FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Procedural Modelling Techniques to Configure Driving Serious Game Scenes

Pedro Miguel Cesário Rosa

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Rosaldo J. F. Rossetti, PhD (Assistant Professor)

July 2015

Procedural Modelling Techniques to Configure Driving Serious Game Scenes

Pedro Miguel Cesário Rosa

Mestrado Integrado em Engenharia Informática e Computação

Approved in a public examination by the Jury:

President: Ana Paula Rocha, PhD (Assistant Professor)

External Member: Luis Paulo Reis, PhD (Associate Professor)

Supervisor: Rosaldo J. F. Rossetti, PhD (Assistant Professor)

---

14 of July 2015

Summary

This dissertation has the objective of tackling the road safety problem in order to further prevent accidents and casualties for both drivers and pedestrians by creating a tool that is capable of loading real world data from user selected locations and render them in 3 dimensional models for further use in serious driving games. These models are then populated with pedestrians that walk around and the ability to drive a vehicle is given to the user. Also the tool should be flexible enough to allow the users to configure the different conditions such as weather, time of day, rendering options, vehicle damage and pedestrian density, in order to conduct studies on different conditions. The project should also be open source, so anyone can edit it and expand it their own way to suit their needs and conduct specific studies. It features Oculus Rift integration, which further extends the possibility of conducting studies to the human driver by giving the possibility to expand this integration to evaluate the driver's behaviours. Another important aspect is the possibility to export the entire procedurally generated scene to a 3D file format that can be edited by an external application.

By providing such tool for free, not only marks the beginning of an open source world 3D generator, but also a framework capable of allowing multiple different usages, such as conducting studies, building video game scenarios or be used as a learning tool by a driving school for instance. Hopefully this will increase road safety if used carefully as a serious tool.

To do so we'll use the game engine Unity 5 to develop the project, CGIAR-CSI to download elevation data, Google Static Maps for the satellite imagery, OpenStreetMap for the location data and everything else is built inside Unity. Also UnitySlippyMap, which is a world map that works with various tile providers was used and integrated on the context of this project to allow users to select a location inside Unity.

Along this document you will find a literature review on the topic, including related work and attempts to do similar projects followed by a comparison between other project and this one. The reader will also find details about development and the system's architecture as well as deep details about implementation. On the end you can find a few screen shots of the results of this project.

Keywords: Procedural Modelling, Driving Simulation, World Locations, Serious Games, Road Safety

Resumo

Técnicas de Modelação Procedimental para Configurar Cenas de Jogos Sérios de Condução

Esta dissertação tem o objetivo de tratar do problema da segurança rodoviária, a fim de evitar mais acidentes e mortes tanto de motoristas como de pedestres através da criação de uma ferramenta que é capaz de carregar dados do mundo real a partir de localizações selecionadas pelo utilizador e transformá-los em modelos tridimensionais para uso posterior em jogos de condução sérios. Estes modelos são então povoados com os pedestres que andam nos passeios e a capacidade de conduzir um veículo é dada ao utilizador. Além disso, a ferramenta deve ser flexível o suficiente para permitir que os utilizadores configurem diferentes condições, tais como o tempo, hora do dia, opções de renderização, os danos do veículo e densidade de pedestres, a fim de realizar estudos em diferentes condições. O projeto também deve ser de código aberto, para que qualquer pessoa pode editá-lo e expandi-lo para atender as suas necessidades e realizar estudos específicos. Possui integração Oculus Rift, que estende ainda mais a possibilidade de realização de estudos para o motorista humano, através da expanção dessa integração para avaliar os comportamentos do motorista. Outro aspeto importante é a possibilidade de exportar toda a cena procedimentalmente gerada para um formato de ficheiro 3D que pode ser editado numa aplicação externa.

Ao fornecer esta ferramenta de forma gratuita, não só marca o início de um gerador do mundo em 3D de código aberto, mas também uma ferramenta capaz de permitir diversos usos diferentes, tais como a realização de estudos, a construção de cenários de jogos de vídeo ou ser usado como uma ferramenta de aprendizagem por uma escola de condução. Esperemos que isto seja capaz de aumentar a segurança rodoviária, se usado com cuidado como uma ferramenta séria.

Para fazer isto vamos usar o motor de jogo Unity 5 para desenvolver o projeto, CGIAR-CSI para baixar dados de elevação, o Google Static Maps para as imagens de satélite, OpenStreetMap para os dados de localização e tudo o mais é construído dentro do Unity. Também é usado o UnitySlippyMap, que é um mapa do mundo que trabalha com vários fornecedores de “tiles” que foi integrado no contexto deste projeto para permitir que os utilizadores selecionem um local dentro do Unity.

Ao longo deste documento, irá encontrar uma revisão da literatura sobre o tema, incluindo o trabalho relacionado e tentativas de fazer projetos semelhantes, seguido de uma comparação entre outros projetos e este. Irá também encontrar detalhes sobre o desenvolvimento e a arquitetura do sistema, bem como detalhes profundos sobre a implementação. No final pode encontrar algumas capturas de ecrã dos resultados deste projeto e a conclusão que refere a satisfação dos objetivos e trabalho futuro.

Palavras-chave: Modelação Procedimental, Simulação de Condução, Locais do Mundo, Jogos Sérios, Segurança Rodoviária

Acknowledgment

I would like to thank Professor Rosaldo Rossetti, my supervisor, for all orientation and help along this dissertation. I also thank him for giving me the freedom to elaborate the main goals of this dissertation along him. This dissertation's idea began on a conversation with him about investigation tastes. We figured out that we had some in common, so he suggested me to create a dissertation project along him based on our ideas. Being so I could arrange the tools to use as well as add details and ask for possible and useful alterations.

I would also like to thank Professor Rui Rodrigues for letting me use his Oculus Rift while being supervised by his personnel.

Content

Introduction **1**
1.1 Motivations and Goals 1
1.2 Structure of the Dissertation 2
Literature Review **3**
2.1 Game Engines 3
2.2 Simulation Engines 4
2.3 Physics Engine 4
2.3.1 PhysX 4
2.3.2 Havok 5
2.4 Render Engine 5
2.5 Driving Simulators 5
2.5.1 Entertainment 6
2.5.2 Research 6
2.5.3 Training 6
2.5.4 The Latest Driving Simulators 7
2.5.5 Driving Simulator Peripherals 10
2.6 Traffic Simulators 14
2.6.1 Microscopic 15
2.6.2 Macroscopic 15
2.6.3 Mesoscopic 15
2.6.4 Hybrid 16
2.6.5 Nanoscopic 16
2.7 Procedural Modelling 17
2.7.1 Procedural Modelling and Games 17
2.7.2 Procedural Modelling Techniques 20
2.7.3 Procedural Modelling Main Usage 22
2.7.4 Extracting Rooftops 31

2.8 Tools to be Used ... 32
2.8.1 Unity 5 ... 32
2.8.2 UnitySlippyMap ... 32
2.8.3 Open Street Map ... 33
2.8.4 Google Static Maps API ... 33
2.8.5 SUMO Simulation of Urban MObility ... 33
2.8.6 CGIAR-CSI ... 33
2.8.7 7zip ... 34
2.8.8 Blender ... 34
2.9 Related Work ... 34
2.10 Summary ... 35

Methodological Approach and System Architecture ... 36
3.1 Problem to Solve ... 36
3.2 System Architecture ... 38
3.3 Unity3D + SUMO (Simulation of Urban MObility) ... 39
3.4 Development ... 40
3.5 Implementation ... 42
3.5.1 Main Menu ... 42
3.5.2 Main Menu Possible Configurations ... 42
3.5.3 UnitySlippyMap ... 43
3.5.4 Elevation Data ... 44
3.5.5 Location Data Parsing ... 45
3.5.6 Satellite Image Gathering ... 47
3.5.7 Terrain Elevation ... 48
3.5.8 Procedural Roads ... 50
3.5.9 Pedestrians ... 55
3.5.10 Procedural Buildings ... 58
3.5.11 Procedural Barriers ... 61
3.5.12 Procedural Land Uses ... 62
3.5.13 Procedural Amenities or City Furniture ... 65
3.5.14 Day and Night Cycle ... 65
3.5.15 Weather Cycle ... 67
3.5.16 Vehicle ... 68
3.5.17 Visual Effects ... 77
3.5.18 Procedural Scene Export ... 79
3.6 Summary ... 79

Results ... 81
4.1 Main Menu ... 82

4.2 Main Menu Possible Configurations 82
4.3 UnitySlippyMap 83
4.4 Elevation Data 84
4.5 Location Data Parsing and Satellite Image Gathering 85
4.6 Terrain Elevation 85
4.7 Procedural Roads 86
4.7.1 Road Enlightenment 87
4.8 Pedestrians 88
4.9 Procedural Buildings 89
4.10 Procedural Barriers 90
4.11 Procedural Land Uses 90
4.12 Procedural Amenities or City Furniture 91
4.13 Day and Night Cycle 91
4.14 Weather Cycle 92
4.15 Vehicle 92
4.15.1 Vehicle Selection 93
4.15.2 Vehicle Dynamics 94
4.15.3 Vehicle Illumination 95
4.15.4 Oculus Rift Integration 95
4.15.5 Vehicle Damage 96
4.16 Visual Effects 97
4.17 Procedural Scene Export 98
4.18 Summary 99

Conclusion and Future Work 100
5.1 Goal Satisfaction 100
5.2 Future Work 102

References 104

Table of Contents 112

List of Figures

Figure 1- iRacing gameplay [83] 7
Figure 2 - Project CARS Gameplay [84] 8
Figure 3 - City Car Driving [85] 9
Figure 4 - Logitech G27 [86] 10
Figure 5 - Playseat SV [87] 11
Figure 6 - Motion Base 301 [88] 12
Figure 7 - Round Screen Projection [89] 12
Figure 8 - Oculus Rift [90] 13
Figure 9 - Real Vehicle Simulator [91] 14
Figure 10 - Fuel Gameplay [92] 17
Figure 11 - Minecraft Gameplay [93] 18
Figure 12 - No Men's Sky in game footage [94] 19
Figure 13 - Example on how L-Systems work 20
Figure 14 - Hand fractal, just an example to better understand them [95] 21
Figure 15 - GML changing the parameter that controls the thickness [96] 22
Figure 16 - World Machine - Mountain Terrain Example [97] 25
Figure 17 - World Composer Example [98] 26
Figure 18 - Geometric Primitives - Floor Plan Generation [56] 28
Figure 19 - BuildR building generation example [99] 29
Figure 20 - Esri CityEngine - generated city example [100] 30
Figure 21 - System Architecture and Module Interaction 38
Figure 22 - Elevation tile vertices and polygon processing order 49
Figure 23 - Elevation Tile UV Mapping 49
Figure 24 - Procedural Road Vertices Placement 52
Figure 25 - Procedural Road Before and After Smoothing Operation 52
Figure 26 - Sidewalk and Step Generation - Vertices relation with each other 53
Figure 27 - Street Illumination 3D Models 54
Figure 28 - Pedestrian Animation Skeleton 56
Figure 29 - Procedural Building - Basement Generation 59

Figure 30 - Window Generation and Alignment 60
Figure 31 - Procedural Fence/Guard Rail Control Point and Vertex Generation 61
Figure 32 - Procedural Wall Control Point and Vertex Generation Orthogonal View 62
Figure 33 - Procedural FloorPlant Generation 63
Figure 34 - Tree 3D model on the left and bush 3D model on the right 64
Figure 35 – Garbage Bin 3D models 65
Figure 36 - Triumph Daytona 675 - Naturally Aspirated Engine (Adapted from [101]) 70
Figure 37 - Drum Brake Parts (Adapted from [102]) 72
Figure 38 - Disk Brake Parts (Adapted from [103]) 73
Figure 39 – Hydraulic Power Steering System (Adapted from [104]) 74
Figure 40 - Shape that the Headlights Projection should have 75
Figure 41 - Main Menu 82
Figure 42 - Advanced Options Screen 82
Figure 43 – UnitySlippyMap 83
Figure 44 - Elevation Data 84
Figure 45 - Bird Eye View of the Suzuka Race Track, Japan 85
Figure 46 - Highest peak of the Mount Everest, Himalayas 85
Figure 47 - Procedural Roads with Street Lights 86
Figure 48 - Procedural Roads Overlap Problem 86
Figure 49 - Procedural Roads Night Illumination 87
Figure 50 - Pedestrians Walking and Getting Hit By Vehicle 88
Figure 51 - Procedural Buildings, Reconstructing Avenida dos Aliados, Porto, Portugal 89
Figure 52 - Procedural Barriers - Guard Rails 90
Figure 53 - Procedural Vegetation - Gravel Path Going Through a Dense Forest 90
Figure 54 - Trash Bin Next to a Street 91
Figure 55 - Heavy Rain and Storm 92
Figure 56 - Vehicle Selection Screen 93
Figure 57 - Vehicle Rubbing a Guard Rail 94
Figure 58 - Vehicle Headlights 95
Figure 59 - Vehicle Damage System 96
Figure 60 – Beautiful Effect of the Sun Setting Behind the Trees Using Cockpit View 97
Figure 61 - Exported Scene Edition in Blender 98

List of Symbols and Acronyms

4WD	4 Wheel Drive
AI	Artificial Intelligence
API	Application Interface
ArcGIS	Aeronautical Reconnaissance Coverage Geographic Information System
ASCII	American Standard Code for Information Interchange
CGIAR-CSI	Consultative Group on International Agricultural Research – Consortium for Spatial Information
CPU	Central Processing Unit
DEM	Digital Elevation Model
FWD	Front Wheel Drive
GeoTIFF	Geospatial Tagged Image File Format (data integration)
GIS	Geographic Information System
GML	Generative Modelling Language
GPS	Global Positioning System
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDR	High Dynamic Range
HMD	Head Mounted Display
LDR	Low Dynamic Range
LED	Light Emitting Diode
LIDAR	Light Detection And Ranging
LOD	Level Of Detail
NASCAR	National Association for Stock Car Auto Racing
OSM	Open Street Map
PC	Personal Computer
PC	Personal Computer
RGB	Red-Green-Blue colour space
RPM	Rotations Per Minute / Revolutions Per Minute (Engine)
RWD	Rear Wheel Drive
SL	Style Library

SRTM	Satellite Radar Topology Mission
SUMO	Simulation of Urban Mobility
TCP	Transmission Control Protocol
TraCI	Traffic Control Interface
UV	U and V axis
XML	EXtensible Markup Language

Chapter 1

Introduction

Along this document, the reader will find the development of this dissertation step by step and in an understandable and simple way, so the reader can more easily read and follow this dissertation. Every significant step is organized in chapters with subchapters. Each subchapter is a specification of the subject about the chapter it belongs to.

This chapter introduces the reader to this project and to which its motivations and main goals are. In the end you can find a small overview on the structure of this document.

1.1 Motivations and Goals

Road safety is a very important problem needing further appreciation in what concerns prevention, especially when drivers are distracted and not paying attention to the road. Thus, any help attempting to study human drivers and their behaviour on important aspects will greatly contribute to road safety. In order to tackle this problem we propose a framework capable of loading a real-world location in 3-dimension scenes that will be flexible enough to help engineers as well as practitioners to conduct different kinds of studies.

From this framework it is expected that it can import real-world data and render it in a virtual environment in 3-dimensions by means of procedural modelling techniques that will then be populated by pedestrians that walk around and vehicles traversing the streets of a network. When all the location generation is finished this framework should allow the user to drive a virtual vehicle and collect metrics while the user drives around. It is possible to configure such metrics on the main menu. Using these metrics, for instance the time the user is not paying attention to the road should be a great contribution to study human drivers and their behaviour towards improving road safety somehow. Since the scene is going to be procedurally generated, it should

be a good feature to allow the possibility to export the entire scene, so it can be used by an external 3D application. The main contribution of this work is to introduce procedural modelling techniques to speed up the process of setting up driving simulation scenes to be used in serious game environments. Contrary to traditional driving simulation setups, which are generally very expensive to run and maintain, although much more limited in terms of experimentation capability, serious games have been introduced as an important resource to the study of subject players. Having so, this tool allows to save modelling time, because of the use of procedural modelling and to save money, because anyone will be able to run this tool on their own personal computers.

1.2 Structure of the Dissertation

Apart the introduction this document contains 4 more chapters. On Chapter 2 you can find the literature review on the subjects related to this dissertation exploring different approaches and theories, as well as explaining concepts and related work. At the end of this chapter you can find the tools that are going to be used, explaining why and their relation. On Chapter 3 there are further details about the project such as the problem to solve and how to, the system architecture and module relation and the work planning. Chapter 4 describes the details about the implementation regarding algorithms with example diagrams and included features and how they relate with each other as well as why they are needed and how they interact with the user. On Chapter 5, we can see the achieved and completed goals, results and future work, including possible enhancements and extensions.

Chapter 2

Literature Review

This chapter contains a few descriptions about the current state of the art. There are different approaches of the same problem by different authors. Starting by the root of this dissertation, the literature review starts by describing computer technologies that enhance user experience, covering driving simulators and their ultimate software relating them to their usages and peripherals that contribute to the immersion of the user. After, there will be covered traffic simulators and their types so the reader can have a notion on what they are and where to apply. There is also a coverage of procedural modelling applications before explaining how it happens so the reader can understand the possibilities before knowing what is behind them. There are also descriptions of the main applications of procedural modelling as well as existing tools to generate any kind of procedural mesh. In the end are presented the tools that will be used to elaborate this project and why they were chosen.

