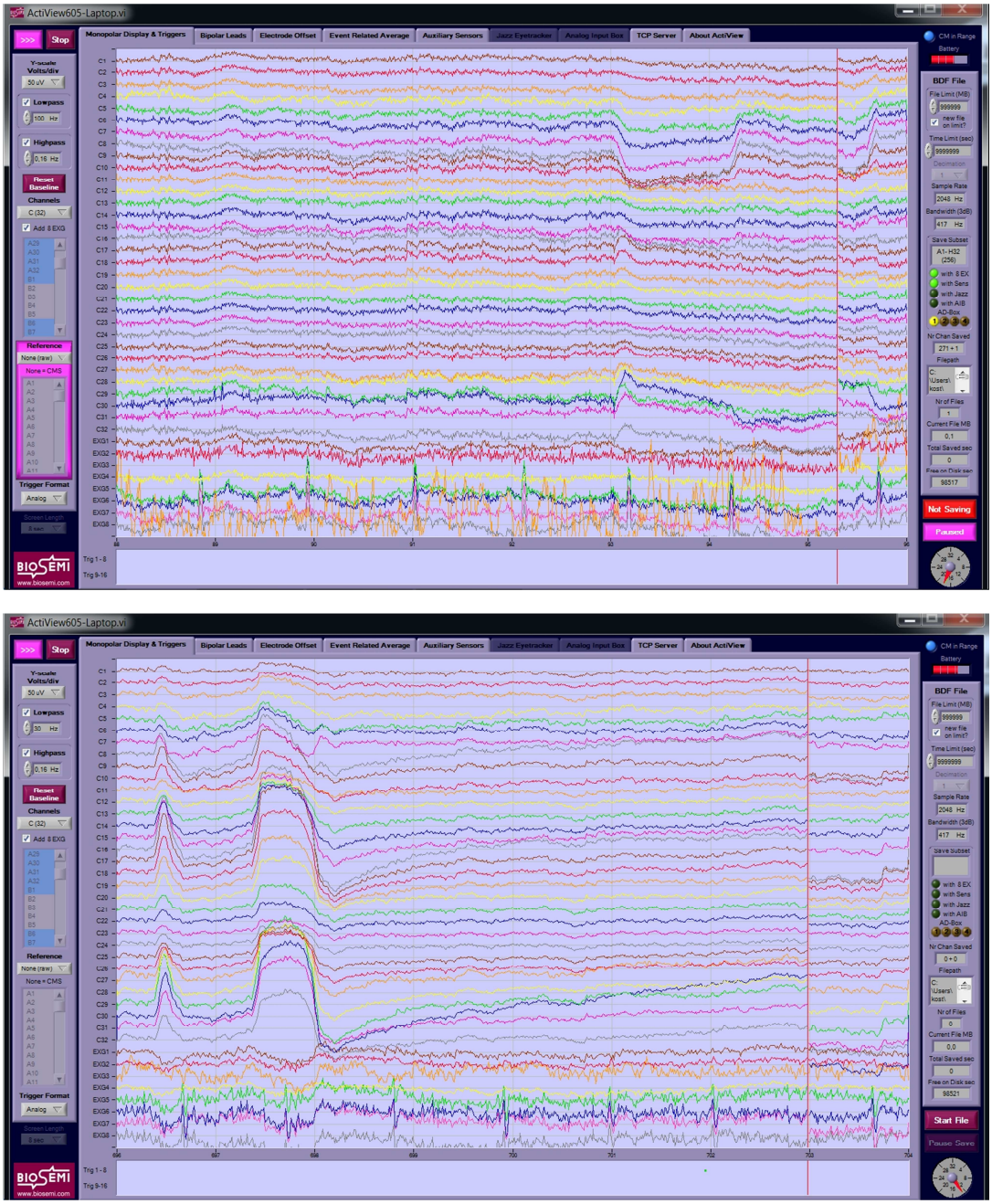


Fig 5: Exemplary screenshots from ActiView EEG acquisition software. Subset of 32 posterior electrodes is shown. Note the two large waves at the bottom figure, corresponding to eye blink and saccadic eye movement respectively.

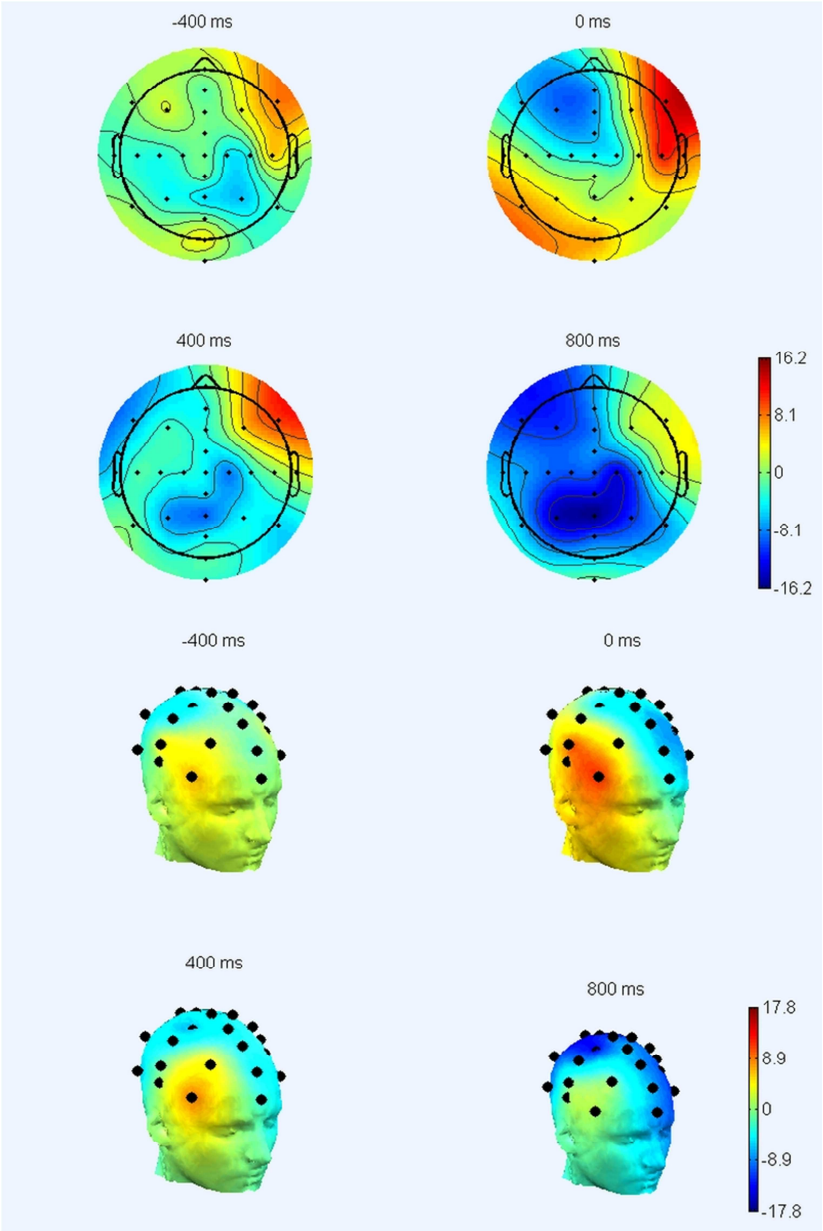


Fig.6: Scalp distribution of electrical activity corresponding to right-turn events (averaged over 12 available trials).