2.1 Game Engines

A game engine [1] is a software frame designed to create and develop video games. They provide rendering engines for both 2D and 3D graphics, a physics engine, collision detection algorithms, that might be included with the physics engine, collision response system, sound manipulation, multiple language scripting, animation engine, AI algorithms, networking capabilities, usually any of those are found in the language libraries, a good memory management, enables threading and a scene graph.

Using a game engine can reduce a lot the development time and cost, because the main core of the game is already implemented by the game engine's package, which mean that allows the creation of multiple game types as well as exporting them to different platforms with just a click.

So, a game engine is a tool that integrates multiple different engines (render, physics and so on) in a single tool.

There are a lot of game engines to choose from. Some of them only have a 2D rendering system, others have just the 3D and others have both. Different game engines often have different characteristics, performances, pros, cons and ease of use. The most evolved game engines are usually paid, but there are a few that are free, or at least have a free license like Unity3D.

2.2 Simulation Engines

A simulation engine can be almost any software that tries to recreate something that happens in real life, usually physically quantified. They are also very specific, providing only the tools meant to simulate a certain range of conditions, for instance a driving simulator is not meant to simulate water behaviour, but is meant to accurately simulate a vehicle's behaviour. That's why they are usually integrated into game engines, so that it can simulate multiple different and distinct behaviours. Having so, the classification on what can be a simulation engine or not is quite big.

There are thousands of software tools that recreate real life experiences.

2.3 Physics Engine

A physics engine [2] is a computer software that provides an approximate simulation of many physical systems, such as gravity, rigid body, soft body, fluid dynamics, collision detection and collision response. Having so they often integrate part of a game engine as a middleware to run the game with a physical simulation in real time. They have use beyond game engines, such as realistic simulation in serious software to study bridges, car deformation, buildings, tyre models, suspension models and the list goes on. Basically, any kind of real life that can be physically quantified can be done computationally with a physics engine.

The most used physics engines are Havok and PhysX

2.3.1 PhysX

PhysX is a scalable multi-platform game physics solution supporting a wide range of devices, from smartphones to high-end multicore CPUs and GPUs. PhysX is already integrated into some of the most popular game engines, e.g. UE3/UE4. PhysX also enables simulation - driven effects like Clothing, Destruction and Particles[3].

2.3.2 Havok

Havok's modular suite of tools and technologies put power in the hands of creators, making sure they can reach new standards of believability and interactivity, while mitigating the overall cost and risks associated with creating today's leading video games and movies.

Fully multithreaded and cross-platform optimized, Havok's technologies offer full support for leading game platforms[4].

2.4 Render Engine

A render engine [5] generates an image from a 2D or 3D model, by means of computer programs and algorithms. A scene file, also called scene graph contains object in a strictly defined language, that contains geometry, viewpoint, textures, lights, and shading information, that describes the scene. The renderer loads that information and calculates every scene polygon, with respective lighting and textures.

Some render engines also have a few algorithms that recreate light behaviour. One example is RayTrace, where the light is calculated through a ray that is casted from a pixel of the screen into the scene, hitting and reflecting on objects until its energy is near zero. This algorithm can create extremely realistic images. A good simulation of the light can be used to conduct studies on object materials or even create after effects on a movie.

2.5 Driving Simulators

A driving simulator is a very refined and complex system that recreates a real vehicle driving experience in a computerized environment that uses a graphical library to represent the graphical environment and code that is divided in a few modules to simulate the behaviours. Some of this modules may act inside a general physics engine such as the tire module that recreates the behaviour of a real tire, suspensions, engine and braking system, real-time telemetry that couples all the information and displays in a graphical way how the parameters are working, visualization module and so on. The content of a driving simulator could be a thesis on its own[6].

There are also three types of driving simulators. One is for training, the other is for entertainment and the third for research. All can be used to analyse people behaviour and create statistics on certain conditions[7].

2.5.1 Entertainment

This kind of driving simulator is meant for amusement, to present the player a challenge on beating a lap time or following a car or even just driving around on a big fantasy city making jumps and escaping the police[8].

Diving simulators for entertainment should not be confused as just driving or racing games. Nowadays there are a few platform dominant driving simulators. Although they are video games, they recreate real environments and the car's behaviour so well that they are used by real race drivers as a training tool at the comfort of their homes.

This kind of driving simulators are widely commercialized and sold on an available disk to anyone.

2.5.2 Research

A driving simulator can replicate millions of controlled and random situations to challenge the user to test his abilities behind the wheel as well as his reaction to a certain situation. Being so, it can have a massive impact on road safety by conducting studies on real drivers and retrieve information about their reactions.

There are hundreds of companies and industries that use these simulators. One good example are the vehicle manufacturers, which build and recreate the planned car on the simulation in order to test it so it can be approved for the production line. By doing so they can measure and analyse the car's behaviour and crash test safety in many possible conditions in a cheap, fast and safe way.

Another example is their usage on civil engineering, where a specific road can be tested before making any changes to the real area. Instead they alter the environment on the simulation and study how the driver reacts to the changes made [9].

2.5.3 Training

Training driving simulators are really serious simulators that can't be considered video games, they are very expensive and meant to just one purpose, to train a certain topic, offering the user a precise goal. That's why they are used on driving schools, police academies and so on. As we can see they are mainly used by entities that require serious and sometimes flat-out driving, as well as data retrieval and person behaviour analysis.

The main difference between the entertainment simulators is that in some cases this simulators are implemented on real cars that have their wheels attached to hydraulics that react to the simulation, the steering wheel, pedals and gear box are all connected to the simulator and in front of them a big curved screen.

This simulators are mainly for training and behaviour analysis to collect data on how people react in some driving situations[10].

2.5.4 The Latest Driving Simulators

The following subsections describe a few of the latest driving simulators. Although there are a lot more, these are just an example of the very best in the market.

Figure 1- iRacing gameplay [83]

IRacing is the most advanced driving simulator on PC. In what comes to graphics it might not be the most beautiful game, but it sure does is perfectly when it comes to driving. Players can experience an extremely realistic and detailed racetrack, modelled by professionals with data acquired through location laser scanning, which means that every bump on the real world is also on the game's racetrack, as well as an extremely realistic vehicle behaviour and damage system [11].

It is available to anyone, although it is paid.

This racing game simulates so well the real experience, that real race drivers use it to train.

"iRacing is the most modern racing simulation ever created. Every inch of every track is modelled perfectly. I've used iRacing to learn new courses such as Virginia International Raceway, or to keep the rust off at tracks such as Infineon Raceway. For the hardcore sim racer, this is your dream simulation. For the actual real-world racer, this is your 'at-home' test vehicle. Acclimate yourself with a new course or learn something new on an old favourite. Even a rookie

in the sim racing world can get up to speed fast within the iRacing system" – says Dale Earnhardt, Jr. 2-time NASCAR Nationwide Series Champion (1998, 1999)[12].

Figure 2 - Project CARS Gameplay [84]

Project CARS is the most recent racing game presenting innovative features. One of them is the capability to simulate a race day on a certain date, this is, the game can retrieve weather information [13], event type, time of day and the cars that raced that day, from a database recreating those conditions in the actual gameplay.

The game also has an extremely precise physics engine, including an extremely advanced tire model [14], suspension model, car handling, it recreates almost any possible racing conditions, such as heavy storm at night or day and any other endless combinations. The car also behaves differently depending on the track's condition, rain, haze, water poodles, small debris affect the vehicle, forcing the player to adopt a different racing strategy. This game does not only simulates a vehicle's handling on a certain weather, it also simulates a real driver's career making the player feel like a real driver, by reading e-mails, signing contracts and moving on up to the top classes [15].

Figure 3 - City Car Driving [85]

City Car Driving was designed to help users feel they are actually driving a car in a big city or in a country under varying conditions.

It is also a traffic simulator, because the vehicles behave themselves in a very realistic way respecting each other and real traffic rules. Some AI vehicles are more aggressive than others, they overtake, they have a target destination, they sometimes crash and the other vehicles have to respond to that situation.

This simulator is specially meant to be played with a gaming steering wheel, it is available to anyone, being an exceptional training tool for people who are attending driving lessons or even for regular day drivers, since it allows the training of the basic handling and feeling of a car, implements road rules such as road signals, simulates driving on different conditions of weather and time of day, it allows users to train parking manoeuvres and a wide set of situations by defining specific goals for the user to achieve [16].

Although City Car Driving is both a driving and traffic simulator some people use it as a game, because it is possible to turn off traffic rules and drive flat-out just for amusement.

Comparing to SUMO, City Car Driving has ad-hoc well defined cities and traffic routes, so the AI does not need to adapt to the current scene because they are already programmed to behave properly in it. SUMO on the other hand is able to load any kind of road network, which makes it possible to go to OpenStreetMap and download a road network and then import it to SUMO [17]. So the driver agents have to adapt to the current conditions of the road, the traffic ahead and sideways.

2.5.5 Driving Simulator Peripherals

Any kind of computer program needs an input from the user to make something happen. Driving simulators are no different, they need to have some sort of input peripheral so the user can control the simulation in an allowed and controlled way.

These input peripherals play an extremely important role in the immersion the user feels when using the simulator. The better the peripheral and its utility on a specific simulation the better will be the immersion of the user and the better will be the results of the simulation. Some of these peripherals also work as an output that transmits, somehow, useful information to the user that enhances the realism of that simulation making the user feel even more immerse.

There is a huge variety of peripherals ranging from a few euros to thousands as well as suitability, shapes, function and quality. The following peripherals are just a few of the whole range, but they are the most commonly used on driving simulators.

a. Steering Wheels

Figure 4 - Logitech G27 [86]

Steering Wheels play the most important role in driving simulation. They make the driver feel like he is holding a real car's steering wheel. They usually come with a set of two pedals, but some of them include the third pedal for the clutch. Another item of the package is the gear box that in some cases comes attached to the steering wheel like rally style handles or formula 1 pads, and in other cases it comes in a separate part that can be attached to the users chair for instance. This kind of peripheral is the most suitable to have at home, because it can be attached to a table and the price range goes from 50€ to around 400€.

Some of the steering wheels come with an intelligent system of Force Feedback[18]. Force Feedback is a system that generates forces on the steering wheel that depend on the vehicle the user is driving, reacting to every road bumps, lateral forces, impacts and so on. This system simulates every force applied on the vehicle of the simulation and passes that information to the

engines that generate force on the steering wheel. Nowadays some of the most recent pedals also have the same system so the user can also feel the experience on his feet.

Having such reaction from the controller makes the user feel more connected to the simulation.

b. Vibration Chairs

Vibration chairs provide a comfortable and usually sporty driving position and have a set of vibration engines distributed along its shape. The sensors react to the vehicles vibrations on the simulation and reproduce an adequate vibration so the user can feel it on his body. This makes the user feel like sitting in a real vehicle.

Figure 5 - Playseat SV [87]

The size of the chairs vary from manufacturer to manufacturer as well as the functionalities.

c. Force Dynamic Simulator

A force dynamic simulator is a platform that comprises a series of hydraulics that are connected to the simulation. These hydraulic systems are connected to the platform containing a chair, a steering wheel and a screen and they react to both acceleration and level differences. Basically the system detects the car's orientation and recreates the same orientation using the hydraulics by moving the whole group up or down independently for each hydraulic. This simulates both acceleration and inclinations and a few of the peripherals have a rotating platform instead of a fixed one like the example below so the whole platform rotates towards the vehicle direction.

Figure 6 - Motion Base 301 [88]

These kind of peripherals are an extremely immersive gadget, because they make the user feel every bump like in real life, all braking and acceleration, all lateral forces, all front and backward forces. This is an excellent tool to train race drivers and police forces.

d. Big Round Screen

A big and round or split group of flat screens are a must on a realistic driving simulator. Their objective is to cover a larger field of view to make the user feel closer to the vehicle he is driving, forcing him to look around in order to see the surrounding environment[19].

Figure 7 - Round Screen Projection [89]

e. Surround Sound System

The sound is a human fundamental navigation and information gathering system. The human brain can capture stereoscopic sound through the ears, giving the sense of distance and position. This feeling of depth is given by the difference between the time of the same sound reaching one ear and the other plus the intensity. For instance, if a sound feels louder on the left ear, it is almost certain that it comes from this side.

Being so, a three dimensional sound representation needs to exist to create the illusion of a 360º degree sound source. When a simulator uses a surround system, either 5.1 or 7.1 it creates the sense that the user is in the middle of the action. The user can capture surrounding sounds to know if there is traffic behind or on his sides.

Also, the sound can be considered one of the most propitious sources to produce alarm sensations. This can be useful to study human reaction on critical situations.

f. Head Tracking

Head tracking is the ability to follow the human head's movement. Nowadays developers are investigating and using Oculus Rift[20] as a head tracker. There are a lot of advantages on integrating this in simulation frameworks, because it is a device that tracks the free movement of the head. It also provides a stereoscopic 3D vision of the scene, due to the existence of two miniature screens inside it that represent slightly different images for each eye. In addition it completely isolates human vision from external factors meaning the user can only see the screens and nothing else.

Figure 8 - Oculus Rift [90]

This can conduct the most realistic studies, because when the human eye is completely isolated from the external factor the only focus is the simulation being displayed on the screens, added to the extremely precise head movement tracking and stereo 3D, so the user is deeply immersed on the simulation (the user actually feels connected and part of the scene). Studies prove that people that use Oculus Rift tend to make similar movements facing real world experiences.

g. Real Vehicles

Real vehicle simulators surround the user with a real car and controls, realistic feelings, but in a closed and controlled environment with a big screen surrounding the vehicle. This simulators also have realistic Force Feedback and take advantage of the car's suspension to simulate the roads bumps, lateral force, forward and backward forces, engine vibration, realistic dampening and shock absorption[21]

On the other hand, these kind of simulators are usually very expensive and hard to maintain.

Figure 9 - Real Vehicle Simulator [91]

h. Extreme Experience

In order to obtain the most realistic experience none of the above are enough by themselves. Therefore, by combining multiple of the previous peripherals it is possible to make the experience rich enough to all human senses and immerge the user deeply in the simulation.

But, of course it depends on the budget one has and on the goal of the study to conduct.

2.6 Traffic Simulators

Traffic or transportation simulation is the mathematical modelling of transportation systems through the application of computer software to improve the planning, design and operation of transportation systems. Simulation of transportation systems is an important area of discipline in Traffic Engineering and Transportation Planning today and it is used by multiple transportation agencies, academic institutions and consulting firms to aid in the management of transportation networks [22].

Simulation in transportation is extremely important, because it can simulate very complex models and present results that can be analysed according to the simulation in matter reaching major conclusions on the planning of transportation routes and so on.

This kind of simulation can also be useful to analyse traffic behaviour in a certain road network, as well as possible improvements to be implemented on a real road network to make the traffic flow better.

There are a few kinds of traffic simulators. They can be microscopic, macroscopic, mesoscopic, hybrid and nanoscopic [23].

2.6.1 Microscopic

"A microscopic model of traffic flow attempts to analyse the flow of traffic by modelling driver-driver and driver-road interactions within a traffic stream which respectively analyses the interaction between a driver and another driver on road and of a single driver on the different features of a road. Many studies and researches were carried out on driver's behaviour in different situations like a case when he meets a static obstacle or when he meets a dynamic obstacle."[24] "Among these, the pioneer development of car following theories paved the way for the researchers to model the behaviour of a vehicle following another vehicle in the 1950s and 1960s." [25]

An excellent example of a microscopic traffic simulator is SUMO (Simulation of Urban Mobility), which consists on a platform developed by the German Aerospace Centre (DLR) that has all the required tools to implement simulations. It can import any kind of road network, even those imported from OSM (OpenStreetMap) and generate a road network according to that data. By processing the network it can generate traffic, traffic routes, traffic lights, pedestrians and different types of vehicles with different characteristics. A great advantage of this software is that it is extremely portable, can be integrated in any external application, because, thanks to TraCI[26], which is a middleware layer that creates a TCP channel and has a very well defined protocol, SUMO can act like a server and be controlled by the external application[27].

2.6.2 Macroscopic

A macroscopic traffic simulator, in contrast to the microscopic one, is intended mostly to the study of transportation planning along a much larger network. This kind of simulator does not show every vehicle on the road nor the streets and lanes, instead it uses a wide map, like the USA and shows results such as efficiency, times, traffic density and flows and so on, on other words, any kind of major information representing a whole and not a single individual [28].

2.6.3 Mesoscopic

A mesoscopic traffic simulator is considered to be in between a microscopic traffic simulator and a macroscopic traffic simulator, because it provides an intermediate level of detail like, for instance, individual's description, but not their interactions. On other words the information is more detailed than a macroscopic traffic simulator and less detailed than a microscopic traffic simulator.

This model describes the traffic entities at a higher level of detail, but their behaviour and interactions are presented at a lesser level. It shows information about traffic density, flow, velocity, average velocity per individual, the number of vehicles on a lane, but never information about a specific individual, so we can say that a vehicle is on a road, but we don't know on which lane, it's speed, it's direction, its own information, and this is because the vehicles are clustered in groups. The types of groups vary from simulator to simulator and they all have their strengths and weaknesses.

The main application of the mesoscopic model is where the detail of the microscopic model is needed, but not all of it. So instead of having a well detailed and computationally heavy road network on a microscopic simulator, the mesoscopic simulators [29] can be used.

2.6.4 Hybrid

A hybrid traffic model is a combination of other models. It combines two or more models into a single model with the advantages and disadvantages of each, but with special focus on the limits of each one individually in order to take maximum profit of each model.

The most used hybrid simulators are mesoscopic-microscopic simulators that allow detailed microscopic simulation of specific areas of interest while simulating the remaining areas in a lesser level of detail, on a mesoscopic level.

This approach comes in handy when there is the need to study a very large road network that is very heavy for the microscopic traffic simulator and requires constant calibration. For instance, if we have 3 districts in a row and we want to study the traffic behaviour in the central one, both the left and right side districts can be simulated on a mesoscopic detail, while the central one can be simulated in a microscopic detail, because it is the main focus and the other districts don't matter so much in a microscopic way [30].

2.6.5 Nanoscopic

A nanoscopic traffic simulator is even more detailed than a microscopic simulator. It is only capable of supporting small road networks and represent them with a huge amount of detail and information. It is the most suitable model to simulate accidents due to its large level of detail [31].

It can also be integrated with other traffic simulators. Since this is just a model it can be programmed separately from the traffic simulators. These models can be applied in building a nanoscopic simulation based on the four wheels of a car taking into account angular velocity, drag, traction and slip, for instance. Having this, the microscopic simulator can create cars that integrate nanoscopic model making them acting as desired [32].

2.7 Procedural Modelling

Procedural modelling is the name for a computer graphics techniques that creates a 3D model or texture by means of a computational procedure that implements a computer graphics algorithm to generate a model. This is usually performed in runtime of an application or done a priori to generate a model that is stored for future use. The resulting model depends on the inputs of the procedure, on the sets of rules the algorithm obeys and also on the goal of the model [33].

The biggest advantage of procedural modelling is the fact that it is a procedure that generates the content with no human intervention and, if needed, in runtime. Being so, it is possible to generate an endless terrain that keeps being created as the user moves around (in the case of a game).

Another advantage is that it saves storage space and, since the content is runtime generated, in some cases, there is no need to store the generated content, because it can still be generated again by the same rule set. This allows newly generated data in runtime and no data at all on the storage.

A disadvantage is the fact that sometimes the content generation might not be perfect and it still needs human intervention. For instance, when loading data from the real world to represent it on 3D by means of a procedural modelling algorithm sometimes there is lack of information and the algorithm cannot find a consistent solution for the problem.

2.7.1 Procedural Modelling and Games

Procedural modelling has its main usage on game scenario generation. It is a resource of game scene generation since the first games appeared and by the years it is getting more and more complex, capable of achieving extremely realistic and detailed scenes.

Figure 10 - Fuel Gameplay [92]

Fuel is an open world racing game set on a post-apocalyptic world featuring extreme weather conditions due to the global warming and day-night cycle. The distinctive feature in this game is the huge map, extending over 14400 square kilometres in size, allowing the player to explore it freely without loading times.

The Executive Producer David Brickley explain that "The technology behind FUEL is based on a concept to procedurally generate data on the fly instead of just loading and decompressing it. Today's hardware now has enough parallel processing power to procedurally create high quality environments, in real-time, while the player is moving around the world. Generated environments are optimized and arranged to get the best performance out of the machine's hardware. As a result, the player gets the maximum amount of detail and quality of environments displayed on screen and a 40km draw distance. Generated doesn't mean random though, it's the same result every time, and owing to the locations we chose is in many cases accurate to satellite data, we just took some liberties for the sake of design, i.e. putting all our favourite locations in one map."[34].

Figure 11 - Minecraft Gameplay [93]

Minecraft is a sandbox open world video game that offers players the ability to build anything they can think of by simply collecting cubes from the map. The generated world is entirely composed by cubes, called voxels. A voxel is a 3D cube that is usually on discrete positions of a 3D grid and is used to create procedural modelled scenes. By clustering many cubes one can create objects such as mountains, lakes, trees, buildings and so on with extreme ease. These cubes act as large scale atoms, enabling the generation of procedural terrains with holes and caves with no gaps between cubes[35].

In Minecraft the voxels are not smoothen, making the scene look cubic. In other cases the voxels can be smoothed to create a continuous shape.

Minecraft is an award winner because of its usage of procedural modelling. Whenever the game has to generate a new world, it calls upon an algorithm. This algorithm will output a pseudo-random value that is then used to determine what the world will look like. However, the algorithm

will always end up with the same value if the starting point (seed) used is the same. Seeds exist, to easily generate entirely different worlds from a single value [36].

In fact, Minecraft can generate a map without size limitations. This is possible, because the map is generated in chunks, when the player explores further a new chunk is generated using the same seed and the relative position to other chunks. This guarantees that if the player moves too far from a chunk it is destroyed, but when the player moves close enough to it that chunk is generated again with the same seed, meaning that chunk is equal to the one destroyed before. Having these there are no memory concerns, but it needs to be kept in mind that any alteration made to a chunk's terrain by a player will be destroyed with the chunk and not recovered.

Figure 12 - No Men's Sky in game footage [94]

No Man's Sky is an exploration game that allows the player to travel through a galaxy flying seamlessly between different planets and stars, seen on the sky. It is about exploring and cataloguing unique forms of life from planet to planet, that the player can share with other players. But of course, there are dangers, both in the open space an enemy player spaceship can appear to kill the player and even on land wild creatures can appear. The player can also evolve by acquiring new and better equipment by trading resources gathered on planet's surfaces [37].

There is an infinite number of planets, creatures, spaceships and every planet is different from the others. This is done using only procedural modelling technologies. Everything the player sees is procedurally generated in runtime, every creature, rock, grass field, texture, ships and planets (there are no pre-defined 3D models) [38].

The developer states that the world is generated in runtime using voxels. These voxels are smoothed to get a smooth surface and are merged with each other. This mesh is showed with different levels of detail according to the speed the player is traveling and his distance to the surface. Every level of detail is also procedurally generated to ensure a higher frame rate and the usage of less memory. The developer also states that the player will never be able to travel faster

than the algorithm's generation time, and the player travels really fast sometimes, which makes these algorithms very efficient.

Every planet is generated using a unique seed, in a similar way than Minecraft. This ensures that every planet is different from the others.

On land (planet surface), the player will find and catalogue creatures. Creatures are procedurally generated using a reference prototype model. This prototype does not have textures and it is a more simplified mesh. The creature generation algorithm receives several parameters as input, including the prototype mesh and generates a new mesh derived from the prototype. This guarantees creature specie similarity, but at the same time uniqueness within the same race and ensures that all races derived from the same prototype are different from each other. Also the prototypes range from very small creatures to colossal beasts taller than trees. The same happens with the spaceships. There is a prototype that is used to generate more complex spaceship models different from each other. The trees, the rocks, the grass, and even the smallest details are all generated in runtime, procedurally.

2.7.2 Procedural Modelling Techniques

Procedural modelling can be achieved by any kind of polygon generation algorithm. Nevertheless, there are a few techniques that provide a decent blend of realism and performance, being used with great frequency by professional developers. The sections below detail the most common techniques used on procedural modelling.

a. L-Systems

An L-System, also called Lindenmayer System, was developed as a mathematical theory of plant development. The original focus was on plant topology by analysing neighbourhood relations between plant cells or higher structures. They consist on a finite set of characters belonging to an alphabet that are arranged in arbitrary length sequences to form strings. Each character is associated with a rewriting rule that is defined by a grammar.

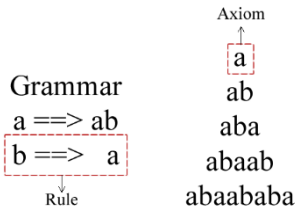

Figure 13 - Example on how L-Systems work

These rules mean that any occurrence of "a" in a string will be replaced by "ab" and "b" by "a". Now the string needs to be converted to a three dimensional model using a technique called turtle interpretation. This technique is based on an imaginary turtle that walks, turns and draws

according to the given instructions. The turtle has a well-known and defined 3D position and a heading vector that points towards the direction of the movement. Each character of the string is interpreted as a command by the turtle. There is a variant of the L-Systems called bracketed L-Systems that provides two extra characters that are usually square brackets ("[","]") that allows the push and pop of the turtle's direction, position and other information in order to enable the generation of branching structures [39].

More information regarding L-System can be accessed in the work performed by Przemyslaw Prusinkiewicz [40].

b. Fractals

A fractal is a natural phenomenon that can be represented by a mathematical function capable of generating a repeating pattern. This pattern can be displayed at every scale. So no matter the scale, the pattern will still be noticeable, making the fractal scale invariant. When the replication is the same no matter the scale the fractal is called a self-similar pattern. It generates the same pattern at smaller and smaller scales as the recursion progresses. Another type of fractals are the Statistical fractals where the pattern repeats stochastically so numerical or statistical data are preserved across scales. This last one can add a little randomness to the fractal meaning that it is not mandatory that the fractal pattern strictly repeats. This is useful to generate flora. This fractals can generate a tree by adding the log and fraction it on each iteration until the leaves are reached (usually a predefined depth).

Figure 14 - Hand fractal, just an example to better understand them [95]

As we can see on the above picture, we have a hand, and on the top of each finger the fractal generates another hand and so on recursively until a defined limit/depth is reached. In this case it is a self-similar algorithm, because the pattern repeats iteration after iteration getting smaller and smaller at each one.

Now that we know what fractals are, there must be a way to generate them. This way is the mathematical function mentioned before. L-Systems can create the exact same patterns in some cases, meaning they are fractal generators. They use a grammar to modify the string that will be processed by the turtle interpreter, but since each character represents a command and is disposed

on a pattern style, the turtle will build that same pattern as well. On the image of the hand the grammar should be specified to determine where to fraction the pattern by specifying where the pattern should be continued, in this case the finger tops.

c. Generative Modelling Language

Nowadays most of the 3D file extensions describe its objects in terms of geometric primitives and respective geometric operations. The term Generative Modelling brings a different approach. Basically the main idea is to replace the 3D objects descriptions on the file by operations that will generate those objects when executed. These operations may be called rules. This means that an object is described by a sequence of processing steps rather than vertices and geometric operations. Other properties of generative modelling are the fact that it is possible to group operations in higher-level operations leading us to Style Libraries. A SL is a library that contains more or less complex operations that differentiate one type of 3D model from another. For instance, one can have a library for generating trees (it contains operations that lead to the generation of a tree) and another for generating buildings [41]. The advantage is making possible to implement a programing language that allows the generation of 3D models by just executing a sequence of commands called Generative Modelling Language. GML provides a set of operations that the programmer can combine to generate 3D models. This not only is a key for procedural modelling, but it also allows simple manipulation of 3D objects in run time by simply adjusting parameters and running the commands again[42][43].

Figure 15 - GML changing the parameter that controls the thickness [96]

2.7.3 Procedural Modelling Main Usage

The sections below describe the main usage of procedural modelling techniques, including what kind of techniques are used and fit best to generate a specific kind of 3D mesh. Also there are a few references about the evolution of some applications.

2.7.3.1 **Terrain**

Terrain procedural modelling might be the main usage for procedural modelling, because as seen above most of the applications that need a huge terrain, in order to save storage and development time it is more efficient to generate it in real time recurring to terrain generation algorithms rather than storing the entire terrain [44].

A height map is a two dimensional grid that represents the elevation values of a terrain, generally used as a basis. There are many procedural algorithms for creating height-maps. Among the oldest algorithms are the subdivision based methods. A coarse height-map is iteratively subdivide each iteration using controlled randomness to add detail. Another algorithm is a variant of the mid-point displacement method, in which a new point's elevation is set to the average of its corners in a triangle or diamond shape and then a random offset is added. The offset's range decreases each iteration according to a parameter that controls the roughness of the resulting height-map. Nowadays, height map generation is often based on fractal noise generators such as Perlin noise that is capable of generating noise by sampling and interpolating points in a grid of random vectors. Rescaling and adding several levels of noise to each point in the height-map results in natural, mountainous-like structures. These height-maps can be transformed further based on common imaging filters or on simulations of physical phenomena, for instance erosion, because a height-map can be displayed on a grey-scale image. Thermal erosion diminishes sharp changes in elevation, by iteratively distributing material from higher to lower points, until the talus angle. The talus angles is, for instance, the maximum angle of stability for a material such as rock or sand. Erosion caused by rainfall can be simulated using, for example, cellular automata, where the amount of water and dissolved material that flows out to other cells is calculated based on the local slope of the terrain surface. Benes and Forsbach terrain model consists of stacked horizontal slices of material, each having an elevation value and material properties like density. It's a trade-off between the limited, but efficient height map structure and a full voxel terrain. The model also allows for air layers, thereby it supports cave structures. While these erosion algorithms add much to the believability of mountainous terrain, they are also very slow, having to run for hundreds to thousands of iterations. Recent research has focussed on interactive erosion algorithms, often by porting algorithms to the GPU. Usually on video game, when the terrain needs to be very big and detailed, the developer adds a progressive LOD (level of detail). The goal is to generate the terrain detail according to the distance to the camera. When the camera is close, a portion of terrain gets detailed to the maximum procedurally. When the camera starts getting further away, the terrain progressively decreases its detail until only the greatest elevations, such as mountains, are visible. This technique increases performance, by not processing irrelevant information while it is not visible and only processing it when it is needed. The basic noise-based height-map generation delivers results that are fairly random and natural. Usually, users control the outcome only on a global level, often using unintuitive parameters. To overcome this problem and allow a more easy terrain configuration several researchers have addressed this issue, such as Stachniak and Stuerzlinger [45] that proposed a method that

integrates constraints into the terrain generation process. It uses a search algorithm to find an acceptable set of deformation operations that conform to the constraints to apply to the random terrain.

Gamito and Musgrave propose a terrain warping system that results in regular, artificial overhangs. A recent method by Peytavie provides a more complex structure that considers the existence of different material layers that support rocks, arches, overhangs and caves. In most of the cases their resulting terrain models are usually visually very appellative and natural [46].

a. Height Maps

A height map is, in terms of computer graphics a raster image where each pixel of this image has a colour between black and white inclusive that represents the height of that point. The whiter the pixel the higher the point. The image is used to create a 3D mesh where each vertex has the same height of the pixel or average of pixel heights for that corresponding mesh point. Being so, and repeating the procedure for every mesh vertex one obtains a 3D mesh corresponding to the height map used. The real height of the 3D mesh is scaled, because since the height map is a black and white image the pixel's values are just between 0 and 1, so the values need to be scaled. Height maps are not just for creating terrains, they also apply on computer graphics materials by acting as a displacement map for a texture to displace the position of the texture points in order to give the sense of depth.

There are a lot of tools that can generate height maps, because they are simple grey-scale images. For terrain height generation developers usually use Perlin Noise, fractals and L-System to paint the height map's image. This produces a pseudo-random height map with more or less smooth black and white transitions in order to create mountains or rough planes. Basically the result of the previous techniques give a realistic feeling to the terrain.

Height maps do not always need to be generated using the referred techniques, they can also be calculated by physical means such as LIDAR, stereo photogrammetry and so on. Data obtained by these means is very expensive and extremely precise, being so it is used by NASA to create a full earth scale height map called Digital Elevation Model (DEM). This model has different resolutions for the images, in other words the data contained on a pixel of the image can have different corresponding distances on the real world. Therefore one pixel can average and cover a certain number of square meters. CGIAR-CSI distributes this kind of data for free, but of course it is not the best data that exists, but it still manages to cover around 80% of the entire world with a resolution of 90 meters [47]. It also provides different downloadable files such as GeoTIFF and ArcInfo ASCII. GeoTIFF is an image format that contains additional information for each pixel, but it needs a special interpreter to open and manage it. On the other hand ArcInfo ASCII is a text file that contains the elevation of each point. It makes it extremely simple to process and manage.

Having so, it is possible to process real world data and create height maps with it, to apply to terrains.

b. World Machine

Figure 16 - World Machine - Mountain Terrain Example [97]

World Machine [48] allows users to generate extremely realistic and detailed terrains using a graph based interface where the user can add nodes that act as modifiers. The user can add as much nodes as he wants to achieve an endless combination of effects. It also implements very powerful fractal generators with the addition of Perlin Noise. Perlin noise is a procedural texture primitive that generates a gradient noise to increase the appearance of realism in computer graphics. Its powerful erosion system allows to naturally erode the surface of the terrain by realistically simulating natural effects that acted on that terrain for millions of years, such as water, wind and others in just a few seconds.

It also allows the user to insert shapes and roads on the terrain with realistic adaptation to the terrain's surface.

For the terrains to look as the user wants them to, it comes with a built-in texture creator that with just a few steps creates very complex textures according to height, erosion, vegetation and so on.

c. World Composer

Figure 17 - World Composer Example [98]

WorldComposer is a tool for Unity game engine that allows users to extract data from the real world. It includes a shadow removal tool, because there are always shadows on satellite images and if they are used in a game or simulation it forces a fixed sun position to match with the shadows. Also shadows in satellite images are almost completely black. WorldComposer solves this by removing shadows with its shadow removal algorithm, which will not only make the satellite images look way better, but also allows you to have fully day and night cycles.