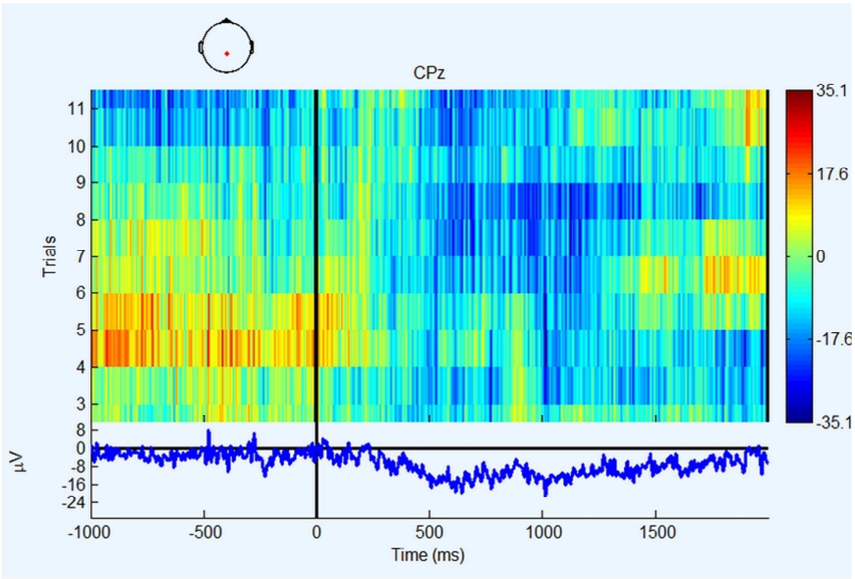


Fig.7: Event related potential (bottom), resulting from averaging through 12 trials (top), time-locked to right-turn decision (black, vertical line). Recorded at medial centro-parietal electrode (CPz).

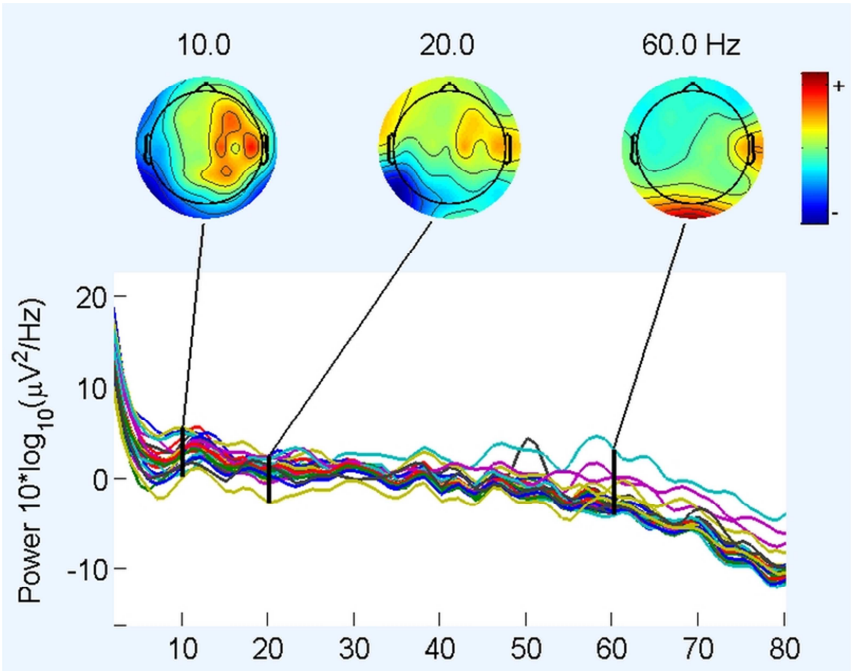


Fig.8: Spectral power distribution, measured at the interval of 1500ms preceding right-turn decision.