WorldComposer is designed like Google maps with a scrolling and zooming functionality to select real world locations directly into Unity. The user can create multiple areas that display the real size in kilometres, and they can be exported with a few clicks of a button. Then for each exported area the user can create the terrains into Unity Scene. Satellite images can be exported to zoom level 19 which is 0.3 meter per pixel resolution. Elevation data can be exported up to zoom level 14, which is 10 meter per pixel resolution. WorldComposer only exports elevation height maps and satellite images, with itself the user can create awesome looking real World terrain. In combination with TerrainComposer it can mix or alter the height maps or add Perlin noise to enhance the detail, mix the satellite images with splat textures, add cloud shadows and so on. Also, the user can place trees, grass and objects and do the unlimited tweak and editing TerrainComposer offers to push the quality to unseen limits. [49]

2.7.3.2 **Flora**

Procedural flora usually includes all the vegetation such as grass fields, trees, bushes and flowers. When generating a procedural terrain the flora also needs to be generated, but the terrain mesh generator cannot do it itself, unless this generator only places pre modelled by hand vegetation, but this will not give the best result, because there will be no variance between vegetation elements.

Flora generation algorithms take care of covering a terrain, either procedurally generated or not, with vegetation. These algorithms generate the vegetation meshes according to input parameters to get different results like pines and palms or similar results like a groups of trees from the same species, but with slight differences like the height, the leave density and so on, depending on the algorithm's complexity.

The approaches to procedural vegetation generation can be divided according to the level of detail they produce in individual plant organs, plants, and complete plant ecosystems. One of the first approaches to plant simulation is the work of Honda [50], who attempted to create branching structures using a set of simple input parameters. Lindenmayer described a linear cellular subdivision mechanism as a rewriting system of terminal and non-terminal symbols of a grammar. This approach, later named in his honour as Lindenmayer's systems, or L-systems, is extended by Prusinkiewicz [51] by syntactic sequence and graphics representation that allows for generation of branching structures and 3D interpretation of the string of generated modules. L-systems are a very powerful system that allows for simulation of individual plants, trees, or entire ecosystems. The two most important extensions of L-systems are differential dL-systems that support continuous simulation of plant development, and Open L-systems that extended the plant simulation to allow for integration of exogenous flow and environmental sensitivity. L-systems can generate the entire scale of plant models from cellular subdivision, individual plant organs, to entire ecosystems.

Pure procedural modelling usually does not allow for the communication exchange between the plant and its environment (exogenous control). However, exogenous control, namely self-shadowing and collision detection, are among the most important factors defining the final shape of the plant. This has been addressed by the environmental sensitive automata of Arvo and Kirk [52], where plants are simulated as autonomous particles that compete for resources. The traces of each particle define the individual branches. Particles can branch and also produce leaves. A similar approach was proposed by Greene, where a voxel space is used as an additional data structure for quick illumination evaluation and collision detection [53]. This approach was later used for interactive plant modelling by Benes [54]. Recently, simulation for resources has been used for modelling leaf venation patterns, and later extended for simulation of plant competition to create individual plants and small plant colonies, where plants compete by growth for a predefined random set of local attractors in space. Most recently, plastic trees allow for an interactive modelling of arbitrary input trees that react to external conditions as if they were grown in the given environment.

Deussen described a first ecosystem simulation model to populate a height map with vegetation using competition for resources on the level of individual plants [55]. The virtual plants grow and seed, and over time their local area of influence increases. If two or more plants collide, their survival is evaluated and one plant is eliminated. This approach has been extended in different directions, such as adding virtual agents that affect the ecosystem, simulating vegetation in urban areas with artificial management of certain areas, or combining with real digital elevation maps[33].

2.7.3.3 Buildings

Procedural building generation is highly connected to city generation by the simple fact that cities have buildings. Although that might not be so simple, because the buildings need to adapt to a road network or the other way around even when they are processed together.

a. Geometric Primitives

This method uses the location of the building in a form of grid coordinates as a seed for the building generation. The building geometry is generated using the concept of combining geometric primitives to form building sections. Each building section is constructed using a different floor plan. The top most section of the buildings are created by extruding a three dimensional shape from the most basic of floor plans, composed from only a few primitive shapes. On each iteration another 3D primitive shape is added an extruded until the desired height is reached.[56]

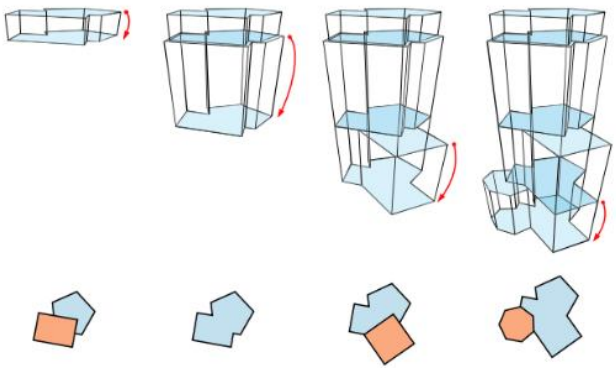

Figure 18 - Geometric Primitives - Floor Plan Generation [56]

b. L-Systems

This method assums that a building is generated by successive clips to its bounding box. The clipped section of its box are each iteration smaller, meaning this is an excelent algorithm to create skyscrapers. [57]

c. Building Generation Tools

There are a few tools capable of generating buildings and populations of buildings, but they usually have a complex system that generates the surrounding environment such as ground, streets and vegetation. The example below focuses on building generation only and does the job extremely well.

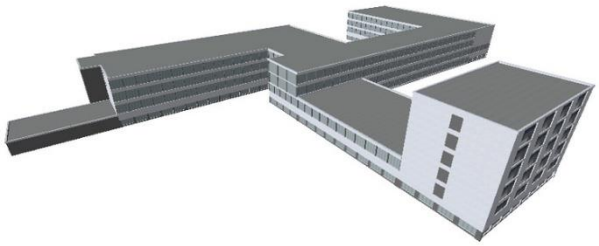

Figure 19 - BuildR building generation example [99]

BuildR is a fast, simple and effective system to create buildings within Unity allowing the user to generate a building as he wants. It provides one click procedural building generation, allows users to customise textures, dimensions and styles as well as generate building interiors, basements, cores and stairs. It can also export user creations to .obj and .fbx files.

2.7.3.4 Roads

Roads can be generated using a lot of different techniques and sometimes even a mixture of algorithms is used. The usage of different kinds of algorithms depends on the characteristics of the road network such as being completely random, following a predefined path or simply adapting to structures. The examples below describe the most used road generation techniques, because they are based on a simple idea and have great efficiency.

a. Grid Layout

When using a grid layout, roads are generated in a pattern of a uniform grid. This method creates a New York style road network. Since this method was very constant and repetitive, there was the need to enhance it. Greuter enhanced this method to create a less regular grid, with road angles different from 90º degrees as well as bigger blocks between each other. [56]

b. L-Systems

L-Systems also have their usage on natural phenomena modelling such as the generation of plants and other branching structures. A road can be considered a branching structure since there may be roads coming out of it. This method can incorporate various types of data such as elevation map, population density maps, water boundaries, vegetation maps and so on. Having such a big

input means that the roads generated will adapt to every condition at a given location. This allows the creation and interaction of the roads with the surrounding environment giving a very realistic result.

2.7.3.5 **Cities**

Procedural cities are actually a combination of multiple techniques to reach a single goal. This goal is a realistic 3D city environment. It is a combination of procedural building techniques, procedural roads, procedural terrain, vegetation, water bays and so on.

There is also the possibility to render a procedural city using bird eye view images, but this technique requires high definition images of the environment and the usage of computer vision algorithms to detect colours and edges in order to recognize buildings and roads as well as other objects. This only has an accurate result when the images have a very good definition, otherwise they won't be that good.[56]

a. Esri CityEngine

Figure 20 - Esri CityEngine - generated city example [100]

CityEngine [58] is a computer application developed by Esri and has the focus on generating extremely realistic cities to help certain industry sectors plan their work such has public transportation, GPS, urban planning and so on. CityEngine can generate anything related to a city, from rural areas to New York style cities. It has the possibility to generate a random city based on street layout type (grid style, web style, rural style and so on) and to load real world data from ArcGIS database to recreate a real world location. This database has extremely detailed and precise data including satellite images and elevation maps. Everything that CityEngine generates is human editable in an included 3D editor. The modification of the 3D model is surprisingly simple. Every road is made with nodes and edges, which further on are rendered with a 3D street model, and to edit them the user simply needs to drag and drop or add nodes to it. While the user moves the nodes the street model is updated, as well as the surrounding buildings that are

constantly modified to fit the area between roads. The user can configure other road parameters such as number of lanes, texture, angle, curvature, mesh detail and so on. As for the buildings the user can change them as well. Their facade can be modified within a huge amount of parameters, the number of floors, the building type and the general aspect can be completely rebuilt. Also, between each road and other roads Esri divides that space in terrain lots. Each lot can have a building or an open area such as a park or a grass field with trees. When a user modifies a road, all the surrounding terrain lots will be modified, therefore the constant change and update of the buildings.

This was just a brief summary of the CityEngine main functionalities, it is much more powerful then what is specified on this report, but for more information you can read their website [58].

Having these, although there are a lot more procedural city generators, none of them is so evolved as CityEngine, because it is extremely complex, well made, perfectionist, but at the same time feels so simple, making the best, largest and most detailed procedural cities. It is almost not possible to distinguish between a handmade city by an artist and a CityEngine city.

2.7.4 Extracting Rooftops

When attempting to procedurally model real world data from a certain location, sometimes the developer finds that there is not enough information. This information is most of the times related to existence of buildings on that location. Most of the wold data providers only concern on having better quality on their satellite images and having the roads the most updated as possible. Of course this has a reason to be so, which is GPS navigation. Having high quality imagery and updated road information, system users can identify much faster their actual position and GPS traced routes. Regarding buildings there is almost no information, at least open source and free of rights. OpenStreetMap has a bit of information about buildings, but since it is an open source project the buildings presence depend on the OSM developers' free will of identifying each building. Of course, there is information on the most important buildings such as Town Malls, monuments, stadiums and pavilions, but only a little about regular buildings. This brings us to the problem of attempting to gather buildings information using available data. The most obvious data source are the satellite images. There are a few studies and project that attempt to overcome this problem. An example of rooftop extraction is the work of Qi Min and Jia Jiankui that suggest an algorithms that begins with a calculated seed that is usually placed on the top of a rooftop. This seed will be grown using computer vision algorithms and aided by a Canny edge detector to limit the bounds between what is a building and what is not [59]. This approach retrieves the buildings contours, but there is still no information about buildings height or rooftop shape. There is an approach to extract rooftop shapes using buildings footprints [60]. This can be applied to satellite images only if the computer vision algorithms can detect all roof top's interior vertices and edges. Now for the building's height, there is a solution, but of course it is very

relative to the imagery quality and contrast. This solution was developed by Mohammad Izadi and Parvaneh Saeedi that attempt to detect building's height using satellite images by detecting the building's shadow and creating an estimate of a projected shadow for each given height. It uses a genetic algorithm that generates different heights for the buildings. Then the projected shadow is calculated and compared to the actual shadow and when they match we have the height [61]. They claim that have achieved a solution that has a mean error of 27cm for satellite images and just 15cm for aerial images. This is because the image resolution and detail as said before can be a problem.

2.8 Tools to be Used

To implement this framework I was free to choose all of my tools, but with the condition that it had to have both a game engine and a microscopic traffic simulator.

2.8.1 Unity 5

Unity 5 is the game engine I choose, because I am familiar with it and have used it to do other projects. It is also very intuitive and simplistic allowing the user to do very complex tasks with just a little effort. There is also a huge amount of very well made documentation, tutorials and even specialized forums. The Unity ecosystem is available to anyone who downloads the Unity engine. The Unity engine integrates into one unparalleled platform the tools to create 2D and 3D interactive content, collaboration solutions, rapid multiplatform deployment, and retention, advertising and analytics services to grow your business.

2.8.2 UnitySlippyMap

UnitySlippyMap [62] is an open source project developed by Jonathan Derrough that aims at helping developers create maps working with a variety of online tile providers including OpenStreetMap. It can create tiles on Unity's 3D space along the XZ plane. The map can be zoomed and dragged using the mouse and new tiles appear or disappear as needed. In order to include OpenStreetMap maps in my application, UnitySlippyMap needs a few changes in the code. One of them was disabling all unnecessary features to make the most simple and easy to use interface. The second was to allow to place a marker on the map using the mouse and when the marker is placed the second maker will define a diagonal that will be used to create a bounding box. This bounding box limits the area to be downloaded.

2.8.3 Open Street Map

OpenStreetMap [63] is an editable map that is a trusty representation of the whole world built and enhanced by the community [64]. Being so, it is almost an open source project with the main goal of sharing with everyone real world map data so people can use it for whatever they want. Being an open source tool some researchers start thinking about solutions to solve real world problems such as Moritz Göbelbecker and Christian Dornhege that attempted to parse the OSM data in order to use it on a robocup simulator for rescue missions [65].

2.8.4 Google Static Maps API

Google Static Maps API allows users to download satellite images based on the parameters sent on a URL by HTTP. These parameters can be the geographic coordinates, the zoom, the address of the location and pretty much anything that Google Maps uses to refer to its locations. Having this, it is possible to download satellite images from Google Maps.

2.8.5 SUMO Simulation of Urban MObility

SUMO [66] is a microscopic traffic simulator, which consists on a platform developed by the German Aerospace Centre (DLR) that has all the needed tools to implement a traffic simulation. It can import any kind of road network, even from OSM (OpenStreetMap) and generate a road network according to that data. Having the network it can generate traffic, traffic routes, traffic lights, pedestrians and different types of vehicles with different characteristics. A great advantage of this software is that it is extremely portable and can be integrated in any external application, because due to TraCI, which is a middleware layer that creates a TCP channel and has a very well defined protocol, SUMO can act like a server and be controlled by the external application [67].

2.8.6 CGIAR-CSI

CGIAR-CSI [47] is a free SRTM digital elevation data provider where all that data has been processed to fill data voids, and to facilitate its ease of use. Their mission is to provide DEM data in an effort to promote the use of geospatial science and applications for sustainable development and resource conservation in the developing world.

The free data provided is from SRTM 90m DEM that has a resolution of 90m at the equator, and are downloadable in seamless mosaicked 5 deg x 5 deg tiles. All this data is available in both ArcInfo ASCII and GeoTIFF formats to facilitate their ease of use in a variety of image processing and GIS applications and cover over 80% of the entire globe.

Also, all the data has been processed to remove possible “holes” on it caused by bad height reading on certain locations due to water basins or heavy shadows.

2.8.7 7zip

7zip [68] is an open source compression and decompression tool capable of high compression and ease of use even by command line. Being open source it can be adapted to integrate multiple compression and decompression algorithms as well as cryptographic algorithms to encipher files in a secure way. The need to use 7zip is because the data downloaded from CGIAR-CSI comes compressed in a zip file and in order to be decompressed automatically by the application, an open source decompressor was needed to be integrated in the application. Since 7zip can run in command line it is easy to execute it and extract the needed files into a specified folder by command line arguments.

2.8.8 Blender

Blender is an open source software aiming to help artists create 3D models by means of free to use tools and software, as well as an active community and development team.

On the context of this project, Blender is used to take care of any manual modifications to meshes, such as enhancing the vehicles mesh, create simple collision meshes for the objects, more easily mapping the UV coordinates and 3D model simplification [69].

2.9 Related Work

There are a few attempts to do a 3D reconstruction of real-world data using OpenStreetMap, but none of them do it so deeply and are far away from having any kind of population of vehicles or walkers. M. Goetz and A. Zipf transform the OpenStreetMap buildings into polygons with no windows and no texture, and there are no roads and no vegetation [70]. Two more examples are [71] and the work by Neubauer, et al. [72]. Comparing our approach to these examples, this project advances a step further as it merges procedural modelling of real-world data and driver behaviour analysis with the possibility to generate a scene that can be exported as a 3D object file, which can be opened by 3D modelling applications.

Of course there are other alternatives mentioned above, such as World Composer that can be integrated with multiple tools to generate the buildings and the roads as well as the vegetation in an extremely optimized way. But on the other hand, each one of these applications is paid and optimized to do a very strict and specific task, making it very hard to expand in terms of functionality. Another problem is that when using a paid application to develop another, the final version of the code can never be open source, because of legal rights, it is obvious, because the

paid code will be available to anyone for free. Having so this tool will only use free and open source software, in order to be freely distributed and modified by everyone. This is the major difference relative to any other existent application/project. Although not so immersive as real scale driving simulators, serious driving simulators can give important contributions to the analysis and proper understanding of driving behaviour, such as what has been carried out in previous projects [73][74][75][76][77].

2.10 Summary

So far, this dissertation presented a detailed review on some of the existing technologies and techniques. First of all there were some descriptions on simulation engines followed by driving simulators, their types and top simulators, as well as detailed functionality of their possible peripherals. There is then a change of topic to cover procedural modelling latest and best applications followed by the technologies behind the content generation such as terrains, flora, roads, buildings and cities with a simple explanation and a few software capable of generating them. In the end the reader finds the technologies that are going to be used and an explanation on why they are going to be used.

There are a lot of techniques and technologies to choose from, every of them with their own positive and negative sides. For this project most of them will be needed, such as terrain generation, height map download, procedural buildings, roads, vegetation, textures, traffic simulation forcing procedural roads to have lanes and traffic rules and microscopic traffic simulator integration as well as a fully controllable and interactive vehicle.

Chapter 3

Methodological Approach and System Architecture

This chapter starts by giving a detailed presentation of the problem to solve. It presents the system's architecture with explanations on why and how it is going to work, on a high level without too much detail regarding algorithms or techniques. This project follows up and extends previous attempts to implement an integrated platform for traffic and transportation engineering and research, focusing on driver behaviour modelling and the interactions of drivers with the surrounding environment [78][79]. It will underlie the integration of concepts such as artificial transportation systems [80] and serious games applied to transportation [81].

3.1 Problem to Solve

Nowadays the configuration of driving simulation environments takes a very long time, because the modelling of 3D scene by hand is a very laborious and time consuming task. If one wants to conduct an experiment on a real world location, the artist that is going to model the scene needs to gather information relative to the elevation, the roads, the buildings and so on, making this a very expensive task. Therefore the driving simulation experiments should be faster, easier and cheaper to conduct, as well as real world generation. There are multiple free world data provider, so it should be possible to procedurally generate real world locations automatically to tackle the production time and cost problems. Serious games provide a cheap, portable and intuitive way to create any kind of experiment, so this is the right way to go.

Therefore, this dissertation has the objective of exploring different techniques and approaches to tackle this problem by developing a framework based on serious games that is capable of:

- Procedurally generate real world locations in 3 dimensions automatically;
- Simulate a virtual vehicle that can be controlled by the user;
- Simulate and give the user the freedom to configure different simulation conditions that directly or indirectly affect the results of the simulation.

To accomplish this goals, the framework should:

- Gather real world data and render it in 3 dimensions;
- Allow the user to configure different simulation conditions;
- Allow the user to drive a virtual vehicle;
- Be completely open source;
- Export the generated scene;
- Simulate pedestrians;
- Integrate Oculus Rift;
- Simulate traffic;

3.2 System Architecture

The image below is a diagram that shows all the system architecture including all the interactions between the system's modules.

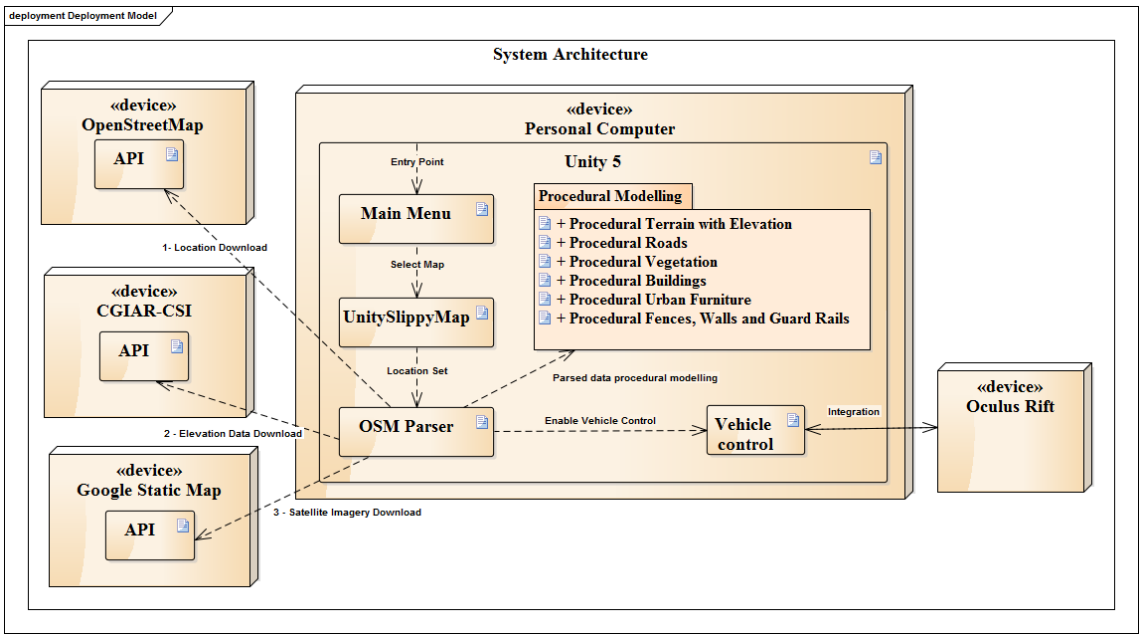

Figure 21 - System Architecture and Module Interaction

In order to accomplish all goals, the system needs to integrate multiple existing tools.

For starters, Unity 5 will be the main application. The user runs the Unity executable and is prompted with the main menu of the application. Here, the user can select options and the location he wants to download, either by geographical coordinates or by address. This information is kept in a class with static variables called "Manager". The "Manager" is responsible for storing key values such as all the options the user selects and the current vehicle. On the menu the user can travel to the vehicle selection scene. There is also an input box for the user to type the desired location. When this box is filled with a valid location, its value is sent to OSM API by http protocol and UnitySlippyMap opens up a map on the specified location. The user can zoom in and out and drag this map. On it, the user can select the bounding box he wants to download. Once it is selected the corner limits of the bounding box are sent to the OSM API and it responds by sending an XML file to the requester with the desired location. This XML file contains all the information about the location the user downloaded. When it arrives to Unity it is then processed by a parser on a new scene where all the 3D construction will happen. Initially, the parser will read the bounding box's limits of the downloaded area as geographic coordinates that will be used to find and download the corresponding tiles of DEM from CGIAR-CSI. When the DEM finishes loading and the terrain is configured with a height and size to fit the mentioned bounding box, the parser will download all the satellite images for that location by making requests to Google Static

Maps API. As the images arrive to Unity they start being placed on the respective terrain tiles as a texture.

While the XML is being parsed, 3D models start being generated by making use of the parsed information and the classes on the Procedural Modelling module.

When the meshes are all generated the user is allowed to control the selected vehicle. If Oculus Rift integration is enabled, then the vehicle will only have the cockpit camera active, not allowing to switch to any other camera.

3.3 Unity3D + SUMO (Simulation of Urban MObility)

First of all, the basis of this integration takes in account the great capability of SUMO that is acting like a server when launched by command line using a specific argument that allows to create and open a TCP port, for an external application to connect. While acting as a server, SUMO responds to commands from the connected application. On the other hand, these commands are very complex and hard to compose, because they have to be managed at byte level following a well-defined protocol. The same happens to the answers received from SUMO. Also, the SUMO's documentation explains all the possible command-answer combinations, but there is no information at all on how to compose a specific command making it very hard to test, since SUMO may not answer or say what is happening with the commands received. There is a platform, created at FEUP's LIACC, written in Java, called TrasMAPI [82] that has a good list of predefined commands where the developer just has to insert the arguments, but it still does not cover this project's requirements since the commands needed to extract all vehicles do not exist yet, which brings us to stack zero.

Another problem of this integration is the fact of being a TCP connection. Unfortunately SUMO can't be part of Unity, it can only communicate by TCP connections. A goal of this project is to allow the user to drive a virtual vehicle, which needs to be in real time with the highest frame rate possible. In order to have a smooth driving experience it needs to feel seamless to the user and this only happens over 25 FPS. The problem is, that to recreate the vehicles of the simulation on Unity the information of their positions needs to be sent to it constantly, otherwise the vehicles movement won't feel seamless, but the information takes time to be written to and read from the TCP channel. This is inevitable. A solution to this problem is, for instance, instead of asking for and receiving the information every update, do it as soon the last information arrives to Unity, asynchronously. Of course the movement will not be seamless like this, but it can be corrected by position interpolation over time. SUMO can send a specific vehicle's speed at the moment and using it, it is possible to interpolate between every frame. Unfortunately this will not work perfectly, because the interactions between the drivable vehicle and the simulation vehicles will not happen in real time, but only when Unity sends the drivable vehicle positions, making the vehicles run over each other, because of the interpolation between SUMO answers. In addition,

this approach adds even more overhead to the TCP connection, because the speed of each vehicle needs to be sent in a different command from the positions commands. So to render the vehicles correctly there are needed 2 commands and 2 answers. The overhead problem gets even worse when lots of vehicles are running, making it impractical.

There can be found on the internet a few tools and tutorials that can help building a traffic simulation inside Unity, which is a very plausible idea to overcome the TCP overhead problem. In addition it makes possible to simulate full physics and collision interactions between all vehicles and also each vehicle can have different characteristics determined by their shape and mechanics and can adapt to the road inclination instead of a 2D model of the network that always considers the roads flat surfaces.

Another problem with SUMO is that it does not provide a seamless simulation. The simulation time step is too big by default and in order to enhance it, some changes need to be done to the source code.

Finally, SUMO's lane changing model is not continuous, it is discrete. This means that the transition of one lane to another is not seamless. On other words, the vehicle simply disappears from its current lane and appears on the lane it wants to go to. The vehicles do not overlap, because they know about each other's positions, but when a human is driving a virtual vehicle on this simulation it will not have good results, because vehicles simply appear in front of him whenever they want, and there is no signal of the vehicles intentions except for signal lights.

Having all these problems definitely makes the SUMO connection a bad way to go, because most of the time would be spent attempting to establish a good connection speed. So the supervisor told me to forget about SUMO and to focus harder on the procedural modelling algorithms and all the other functionalities. Therefore this project is supposed to do everything even better than what it was expected on the beginning, with the exception of the SUMO integration that fortunately was abandoned.

3.4 Development

This section intends to clarify the reader on development matter, because this is a very ambitious project with lots of features to implement. The architecture described above might seem simple, but the underlying implementation is very complex, because it requires the study and knowledge of the tools to be used and every component that interacts with them needs to be carefully designed and tested to grant no space for errors, malfunctions and bugs.

The development was based on the implementation and merging of different functional prototypes, each with a different specific functionality. What this means is that the development was done by successive iterations, each one consisting on a functional prototype construction and respective testing. The validation of each one was done simultaneously by half of the time together with the supervisor. A functional prototype is an application that can perform a very strict

and specific task and are usually a separate scene on the Unity project, isolated from the other scenes. By creating a different scene for each prototype it is possible to easily merge them with the main scene, because they belong to the same project and automatically share the same logic. Some prototypes where implemented along the 1st semester, explaining the main reason for this project to be so ambitious.

The first prototype developed is able to load and parse a file from OSM. While loading the file, Unity draws lines that correspond to the roads and buildings. Meanwhile it starts calling the Google API to download satellite images and display them on tiles to cover the entire region, with the best resolution as possible.

The second one allows the user to control a vehicle. This one is meant to fine tune the handling and physics of the vehicle to control.

Now, it was time to build the main scene. Therefore, the first and the second ones were merged into the main scene.

The third prototype generates procedural roads along a predefined path.

Next, another prototype was developed to create the day and night cycle. This prototype is very simple, it is just a script, so it was merged with the road generation prototype to make it easier to build the illumination system. The street illumination is generated on this prototype.

The fourth prototype can load an OSM file and download a DEM, corresponding to that location, from CGIAR-CSI and unzip it using 7zip by command line. After the download and the unzipping are complete the corresponding region of that OSM is extracted from the DEM file and applied to a terrain to generate heights from the real world.

Having these prototypes it was time to merge them all into the main scene. By this time the work was stopped for a time period of around a month due to personal setbacks. When the work was resumed a new version of Unity had been released. The work was started on Unity 4.6, but then Unity 5 came out, introducing a lot of new features as well as API modifications. After the update a lot of scripts were needing an API update, both automatic and manual. Unity 5 includes the new Physics 3.3, which provides major modification to the vehicle's physics. After the update the current vehicle physical configurations and scripts didn't work at all. Not only the work was delayed a month, but also the vehicle physics had to be remade from scratch.

The first step after the update of Unity and the project was to quickly fix the vehicle physics in order to at least have a more or less drivable vehicle.

Next the buildings prototype was created. This prototype includes everything about the building generation and consists on a script that generates a building based on a path.

Then it was time to generate the vegetation. This took a bit more than the buildings, because there was a need to find the balance between the number of vegetation elements and the performance of rendering such objects.

After integrating the vegetation on the main scene, a new prototype was started, this time to generate the weather conditions. The weather conditions depend on the duration of the day, therefore this script depends on the script that controls the day and night cycle.

The project started getting more composed and complex, so it was starting to lack a main menu with configurable options. So a new scene was created to build the menu and the options. As the project evolved more options were added to the options menu.

By having the menu, the next step was to implement the connection channel between it and the main scene. This channel is UnitySlippyMap that after some modifications was ready to be integrated in between the menu and the main scene. UnitySlippyMap is the responsible for displaying the world map of a desired location and allowing the user to define the bounding box for the specific area to download from OSM.

All the prototypes were developed along with the supervisor and fully tested before and after merging them into more complex scenes, to guarantee quality, efficiency and functionality. The entire ecosystem of merged prototypes was also fully tested by multiple people, after each prototype merging, to ensure diversity in the way people use the application, to find different glitches, to determine improvements and guarantee that the application is intuitive to use.

3.5 Implementation

This section describes the lower level details relative to the details mentioned on the above section. It starts by describing each scene that constitutes the project associated with the respective flow of the architecture diagram.

3.5.1 Main Menu

The main menu provides the initial interaction between the user and the application and is extremely important, because it allows the user to choose options and modify parameters that will affect the application itself.

This menu has an appellative look and it is very simple to use. The main screen, shows the user very fast options and buttons to start and quit the application, to go to the advanced options, to go to the vehicle selection screen and to activate the Oculus Rift integration. There is an input box in the middle left panel of the menu where the user can enter the location by address or by coordinates in degrees. The application only starts if the input box has a valid location.

3.5.2 Main Menu Possible Configurations

On the main menu it is possible to go to the Advanced Options. This is a more complex menu that allows to choose render option, which limit or allow what objects are going to be rendered in 3D from the real world data. It also allows to configure the duration of the application's day in minutes and the starting hour. Another feature on the advanced options is the adjustment of the pedestrian density, which is presented to the user as a percentage relative to the

size of the location, to prevent an extreme number of pedestrians in the scene, avoiding frame rate decrement. It is also lets the user select what weather conditions he wants present on the scene and the weather changing probability. Finally, it gives the possibility to configure the vehicle's damage system. The user can choose between no damage at all, visual damage only or visual and mechanical damage. On the right corner there is information regarding the controls of the vehicle that can be configured before the application starts.

Both the Main Menu and the Advanced options are built on UI elements called Canvas. These Canvas react to the screen's resolution. An advantage of the Unity's GUI builder is that it is possible to configure the layout of all the elements inside a Canvas, such as the anchor point, which is responsible for the relative positioning of the element and the scaling properties. Therefore the user interface was designed to adapt to the resolution of any screen in use.

The Main Menu starts an instance of a class called "Manager", which is responsible for managing useful data between different scenes. Each button or toggle of the menu has an event listener attached that calls a specific function, declared on the Manager, to perform an action. These actions are mainly to update the values of static members of the Manager class that will be used on the other scenes. When Unity changes scenes, all the elements and instances of scripts belonging to that scene are destroyed and replaced by the elements of the new scene. The same happens to the Manager script, it is destroyed, but since the relevant values of the options menu are kept stored in static variables, they remain upon the instance destruction, because static values do not need an instance, they are stored in memory while the application is running.

3.5.3 UnitySlippyMap

UnitySlippyMap creates a tiled map on Unity's 3D space along the XZ plane that can be zoomed and dragged, making new tiles appear or disappear when needed.

Since this project needs to download data from the real world, mainly relative to the streets, OpenStreetMap is the best bet, because the data is open source, free and is constantly being updated by the community. To include OpenStreetMap maps in this project, UnitySlippyMap needed a few changes in the code. One of them was disabling all unnecessary functionalities to make the interface simpler. The second was, allowing to place a marker on the map using the mouse. The user can hold the left mouse button and drag to move the map, and to place a marker simply press the right button. The marker is placed on the target that is always on the centre of the screen and when the marker is placed, the second maker will define a diagonal that will be used to create a bounding box. This bounding box, limits the area to be downloaded. When the user is satisfied with the marked area, one can simply press the "Location Selected" button and a request to the OpenStreetMap API is done by URL. This request contains the coordinates of the corners of the bounding box, and will download the specified location as a .osm file that goes into "Assets/OSM". Otherwise, if the bounding box is not the desired, the user can drag the map and place a new marker, making the current bounding box disappear.

The location displayed on the map is the location the user entered on the main menu. In case the location is wrong, the user can zoom in or out and search for the desired location or simply press the "Main Menu" button to get back to the main menu and enter a new address.

When the Start button is pressed on the Main Menu, a short request to OSM is made to check whether the location exists, if it does not, the input box starts flashing red, otherwise the address's coordinates are passed to UnitySlippyMap to display the location's map when its scene is loaded.