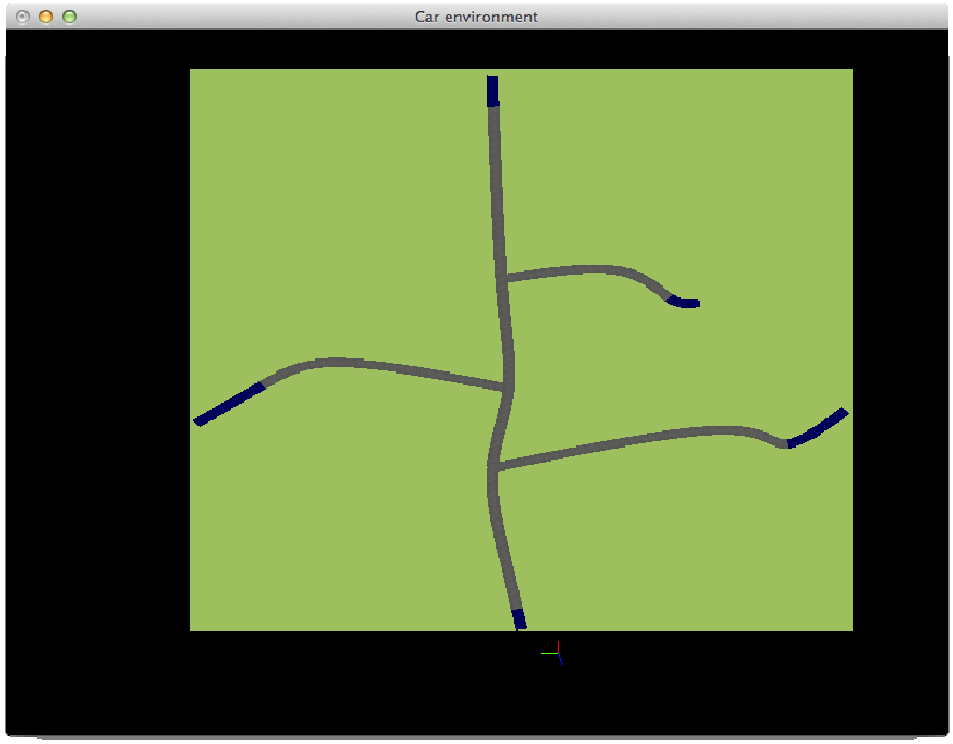
Appendix C

How-to guide

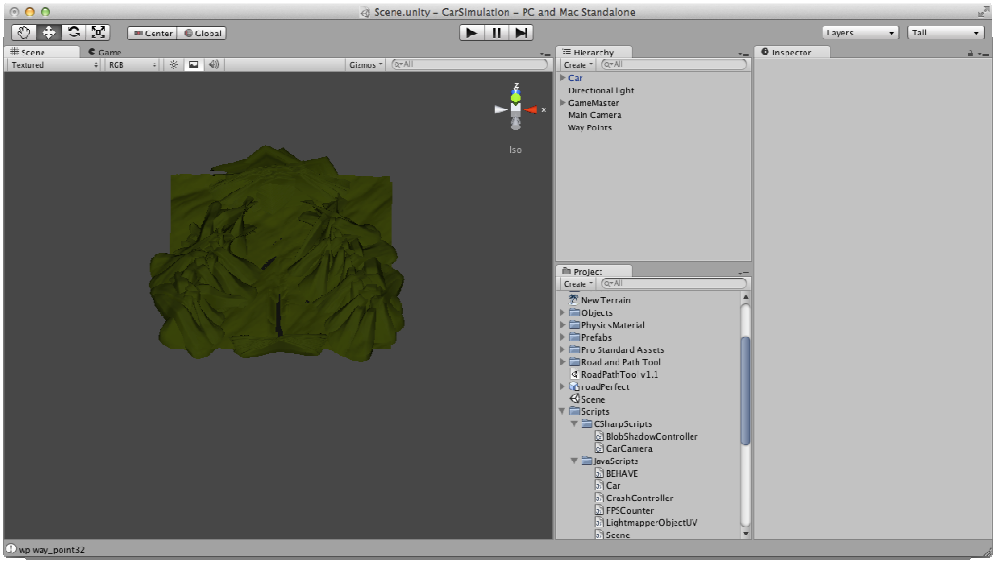
Below is the guide used to guide the research group at Hvidovre Hospital on how to compile, build and set up the program by use of their environment XML file.