3.5.4 Elevation Data

Elevation data from the real world is usually gathered by expensive methods such as LIDAR, SRTM and so on, meaning that it is hard to find free to use data. Another problem is the data resolution and coverage of the world surface. Therefore, there was a lot of research on this topic to find the best data resolution possible that covers the largest area of the Earth's surface.

The best provider found was CGIAR-CSI that provides free data gathered by STRM technology and covers around 80% of the Earth's surface, leaving apart the North and South Poles, and has a resolution of 90 meters, which means that it divides the Earth in 90 meter tiles, where each corner of a tile contains an elevation value. Also, this provider divides the world in a grid layout, where each cell covers 5 degrees of the Earth's surface both across latitude and longitude. The data available to download is a 5 degree cell of this grid and can be a GeoTIFF file that is usually used by specialized applications that read this image file format and the other is an ASCII file.

The ASCII file is a text file that has all height values for that area in a grid layout. It contains information about the number of rows and columns of this grid, as well as the lower left corner longitude and latitude. Another important parameter provided is the size of the cells of the grid. As explained before, each file downloaded corresponds to a well-defined cell that covers 5 degrees of the Earth. This file itself, divides that specific area of the Earth in 90 meter tiles. The elevation values are expressed in meters and are always greater or equal to zero.

CGIAR-CSI divides the Earth in 5 degree cells, from -180º to 180º longitude and -60º to 60º latitude, which give a total of 360º/5º = 72 cells horizontally and 120º/5º = 24 cells vertically. To download data for a location the algorithms needs to know the cell's index that corresponds to that location. To do so, this project uses a function that starts at the minimum latitude and longitude, which is the lower left corner of the planisphere and iterates across the Earth by 5 at 5 degrees, incrementing an index counter for the latitude and longitude until it finds the desired location's central coordinates. When the index is found, the application builds an URL that is used to request the CGIAR-CSI's website to download the desired data. Their servers are usually very slow and since the data is usually around 20MB it can take around 5 or more minutes. While the application downloads the file the user can follow the download progress. The downloaded file goes into "Assets/TRDZ" folder and comes compressed in a ZIP file. It needs to be extracted in order to access the ASCII file. To do so, the application uses 7Zip, because it is a very good

program and open source. To extract the ZIP archive the application creates a process that will launch 7Zip on command line. Since C# is the programing language of this application, there is a method that can launch a new process with command line arguments, in order to extract the file to “Assets/TD”. This allows to extract the ZIP file using just a command. To use 7Zip, it needs to be on the project Assets folder, so this folder should not be moved or altered anyhow.

Before downloading the data, the application checks first if the desired file is already stored on the “Assets/TD” folder, if it is, the algorithm moves to the next step, otherwise it starts downloading. This avoids wasting time downloading the files if they were downloaded before.

3.5.5 Location Data Parsing

A location file has the .osm extension, but it is nothing more than a regular XML. Being so there are tools capable of parsing such files almost automatically. C# contains a library belonging to System.Xml that allows to create an instance of an XmlTextReader, which opens a file and starts reading it node by node. It contains functions that allow to read node attributes, node startings, node endings, and anything else related to the XML structure. Making the parsing of the file much easier and intuitive.

The file contains multiple different nodes, but only a few are relevant for this project. Its nodes are “bounds”, “node”, “way”, “nd”, “tag”, “relation”, “member” and the “osm” itself.

The node “bounds” refers to the bounding box of the downloaded area, its attributes are the minimum and maximum latitude and longitude.

The node “node” refers to a node of the map. Its relevant attributes are its location in latitude and longitude as well as its ID. It can have multiple sub nodes called “tag”.

A “way” node represents a group of “node”. It contains sub nodes “nd” that are a reference to “node” nodes. A “way” can also have multiple “tag”.

A “tag” identifies what the “node” or the “way” is. For instance, it is by the “tag” that one can know that a “way” is either a building or a street.

In order to organize and store the information there were created three classes called “Node”, “Way” and “Tag”.

The parsing of the location data is done on the same scene as the elevation data download and retrieval. This scene contains a few game objects essential to the algorithms used. Most of them are used as managers that store data and references to materials. The main game object holds the elevation data processing and OSM parsing scripts. There is another game object called “Scene Manager” that has multiple scripts attached to keep references to all the materials that can be used by the parser, to keep references to some assets that are used to generate vegetation, such as trees 3D models and bushes 3D models, stores references to street lights 3D models, generates pedestrians, controls the level of detail (LOD) for the pedestrians, and has an interaction controller that brings up the pause menu and respective interactions.

The first thing the parser does is to create game objects that will separate the different assets. There are game objects that will contain and separate all bushes, trees, amenities, buildings, walls, roads and terrain tiles from each other. It also creates a game object that will contain all the game objects above called “SceneG”. Then the parent of all the game objects above is set to the “SceneG” game object. This creates a hierarchy, making it much simple to export the entire scene further on, because all the information is organized.

The next step is to call the routine that will parse and separate the information. This routine starts by processing the node “bounds” by retrieving the bounding box values. This was already done when the user selected the location, but the requested location’s bounding box and the downloaded location bounding box might be a bit different, because OpenStreetMap approximates that bounding box to one that fits best according to its database. Having the correct bounding box it is time to generate elevation tiles.

After the tiles have been generated the parser creates two lists. The first one will contain the Nodes and the second the Ways. Then it starts reading the file node by node. When the parser reads a node “node” it creates an Object from class Node that stores the node id, longitude and latitude and initializes a list of tags for this node. This object is then added to the Node list. The tags are mandatorily inside “node” nodes or “way” nodes, so the algorithm mandatorily reads a node “node” or a node “way” before reading any tag. Having this the algorithm knows if it is reading the children of a node “node” or of a node “way”. So, when the algorithm finds a node “tag” it reads that tag’s values and stores them on a Tag object. This object is then added to the list of the last Node or Way added to the lists depending whether the algorithm is reading the children of a node “node” or of a node “way”. The algorithm moves on reaching the first node “way”. A node “way” has only one value, which is its id and has multiple children nodes called “nd”. A node “nd” is a reference to a Node, so the only value an “nd” has is the id of the referenced Node. Just like the tags, nd’s are read and the algorithm searches for the corresponding Node on the Nodes list and stores this Node on the list of Nodes of the Way being processed. When the tags are reached, the process was already explained above. When the way finishes being parsed it is added to the game scene as a series of game objects that have the same parent whose name is the id of the way. These game object are empty, they only mark a position. Further on we’ll see why this game object were created. The Way keeps a reference to the parent game object.

When the algorithm reaches a node “Relation” the parsing stops, because all the information below is not relevant.

Next the algorithm iterates through each Way and Node stored on the lists and attempts to classify them by analysing their tags. The tags should be mutually exclusive, what this means is that, if the first tag is “road” the next cannot be “building”, but can be “residential”, so the algorithm tries to filter the tags, exclusively. Above, it was described that there are game objects called bushes, trees, amenities, buildings, walls and roads to separate the different assets. What the algorithm does, is creating lists of Node and Way that are associated to only one of the above

game objects, meaning we have lists for bushes, trees, amenities, buildings, walls and roads. Once all the tags of a Node or Way are processed, they are added to the respective list.

Now, that all the information is organized it is time to procedurally generate meshes. The generation follows a logical sequence to avoid assets overlapping each other. So, the first asset to be rendered are the roads, then the buildings, followed by the vegetation, amenities and finally walls. The algorithm starts by checking, if the option to render this specific asset is toggled on in the Manager script and then, the algorithm calls the respective procedure to generate this type of asset. To generate an asset, the algorithm iterates along the list of this type of asset and for each Way or Node on the list, the algorithm adds the respective generation script to the game object this Way refers to and iterates along all the tags of this Way once more, to configure the script values. This process is repeated on purpose, to simplify the code, because there are hundreds of "if clauses" on top of each other and this repetition allows to separate a bit more the "if clauses", helping the developer to focus only on the characteristics of the asset that function will process. When all the tags are once more processed and the script values configured, the procedural generation function of the added script is called to generate the mesh. The process repeats for each type of asset. All the generated meshes have a common parent, which is "SceneG" as stated above.

When the generation is complete, the algorithm creates an instance of the drivable vehicle, selected by the user on the vehicle selection scene, and adds a script to the main camera to follow this vehicle. Finally, a script is added to "SceneG" that will allow the user to export it as a .obj file.

3.5.6 Satellite Image Gathering

Having roads and buildings with no surrounding environment is not immersive enough. To enhance the user experience, satellite images of the respective location are gathered from Google Static Maps. The only needed information, to download the images, is gathered from the .osm file. This information are the bounding box limits.

Google Static Maps provides various images for the same location. For each level of zoom there is a different image and so, a different scale. The zoom level used was 18, which is the maximum for the free version of the API and is capable of covering an around 382x382 meter tile. This zoom was chosen, because it is the closest to the earth's surface. This means more detail, but a smaller field of view. Having this zoom level means that the images are only a very small part of the location and, to download a consistent detailed photograph of the entire location, the images have to be downloaded tile by tile.

A tile is a square plane, which has a satellite image as a texture. The initial square plane is just 10 by 10, but since the map is built in real scale these planes have to be scaled by a factor equal to the length of the satellite image, in meters, over the default size of a tile, meaning the planes will be scaled by 382/10 = 38.2.

If there needs to be more than one satellite image to cover the entire area, then more than one tile needs to be generated.

Knowing the bounding box limits and that the satellite images cover a constant distance of 382 meters, the algorithm iterates by 382 meters at 382 meters, starting from the minimum latitude and longitude, until it reaches the maximum latitude and longitude and for each 382 meters, a tile is created and positioned on the respective distance from the centre of the locations bounding box. Then, a request is made to Google Static Maps to download the image. Now, if the tiles were placed on their exact position, corresponding to the real Earth's position, Unity would throw an overflow exception, because the position values are all floating point values and the desired position's value may not fit on a float, so the centre of the location is translated to the origin of the Unity's referential and the tiles are placed on their correct real world scale position, but relative to the translated centre, preventing the overflow exception, since the position values are never that large this way.

As soon as the download of the image finishes, a material is created using this image as a texture and it is applied to the tile. This tiles are simple flat planes and do not have any information regarding heights. Each elevation tile, is generated right after the respective flat tile, but in a much complex way. The plane is only used to hold the texture when it arrives and to simplify the positioning of the elements that constitute the elevation tile.

3.5.7 Terrain Elevation

To generate terrain elevation, in this case, elevation tiles, the algorithm uses the data gathered from CGIAR-CSI and creates tiles that have the correct elevation, for that specified location.

Each of this tiles, is generated using the same conditions as the ones in the section above, such as the scale, position and translation and are generated right after the respective plane.

For starters, each tile is composed by 16 square polygons, which is the result of the following calculation:

- A tile has to cover around 382 meters on x and y axis to be in real world scale, meaning it will cover the satellite image completely and we also know that each elevation point is separated by 90 meters from the others. If each polygon that makes up the tile covers 90 meters on both axis, then we will have 382/90 = 4.244 polygons. The result is a decimal number, but it does not make sense to have 0.244 of a polygon, therefore to cover the area, the value e rounded making 4 polygons. To cover the elevation along x and y axis we need 4x4 = 16 polygons in total.

Although, this approach introduces an error of around 5.5 meters per polygon, meaning that the vertices that contain the height values will be 5.5 meters offset from their real world position, but since the data resolution is 90 meters the error is almost insignificant.

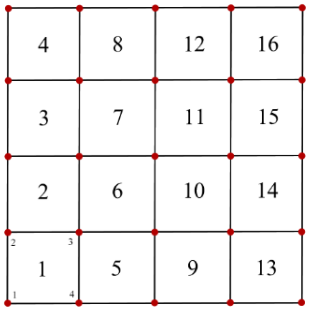

Figure 22 - Elevation tile vertices and polygon processing order

The image above show the order each polygon is processed and the order the vertices are used, following the left hand rule, so the front face is facing upwards. Each red dot is a vertex that contains a height value. Every time two black lines cross, means that vertex is shared by other polygons. To apply the whole texture on the entire tile, the UV coordinates need to divide the texture in 16 smaller textures, each one covering the respective polygon. Since the range of the UVs is from zero to one, where (0,0) represents the bottom left corner of the texture and the (1,1) the top right corner, then the UV's for each polygon are set based on its relative position to the others. For instance, the UV coords of the first polygon will be (0, 0), (0, 1/4), (1/4, 1/4), (1/4, 0), following the order of the vertices and for the second (0, 1/4), (0, 2/4), (1/4, 2/4), (1/4, 1/4).

Figure 23 - Elevation Tile UV Mapping

The image above, attempts to illustrate the basic idea behind the UV mapping. The algorithm takes the image on the right and slices it in 16 pieces that will fit the respective polygon. If the UV coordinates were not set, the image on the right would repeat on each polygon, so the UVs of each one are mapped to a percentage of the texture.

When the tile is generated, it is composed by 16 separate polygons. This is not a bad thing for one tile, but when the scene has a big number of tiles the processing power to draw those increases rapidly. Unity renderer can process a mesh with thousands of vertices much faster than processing thousands of meshes with only 4 vertices. Having this principle, the tile's polygons

are combined by a combining function, found on Unity Community Wiki and this clearly improves performance by reducing the number of game objects on the scene, thus the draw calls.

This process repeats for each tile the algorithms needs to generate, so it can cover the desired location. To do so, the algorithm iterates 382 meters at 382 meters, until the location is all covered, just like the flat planes explained in the section above.

3.5.8 Procedural Roads

Having all the "ways" read from the .osm and organized, it is time to render all the roads. For starters, let us remember what was explained in section 3.5.5. All the Ways are separated by their tags and classified, so they can be stored in the correct lists. Also, remember that each Way has a reference to the game object it represents. For each Way on the list, the algorithm calls a function that adds the script responsible for generating the road's mesh and the tags are processed once more, to configure the values of the added script. This values are information that allows the generation procedure to know about the existence of sidewalks on the left or on the right, the road and the sidewalks width, the surface type, the road type and whether it is lit or not.

OpenStreetMap calls "highway" to any kind of road. So, the algorithm that searches for roads, looks for the tag "highway", in order to add the way to the roads list. Also, OSM considers the existence of around 27 different types of roads, but we only consider 14 for simplification and some of them have predefined values, because they vary from country to country. Some types of road require a very specific mesh, such as motorway nodes, bridges, tunnels or have less relevant and frequent usage, such as bridleway (specific for horses). So, due to the short delivery deadline, those kinds of road types were not taken in account. The considered ones are:

- Primary roads – usually 6 meters width, no sidewalks, two lanes, not lit
- Motorways – same as Primary roads
- Secondary and Tertiary – same as primary roads, but around 4 to 5 meters width
- Residential and Living Streets – inside city roads, so they are always lit, they have sidewalks on both sides, unless a tag defines it as sidewalks on the left or on the right or none and they have two lanes.
- Trunks – like motorways, but do not meet the same requirements, such as the speed limit or safety measures.
- Tracks – usually a dirt road used for walking or off-roading.
- Not clearly defined roads are considered non-laned asphalt roads

- Paths – similar to tracks, but for undefined roads, dirt surface usually
- Raceways – usually over 6 meters width, not lit, no lanes, and race track surface.
- Footways – not over 2 meters width, not lit, no lanes, tartan surface (red to better distinguish), may have sidewalks of not
- Cycleways – same as footways
- Road Construction – usually around 5 meters width, may be lit or not, may have sidewalks or not, usually have a concrete or dirt surface

When all the parameters are configured and assigned, it is time to generate the road mesh.

There are a few techniques to generate the road mesh, such as Bezier curves, round splines or grid layouts. These are all good techniques for completely procedural scenes, but this project is not the case, because we already have the paths a road has to follow, on other words its nodes. Having these nodes restricts the road's shape and roundness. With Bezier curves, we could not have hard bends or non-smooth corners, due to the control point. Regarding the round splines, the paths are always straight lines that connect nodes and are never round. So, the solution found to create the best roads possible, was actually very simple.

To generate the road mesh, the procedure divides the distance between 2 consecutive nodes by a predefined value, to determine the number of segments this piece, of the entire road, will have. Then, the length of the segments is calculated. All the segments have the same size, so to determine their length, the algorithm simply divides the distance between the two nodes by the number of segments discovered before. When the length is found, the procedure iterates from the initial node to the next by the length of each segment and creates a control point on that position, storing it on a list. When this control point is created, the procedure create 2 vertices that will belong to the road's mesh. Each one, is placed on each side of the road at a distance from the control point equal to half the road's width and perpendicularly to the path's current direction. The Y coordinate of these points, which is the height, is determined by casting a ray down, from the highest Y possible on that point's position and getting the Y coordinate of the hit point with the ground. This procedure is repeated for each pair of nodes.