How to use create environments?

1. Add the *'environment.xml'*, in the same directory as the C++ program, named *'CarEnvironment.exe'*.
2. Open the *'CarEnvironment.exe'* program. This one will generate 150 *'.obj'* files, named *'xmlemXXX.obj'*, and store these in the *'Assets/Objects'* folder. The *XXX* is the number of the scene (ranged from 000 to 150), according to the information given by the xml file.
3. Verify the shapes of the terrains, shown by the C++ program (like picture below). Press the right and left arrow keys, to go from one terrain to another.

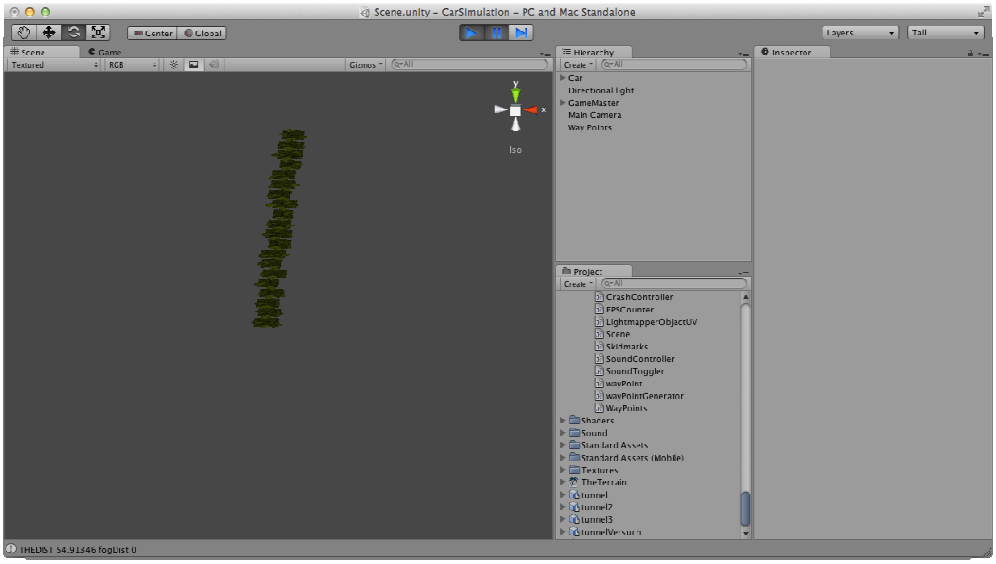


4. Close the program.
5. Locate the *'Scene.unity'* file inside the *'Assets'* folder.
6. Open the *'Scene.unity'* file (It will take a little while for Unity to import all scenes).
7. When it is done importing, Unity should show something like this.

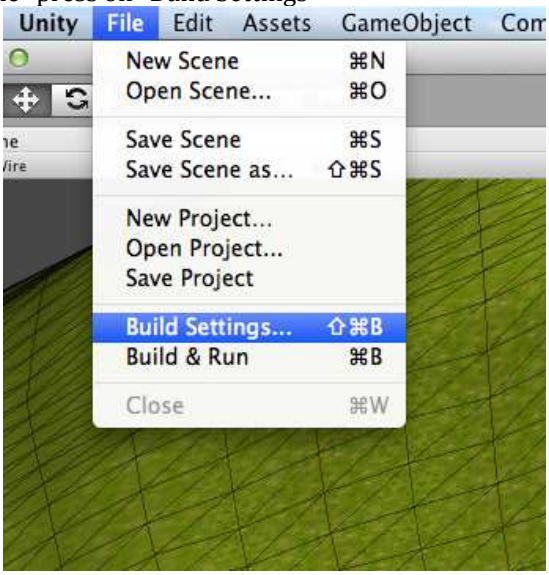


a. If you wish to check whether the environment, as it will look when running, press the **Play** sign and the **Pause** sign (designated by the arrows on the picture)