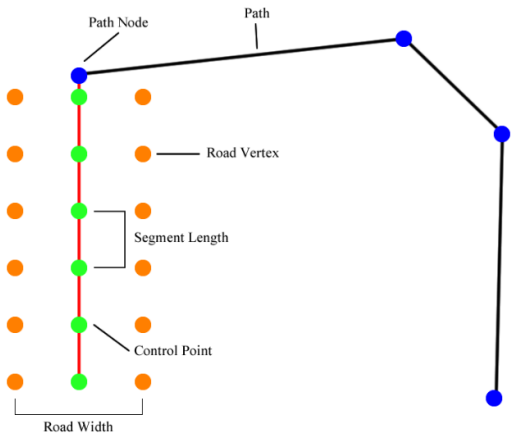

Figure 24 - Procedural Road Vertices Placement

Now comes the solution found. When the entire path is processed and the points stored, the next step is to smooth the path, to make it look more realistic and to avoid sharp square corners and possible overlaps caused by them. As explained before, a different method had to be used. This method attempts to smooth a road by making an average of its points, with some rules. The first set of points to be smoothen are the control points, because they represent the middle of the road. The procedure picks the first and the third control points and calculates their average position. The second control point's position, will be equal to this average and its height is calculated using the ray cast method, mentioned above. Then, the procedure goes on to the next point and the process repeats. For the road vertices, the average is calculated the same way, the only difference is that the point needs to maintain its distance from the control points (half the width of the road). If the distance is smaller or greater, then the point is translated by a value equal to the difference between the road's width and the point's position and once more the height is calculated.

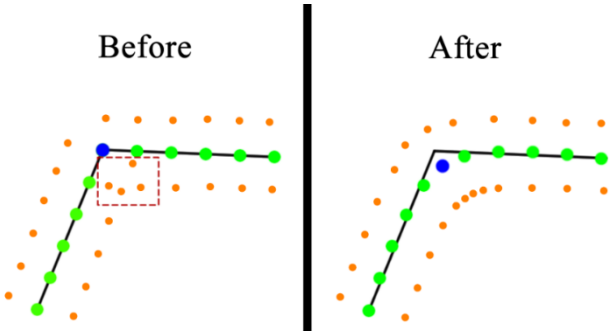

Figure 25 - Procedural Road Before and After Smoothing Operation

This process is repeated a determined number of times, called "smoothing level". After a few testing, it was found that the smoothing level that offers the best results is 8.

In case the path is a closed one, for instance, a roundabout, there is an extra step, which is to include the smoothing of the first and the last vertices of the road.

After the smoothing operation, the vertices are ready to be used to build a mesh. The mesh is built by getting 2 vertices from the vertices list that belong to the left side, 2 control points and 2 more vertices that belong to the right side.

Unity uses the left hand rule, meaning that the vertices should be ordered clockwise. This grants that the normal of the generated polygon points up and the front face is visible.

Having this in mind, the road is generated and the front face is visible. The road is composed by multiple polygons, all well-arranged, well positioned and dependent from each other, but, to Unity, they have no relation with each other, they are simply separate pieces of a puzzle. This is very bad, because there are so much separate pieces in the scene that the frame rate will suffer a massive penalty. To fix this, the road mesh is merged into a single mesh, using a function from Unity Community Wiki. For Unity, processing thousands of very small meshes is much slower than processing a huge and complex mesh with thousands of vertices.

Regarding the sidewalks, they are generated exactly the same way as the roads, the only difference is that they have an offset equal to the road's width and do not have control points. Therefore, to generate a sidewalk the algorithm gets 2 vertices from the sidewalk vertex list and 2 more from the road, belonging to the respective side. Having these 4 vertices, the sidewalk height is added to each vertex's Y coordinate, generating a polygon with 4 vertices. There is also a step that closes the gap between the road and the sidewalk. This step's vertices are equal 2 by 2 to the vertices of the sidewalk and the road they are going to join, respectively.

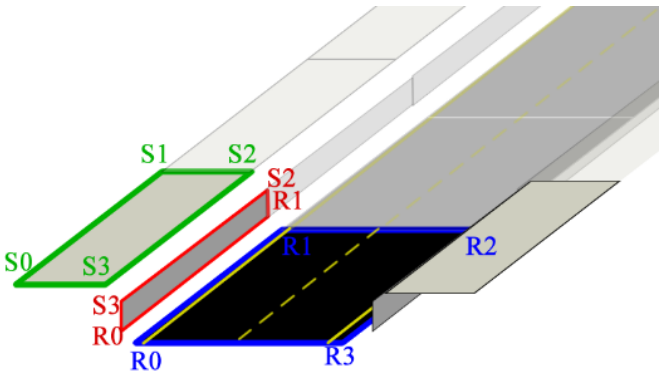

Figure 26 - Sidewalk and Step Generation - Vertices relation with each other

As seen on the image above, the points that are used to generate the sidewalk step, all belong both to the sidewalk and the road, so its generation is very simple and easy. To follow the left hand rule, the vertices are processed on the following order: S3, S2, R1, R0 for the left sidewalk step. And for the other side: R3, R2, S1, S0, where S1 and S0 are not the ones represented, but instead the ones that belong to the right sidewalk step.

a. Road Enlightenment

Road enlightenment is a key feature on road safety, both for pedestrians and drivers and its presence and type depend on the type of road they are illuminating. Residential areas always have some sort of lighting, on the other hand, highway/motorways are almost never lit, such as city links, very rural areas, secondary and tertiary roads. Due to its importance, it is fundamental to have illumination on the procedural roads.

The illumination is generated by two types of street lights. The first ones are regular street lamps that can be found on city locations. The second ones are out of city illumination, therefore they are street lights like the ones found on highway/motorway links. Each one of these lights can cast shadows, and be displayed with different levels of detail, depending on the distance to the main camera.

Figure 27 - Street Illumination 3D Models

City Lamp on the left and out of city light pole on the right

The lights are placed close to the road, usually on the sidewalk step, but they can be offset, to be placed further away from the road itself. They are placed on a regular interval, on both sides of the street, and are generated as soon as the road they are going to be placed on is finished rendering. To place the lights, the algorithm simple gets vertices of the inner part of the sidewalk on a regular predefined interval until reaching the end of it and places a street light on that vertices position. When all the roads are rendered, there is an algorithm that gets all the street lights and, for each one, it casts a ray down from the position of the light and if this ray crosses more than one road, then probably this is a zone where two or more roads overlap, which means that the street light needs to be removed from the scene. This prevents having a street light in the middle of an intersection for instance.

This project also features a full day-night cycle, meaning that the lights turn on and off when needed. When the sun is rising, a messaging system sends a message to all the lights to turn “off”, when the sun is setting, the process repeats, but sending the message “on”. The reason for sending a message, instead of a polling operation, is that every single light would have this polling operation on every frame, this would cause a massive drop down on the frame rate specially when

there are a big number of lights, so instead all the street lights have an event listener that reacts only to “on” and “off” messages.

3.5.9 Pedestrians

When driving a vehicle, pedestrians usually have priority over it, when it is allowed. They interact with the roads and some of them may be too distracted to pay attention to the surrounding environment, for instance, if they have headsets or are on the phone. Another thing, is that some pedestrians may not have a driving license, therefore they do not know the traffic rules, so they think they can cross a road anywhere knowing that there should be a road crossing somewhere. For last, pedestrians are very unpredictable, especially children. A driver should be paying attention to the road, to the surrounding vehicles and to the pedestrians, but contradictorily, he should be capable of predicting other drivers and pedestrians behaviours, to increase road safety.

This project implements a simple pedestrian system that attempts to simulate small crowds of different persons with different behaviours within some limits.

When the road finishes being generated, a script is attached to every walkable element, in this case the sidewalks and the footways. This script, stores their vertices, so they are easily accessible. This is done like this, because all the information regarding the roads is destroyed as soon as the road finishes generating, to save resources. So, there is the need to store the relevant information elsewhere, in this case, on the attached script.

When all the roads are generated and all the walkable areas contain their own vertices stored, an algorithm that will generate the pedestrians, is called. This algorithm generates pedestrians based on the pedestrian density, configured on the main menu. This density is relative to the size of the location selected, to prevent an extremely large crowd that would kill the frame rate. If the map is made just by one tile, the amount of pedestrians compared to a 10 tile map, for the same density, would be 10 times smaller. The pedestrians are not distributed on equal number per tile, but instead, they are randomly generated along the scene. Although the density is relative to the location’s size, there may be smaller or larger crowds on some places. The objective is to have harder places to drive on and some other easy, so the driver can relax.

The generator picks a random walkable area, a random vertex belonging to that area to define the position where the pedestrian is created and a random human character. Each pedestrian has an AI script attached that receives, as parameters, the game object the pedestrian is walking on. Also, this walker is added to a level of detail controller explained below.

There are six different walker 3D models, three female and three male, with three animation states each. There are walking, running and idle animations.

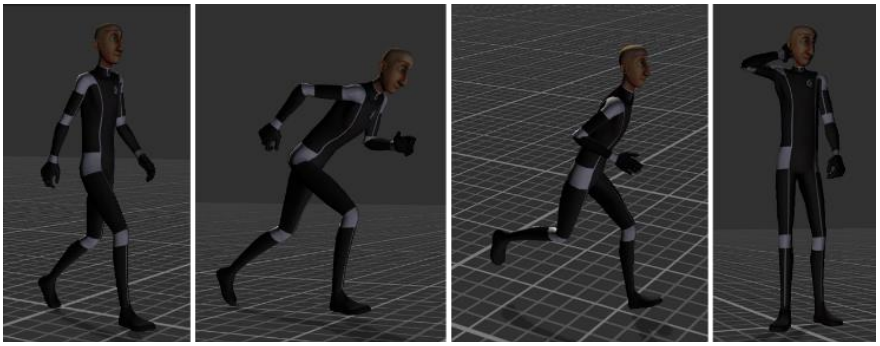

Figure 28 - Pedestrian Animation Skeleton

Walking – Running – Walking 2 (Jogging) - Idle

Each one has its own AI script, whose values are configured automatically and randomly, within certain limits. These values are the walking and running speeds, the force the walker does to avoid obstacles, the ahead visibility and the acceptable node distance that indicates the limit distance that the pedestrian has to pass from its next node on its route. The walker can be in one of four possible states that identify whether the pedestrian is idle, walking, running or has been hit by the vehicle. If the pedestrian does not have a path defined yet, then it will calculate its own path over a walkable surface. A pedestrian always has a walkable game object associated, but that does not mean that its path along that walkable surface is calculated. If that path is not defined yet, the algorithm iterates along the entire walkable surface, calculating an average of 4 vertices of the walkable area (2 from the inside and 2 from the outside), adding this average point to a list, until the last set of 4 vertices is reached. By this method, every pedestrian on the same walkable area has the exact same path, so to avoid this and to make their movement feel more natural a random offset is added to the average points of their path. This offset is a random value between the sides of the walkable area and since this area's width is constant along the entire mesh the offset value can and should be constant, therefore the first two vertices of the walkable area can be used to calculate this offset (vertices 0 and 1). When the path finishes calculating, the pedestrian decides randomly to go left or right and finds the closest point to his position, which will be his first target. Then it can walk from node to node. To accomplish this, the pedestrian selects the next node he needs to go to depending whether it is going left or right. Having the next node selected, it analyses the scene ahead of it, using a defined distance of vision and attempts to calculate the force it needs to avoid the objects in front. The force is calculated as a vector on space, equal to the difference between the detected object's position and the pedestrian's position. To detect an object, the pedestrian casts a ray forward with a distance equal to the ahead visibility. If the avoiding force isn't enough, then its value is doubled. Another thing the pedestrians do, is avoid being out of a walkable area by running from non-walkable areas to walkable ones. As soon as the walker detects it is on a road, it runs to the next point of its old path. A non-walkable area is always a road where traffic moves, other areas that are not considered walkable do not interfere,

so the pedestrian does not feel the necessity to run. Having analysed the environment to calculate the avoidance force and determined whether to run or not, the movement force is ready to be applied to the character. The walker is always looking to his next point of the path and moves forward with walking or running speed, depending on his state. After applying movement to it, the avoidance force is applied, to make the walker move left or right depending of the force signal.

When the pedestrian is over more than one walkable area, it has the freedom to choose whether he wants to change to another walkable area or keep on the same one. If it changes to another one, the walkable area vertices are retrieved from the identified game object and the old walkable vertices that the walker had stored are replaced by the new ones, then the path along the new walkable area is calculated and the walker changes its route.

Till the pedestrians are hit by the vehicle, they do not have any kind of physics applied. They are just game objects that move at a certain speed and fall over heights, because there is a programmed constant velocity (the move function receives velocities, not forces) pulling them down. When they get hit by the vehicle, their movement characteristics and animations are disabled and they turn into a ragdoll. A ragdoll is a physics puppet, where each member of the body has physical properties such as weight and joints to other parts that can break and have torque limits, making the character a free physics object, where each part of its body can interact with any other physics object. This changes their state to the "hit by car" state and counts a casualty for the statistics.

The ragdoll cannot be enabled or disabled, it can simply be disguised by the animations, so at each Unity's physics update the physics calculations for each pedestrian are done, but not applied. When the number of pedestrians reaches hundreds on the entire scene, the frame rate drops a lot. To fix this, a level of detail (LOD) controller was created. This LOD controller has a reference to all the pedestrians in the entire scene, stored in a list and at each frame the algorithm runs through the list and for each one, it calculates the distance between it and the main camera, if the distance is bigger than a certain value (in meters), then the pedestrian is disabled, otherwise the pedestrian is enabled. Disabling a game object removes that object from any processing routines or scripts, so there are no calculations at all for the disabled, it is like the object does not exist, although it is not removed from the scene. By doing this, only the pedestrians at a certain range are processed and visible, but when they are outside that range they stop being processed. By removing the ragdoll from the characters, a much greater number of visible walkers could be achieved, but then there would be no kind of interaction between them and the vehicle.

3.5.10 Procedural Buildings

This project has the objective of creating a tool that can recreate any place in the world, in real time. Having roads, walls, fences and so on with no buildings may feel a little odd. So, one important aspect is the building generation from a simple closed path/way. Every procedural building has the basics of any architectural structure such as walls, floors, basement and windows. The roofs are not implemented, to save computational resources. Since this is a driving simulator, everything happens on the ground, so having rooftops is unnecessary, because they will never be visible.

As written before, a building is generated from a closed path/way. According to a study, the building's height is usually more or less proportional to the bottom's area, therefore the height of a building must depend somehow on the area of this path. After a few research, it was concluded that, in average, a building's height can be up to 4 times the bottom area, divided by the height of the floors. In this case, since the area is a complicated calculation, due to the shape of the path, the algorithm uses its perimeter and a few random factors to determine the height. The first one, is to simply add one more floor or not, so buildings with the same perimeter may be different and the other is the number of surrounding buildings. This last one is based on the assumption that the more buildings around, the bigger the probability of being in a big town, the less buildings, the bigger the probability of the surrounding buildings being houses and not skyscrapers.

In this project, a building is considered a skyscraper, if it has more than fifteen floors. Skyscrapers are usually buildings with lots of windows and almost no concrete walls visible from the outside. Therefore, they will be different from regular buildings, because the regular ones have concrete walls and much fewer windows, but on the other hand they are computationally heavier to process as explained below.

The perimeter of the path, is a key feature in what comes to defining the appearance of a building, because it defines the seed of a random generator in the very beginning of the building construction. It means that for the same building/perimeter the final result will be unique and always the same execution after execution, this value is responsible for determining all the materials and the number of floors.

The next step to generate a building, is to determine whether it is a skyscraper or not and select a material accordingly. The materials of a skyscraper are different from a regular building. To simplify and save resources, the material for a skyscraper has a texture with lots of windows and lots of glass, with almost no concrete structures visible. On the other hand, the texture of the regular buildings does not have windows, it only has the basement texture and the concrete walls texture, meaning that the windows and the door will have to be generated as a mesh. By generating a window mesh, instead of using a texture it is possible to configure its size and position along the walls of the buildings. After the material is selected, if the building is not a skyscraper then the door and windows material is also selected.

The door is generated on the biggest wall of the building and placed on the average point between the two nodes that make this wall. It is also offset a bit forward from its original position, to prevent wall overlapping. Its position is then stored for further use when generating windows.

The basement is what allows the building to be perfectly levelled on an inclined surface. This means that the basement will search for the lowest and the highest points to calculate the basement's height, based on the difference between those two. After determining the points, the entire path is moved up or down, to modify its Y coordinate to the highest point's Y coordinate, allowing to build the floors, from the path upwards, instead of the original path's position upwards.

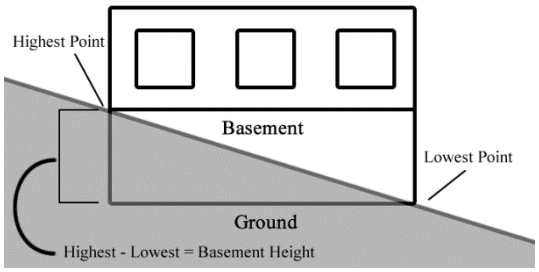

Figure 29 - Procedural Building - Basement Generation

The basement's mesh is generated by iterating along the entire path and creating a procedural four vertices plane, which uses the vertices from the path and the highest and lowest points to determine the height. Now, it is possible to cover the entire basement, from the highest to the lowest point, guaranteeing that there are no gaps between the basement and the ground, so the vehicle or the walkers can never go under the buildings.

Now, the procedure is ready to generate the floors and the windows for each floor, for regular buildings only. To generate a floor, the procedure uses the same structure and logic as the one used for the basement, but instead of using the highest and lowest points, it uses the ground landed point and the floor height to generate a plane. This process is repeated until the starting vertices are reached. When a floor is finished, another procedure is called to create the windows, in the case of a regular building, on the other case this step is skipped. The windows are generated along the surface of the walls and depending on the available space a wall can have more than one window or no windows at all.