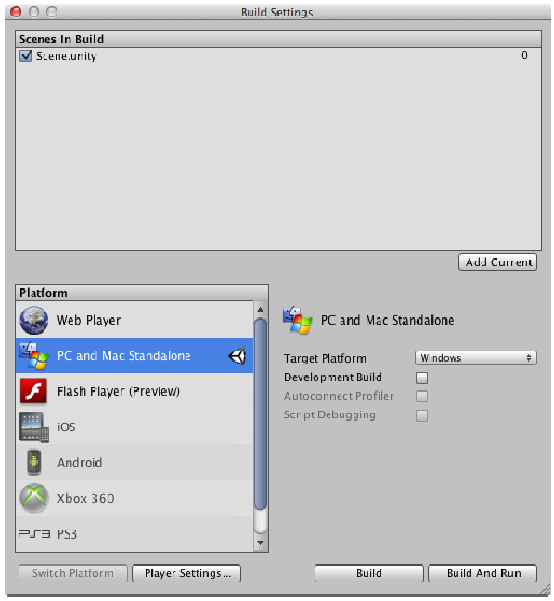
b. If everything works, the environment should look something like this



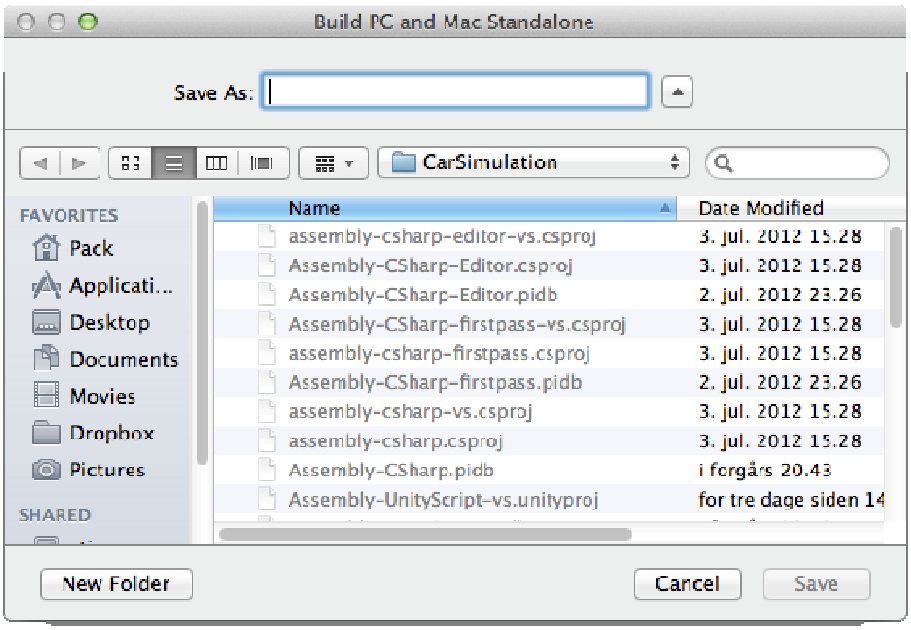
8. Under the menu “File” press on “Build Settings”



9. A window will appear (picture below)



10. Verify theat “Scene.unity” is checked
11. Verify that under “Target Platforms” it says “Windows”
12. Verify that “Development Build” is unchecked
13. Press Build



14. Give it the name you want, and put it in the directory you want
15. Press Save

How to use the Program?

1. Locate saved unity “exe” file generated
2. Make sure you have a file with the name “input.txt” in the same directory
3. Open the unity “exe” file
4. A black screen will appear. The game waits for a “start” command from the text file to start
5. Once the game is started you will be able to control the car with 4 different commands (please note that you can combine several commands per line):
    a. “left”
    b. “right”

c. “up”

d. “down”

6. According to car mode you can use the commands:

   a. If the car is on follow-track mode, the left and right commands are used to decide if the car should turn at a crossing.

   b. If the car is free-ride mode, then all commands are free to be used.

7. When done with the program, you will find the “logging.txt” file inside the directory where the unity “exe” file is

8. If you wish to edit things that have nothing to do with the environment (car view, default speed, car mode, fog density), you can just do so in the xml file, and the game will have those specifications upon next startup

Bibliography

[Bæ12] Jakob Andreas Bærentzen. Gel library. http://www2.imm.dtu.dk/projects/GEL/GEL-docs/index.html, 2012.

[Dig97] Polyphony Digital. Grand turismo. http://www.gran-turismo.com/, 1997.

[DSEW02] Darwyn Peachey Ken Perlin David S. Ebert, F. Kenton Musgrave and Steve Worley. *Texturing and Modeling, Third Edition: A Procedural Approach*. Morgan Kaufmann, 2002.

[Fil12] FileFormat.info. Wavefront obj file format. http://www.fileformat.info/format/wavefrontobj/egff.htm, 2012.

[Joy00] Kenneth I. Joy. Bernstein polynomials. http://idav.ucdavis.edu/education/CAGDNotes/Bernstein-Polynomials.pdf, 2000.

[MHDB11] R. J. Stone M. H. Depledge and W. J. Bird. Can natural and virtual environments be used to promote improved human health and wellbeing? 2011.

[Mic12] Microsoft. System.xml. http://msdn.microsoft.com/en-us/library/system.xml(v=vs.71).aspx, 2012.

[Mor12] Chris Morris. Road and path tool. http://sixtimesnothing.com/road-path-tool/, 2012.

[PM01] Yoav I H Parish and Pascal Müller. Ciyengine. http://www.vision.ee.ethz.ch/~pmueller/documents/procedural_modeling_of_cities__siggraph2001.pdf, 2001.

[QWN10] Olga Sourina Qiang Wang and Minh Khoa Nguyen. Eeg-based "serious" games design for medical applications. 2010.

[SO11] Nguyen MK. Sourina O, Wang Q. Eeg-based "serious" games and monitoring tools for pain management. 2011.

[TAM08a] Naty Hoffman Tomas Akenine-Möller, Eric Haines. 12.4.4 triangle meshes. In *Real-Time Rendering*. 2008.

[TAM08b] Naty Hoffman Tomas Akenine-Möller, Eric Haines. 6.3 procedural-texturing. In *Real-Time Rendering*. 2008.

[TS05] Playground games Turn 10 Studios. Forza motorsport. http://forzamotorsport.net/, 2005.

[Uni12a] Unity. Car tutorial. http://unity3d.com/support/resources/tutorials/car-tutorial, 2012.

[Uni12b] Unity. Fixed update. http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.FixedUpdate.html, 2012.

[Uni12c] Unity. Fog. http://docs.unity3d.com/Documentation/Components/class-RenderSettings.html, 2012.

[Uni12d] Unity. Input. http://docs.unity3d.com/Documentation/ScriptReference/Input.html, 2012.

[Uni12e] Unity. Mathf. http://docs.unity3d.com/Documentation/ScriptReference/Mathf.html, 2012.

[Uni12f] Unity. Monobehaviour. http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.html, 2012.

[Uni12g] Unity. Prefabs. http://docs.unity3d.com/Documentation/Manual/Prefabs.html, 2012.

[Uni12h] Unity. Rigidbody. http://docs.unity3d.com/Documentation/Components/class-Rigidbody.html, 2012.

[Uni12i] Unity. Transform. http://docs.unity3d.com/Documentation/ScriptReference/Transform.html, 2012.

[Uni12j] Unity. Unity publishing. http://unity3d.com/unity/publishing/, 2012.

[Uni12k] Unity. Vector3. http://docs.unity3d.com/Documentation/ScriptReference/Vector3.html, 2012.

[Wei12] Eric W. Weisstein. Rotation matrix. http://mathworld.wolfram.com/RotationMatrix.html, 2012.