A window needs a few parameters to be correctly generated. The first one is the window size in meters. In this project, windows are considered square polygons, for simplification, therefore only a size is needed for both height and width, since they are the same. Another parameter is the margin, which defines the minimum distance another object can be, from any border of the window. The third is the window spacing. The window spacing is the space that the window occupies, both vertically and horizontally and it is equal to twice the window margin plus the window size. Dividing this value by two, we obtain the minimum distance, some other object

needs to be apart from the window, centre to centre. The last parameter is the offset, which is the gap that should exist between the window and the wall and it is useful to prevent mesh overlapping, because when meshes overlap, the graphics engine does not know which mesh it should render first, so it flickers on the overlapped area.

To generate the windows for a wall, the algorithm iterates along it at a value equal to the window spacing and sums this distance to a "distance counter" until the "distance counter" is greater than the wall's length. On each iteration, if the window is not going to overlap the door and, if the "distance counter" is not greater than the total wall length, then a window is created, because there is an available space. The window's mesh is generated just like the door. The four vertices are calculated and defined around the centre of the window (window spacing over 2) and used to generate a procedural polygon that is offset from the wall to prevent the overlapping problem. At the end of the cycle, the windows are equally distributed along this wall.

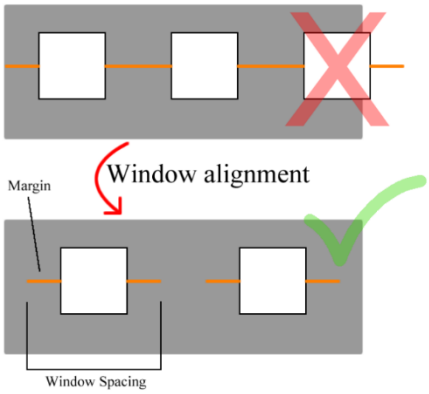

Figure 30 - Window Generation and Alignment

The same process repeats for the other walls, using the same rules. To find out if the window overlaps the door, the previously stored value of the door's position is used to calculate the distance between the door's centre and the window's centre. If this distance is bigger than the window spacing, then the window and the door do not overlap, therefore the window can be generated.

When the floor is generated with all the windows positioned, the algorithm moves to the next floor until reaching the last one, applying the same procedure, but incrementing the height of all vertices by the current floor times the floor height (the floor count starts at zero).

3.5.11 Procedural Barriers

In this project, a barrier is considered anything that is extended over a path that restrains moving objects of going through it. In this case, barriers can be walls, fences and guard rails.

This project has an algorithm that can be configured to create the three types of barriers, allowing to configure the thickness of the barrier, its height and its material. If the thickness is configured to be zero, then we are talking about a fence or a guard rail, otherwise it is a wall.

Barriers are generated the same way the roads are, there are control points along the path and a fixed height value. In the case of fences, it is simple, the algorithm just adds a vertex on top of the control point, with a distance equal to the height value and when all the path is finished processing, to generate the mesh, the procedure receives the first and the second points from the bottom and the second and the first points from the top. The order of the points is then inverted, so the mesh is visible from both sides and the algorithm moves to the next node until it reaches the end.

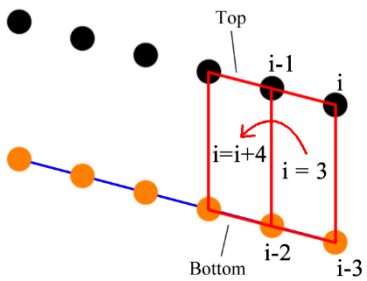

Figure 31 - Procedural Fence/Guard Rail Control Point and Vertex Generation

As shown on the image above, the procedure starts at the first node of the path and creates separated control points from each other, at a fixed distance, until it reaches the next node of the path. When a control point is created, the procedure creates a point on the same position as the control point, but increasing its height (Y value) on a predefined value. To create a control point, the algorithm iterates at a fixed distance along the path and on each iteration, it casts a ray down on the current position from the highest Y value possible and creates the control point on the collision point between the ray and the ground. The reason a barrier is created like a road, using multiple control points along the path, is to make sure the mesh adapts to the ground elevation, avoiding any gaps between the mesh and the ground.

Building a wall has the same basis as the fence and rail generation and the same iterations, with the exception that the vertices are doubled and half of them are offset to the left and the remaining to the right, half the wall's thickness/width. In the end, the upper part of the wall is generated by creating a mesh that uses the top right and top left vertices.

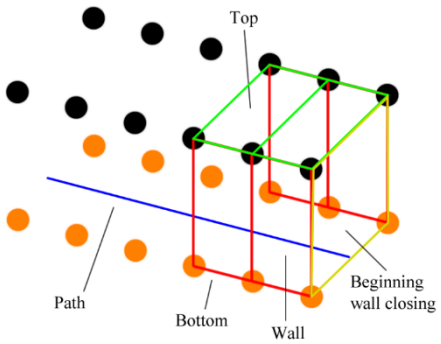

Figure 32 - Procedural Wall Control Point and Vertex Generation Orthogonal View

Control Points on orange and Vertices on Black, placed side to side along the path

The picture above, is an orthogonal view of a wall. The red lines are the lateral wall, the green are the top of the wall and the yellow are a lateral wall that closes its frontal face (there is another in the end of the wall). The orange points, represent the control points and the black points are placed over the respective control point at an equal distance to the wall's height. The points on the left are separated from the points on the right by an equal distance to the wall's width.

When the mesh is generated, the material is selected accordingly, depending whether it is a wall, fence or guard rail.

3.5.12 Procedural Land Uses

According to OpenStreetMap, a "land use" is a tag mainly used to describe a primary use of the land by humans or by nature, so every vegetation, forest, farmlands, cemeteries, grass, green fields between much others are tagged as "landuse". A land use is a closed path where objects lay, on a random or relative position.

The main concern of this project on what comes to land uses, is the vegetation. It is important to have vegetation on the scene, not only to increase the visual appearance and enrich the scene on realism, but also to create obstacles on off-road paths.

To create vegetation, the project uses a few algorithms, first to create and define the area where the objects will be placed on and then place the objects on this area, on a random position.

In order to place the objects on the area covered by the path, a game object that is capable of filling the path's area needs to be created, to identify whether the objects are going to be placed inside or outside the area. Let us call this object "FloorPlant".

To generate the FloorPlant, the responsible procedure picks up four points that belong to the path, the first two and the last two. Having this four points is enough to generate a polygon with

four sides. So, this polygon is generated and it should cover a small area belonging to the path. This process repeats by advancing to the next nodes, in this case, the third and the fourth nodes with the last minus 3 and last minus 4 until the entire area is covered. This works perfectly in the case of a path with an even number of nodes. When the number of nodes is odd, the last polygon will not have 4 vertices, but 3 instead. This is generated after all the iterations and will close the remaining area by creating a polygon between the point that does not belong to any mesh and the last two points used on the last iteration. This entire process needs to repeat inverting the order of the nodes, because since we do not know anything about the position of the vertices we do not know whether the mesh is going to be generated clock or counter clockwise, therefore we do not know what face will be visible. To avoid this, it is better to generate the mesh for both sides. The terrain elevation is not important in this case, as explained below, which means the meshes will be flat.

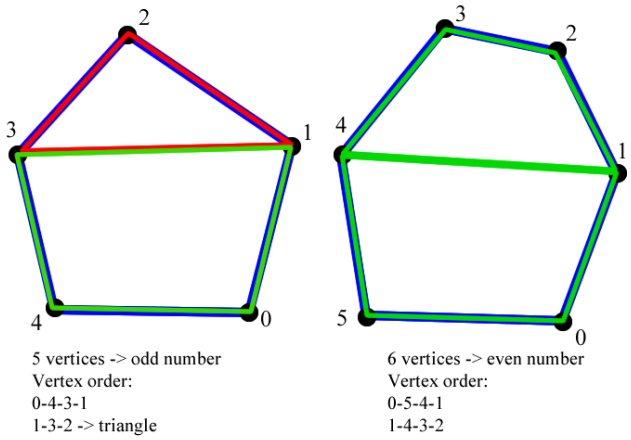

Figure 33 - Procedural FloorPlant Generation

Odd number on the left, even on the right. The odd number generates a triangle on the last iteration while the even generated another 4 sided polygon

The reason the FloorPlant is created is because the objects will be positioned using the ray cast technique. By casting a ray, it is possible to know whether it will intersect the FloorPlant and the ground. If it intersects both, then the object can be placed, because it is inside the area, otherwise the object cannot be placed, because it is either outside of the map or outside the path's area.

When the floor plant is completed, it is time to generate the vegetation. The vegetation is composed by bushes and by trees, with a very low polygon resolution, to avoid a frame rate drop.

Figure 34 - Tree 3D model on the left and bush 3D model on the right

There are only being used two models, one for the trees and one for the bushes, because, first of all, it is not important to procedurally generate the tree or bush meshes in this project, since its goal is to procedurally generate a world location, not necessarily all the location's assets and secondly, it is just a detail, it is more important to have quantity instead of variety.

For starters, just like for the buildings, the perimeter of the path is calculated and the seed of the random generator is set to the perimeter value, which guaranties that the vegetation will be distributed randomly, but at each execution, for the same path, the vegetation objects will always be in the same place.

To generate the vegetation, the algorithm gets the bounds of the area in order to know what distance needs to be covered on X and Z axis. Then, a cycle starts from the minimum bound until the maximum bound iterating at a value that depends on the density. The bigger the density, the lower this value and the more vegetation objects will be rendered. On each iteration, the algorithm generates a random value that identifies the probability of the object being generated at this position. This probability depends on the seed of the random generator. If the object is going to be placed, a random value is added to its position and then an instance of a vegetation object is created. This object needs to be placed on the ground taking in account the terrain elevation, so a ray is casted downward from the objects position X and Z and the highest Y value possible. If the ray hits the ground, the position of the vegetation object is equal to the hit point. Another condition mandatory to place a vegetation object, is that when casting the ray, it cannot hit any road or sidewalk, therefore the land use generation procedure needs to execute after all the roads are complete.

When the area is covered, the FloorPlant can be destroyed, because it is no further needed.

3.5.13 Procedural Amenities or City Furniture

An amenity is a community facility such as, garbage bins, bus stops, health facilities, education facilities, monuments, banks, toilets and any other kind of urban furniture. The amenity tag, is the tag that has the most keys on the OpenStreetMap system. That is why this project does not cover almost any of those amenities. There are hundreds of different objects possible and lots of combinations, making it a very complex task, with a large number of 3 dimensional models to group with each other, get the entire coverage of all the amenities. This cannot be accomplished on the context of this project, because hundreds of 3 dimensional models need to be found for free, with relative good quality and low polygon count and then, they all need to be processed, recognized, and merged with each other based on "if clauses". All of this work, would require almost the same time this project took to be accomplished, so only very little different objects were considered, in this case, only the trash and recycle bins.

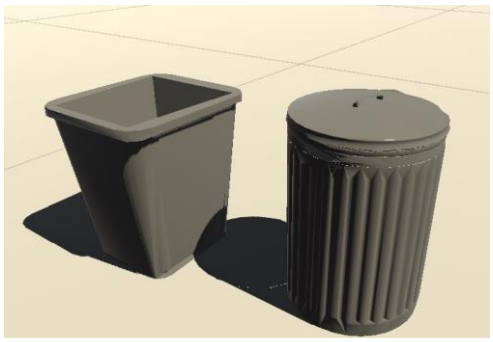

Figure 35 – Garbage Bin 3D models

This objects are placed on the map, but they do not correspond to a path, they correspond to single separate nodes. So, the algorithm just needs to identify the tag value and instantiate the corresponding object on the position of the node. Both of this objects, contain a rigid body and have colliders attached. The rigid body allows the object to behave physically, applying the gravity force and external forces upon collision.

3.5.14 Day and Night Cycle

Driver awareness and reaction time are often affected by the global illumination and sun position relative to the driver's eyes. When the sun is low, it can strike the driver directly in the eyes causing temporary blindness, making the driver not able to see, due to the intensity of the sun's light, therefore there is an extremely increased difficulty in seeing pedestrians, traffic and objects. This project attempts to recreate such conditions. This can be found in more detail in the Visual Effects section, while this section's objective is to describe the algorithms behind the day-night cycle and their capabilities.

The sun is a game object that rotates on the X axis of the Unity's scene. It rotates a percentage of the day's duration every frame. The day's duration is configurable on the main menu and its units are minutes. Having this in mind, it is possible to calculate the rotation of the sun on each frame:

DayCycleInSeconds = dayCycleInMinutes (from the menu) * 60(minutes)

RotationEachFrame = (360º/DayCycleInSeconds)*Time between 2 consecutive frames

The RotationEachFrame is applied on the X axis every frame, making a smooth sun movement along the entire day.

Unity has a feature called procedural skybox. This skybox is generated by a shader, whose values can be configured, such as the sky's colour, atmosphere thickness, sun size and exposure and it can detect the main directional light on the scene (the sun), to create the sun's representation on the it. As the sun rotates, the skybox shader blends its colours to recreate the actual sun effect on the atmosphere, meaning that, for instance, starting from noon, the skybox gets more orange and darker as the sun goes down.

This algorithm also includes "events". This events occur at a defined time of the day. There are two events, sunrise and sunset, determined by a percentage of the day's duration from 0 to 1. The sun starts always at midnight, so this percentage is zero, at noon the percentage is 0.5, so from there it is possible to configure the time when this events happen by just changing the percentage, to when we want it to happen.

These two event are used to send a message to the street lights. When the sunrise event occurs, the algorithm sends a message containing "off" to all the street lights, this will make all of them turn off. On the other hand, when the sunset event occurs, the algorithm sends the message "on" to all the street lights.

Another important feature is the global illumination. In Unity 5, global illumination is not generated by any light source, instead it is generated by the skybox, by a colour or a predefined gradient and its intensity can be configured. In this case, the global illumination is being generated by the skybox. So, while the sun goes up, the global illumination starts going up to the maximum to create a nice and bright day, when the sun starts coming down, the algorithm does the opposite and the global illumination starts going down till the minimum, to create a really dark night. At night, the illumination depends only on the street lights and on the intensity of the moon's light.

This algorithm supports more than just a single sun, in fact it supports an unlimited number of suns. A sun can also act as a moon. This is because a sun is simply a directional light that is positioned at infinity, so its intensity, colour and flare can be configured to make it look like a moon. The moon is positioned on the opposite side of the sun and rotates at the same speed and axis as the sun, meaning that there will never be an eclipse.

Another important feature is the starting time. On the main menu the user can configure the starting hours of the day, ranging from 00:00 to 24:00. When the user configures the starting time,

the sun is rotated to the correct position, such as the moon, and the global illumination is configured accordingly as well.

3.5.15 Weather Cycle

Weather conditions are the main reason of road accidents, right after driver neglects, such as excessive speed, driving drunk or lack of attention, so it is important to recreates a few weather conditions that affect visibility, pedestrian behaviour, road conditions and vehicle tire grip, causing the driver to increase the reaction time and the vehicle to increase braking distance and decrease its stability.

The featured weather conditions are sunny, cloudy, rainy, and stormy and any of these conditions can have fog added to the mix.

On the main menu, the user can select what weather condition he wants to happen on the simulation, as well as setting the weather changing probability. The weather changing probability, is a number ranging from 0 to 0.9 and controls for how long a condition is present until changing to another. The lower the change probability, the longer a condition will stay on the scene. The duration and the transition between conditions, also depend on the duration of the day, meaning the longer the day's duration, the longer the conditions will maintain and the longer will be the transition between each one of them. On the other hand, let us say, for a full day cycle that lasts one minute and for a changing probability of 0.9, everything will happen very fast and for a changing probability of 0.1 the conditions will last longer, but the transitions will be as fast.

Another less important aspect is the sound of the rain and the lightning, which contributes for a more immersive experience. There is also a flashing light when a lightning strikes.

Regarding the vehicle, it is important to have different behaviours and responses on different weather conditions. When it is raining, the grip of the tires is reduced almost by half, like in real life, making the vehicle harder to drive. Having different tire responses, the vehicle's response, acceleration and braking distance will be affected as well. The vehicle's response will also depend on its wheel base, axel distance and drivetrain. To adjust the tire grip, instead of changing the tire friction values, it is easier to change the stiffness of the friction curves. The stiffness is basically a friction curve multiplier that increases or decreases all the values of the curve.

The fog is one important factor, and Unity has a limitation in what comes to it. Unity renders fog as a post process image effect, it is rendered after everything else and in image space, so it picks every element of the scene, calculates its distance to the main camera and applies the fog according to the distance. The problem is that the skybox is not a game object, so no fog effect is calculated on it. The algorithms attempt to blend the skybox's colour to a more or less uniform grey tone, to overcome this problem.

3.5.16 Vehicle

The idea is to have different types of vehicles with different characteristics that the user can select, on the main menu. For now, this project has a sports car, with rear wheel drive and a 4x4 pickup truck, with lifted suspension and larger wheels, in order to explore different wheel bases, axel distances, weights, braking distances, accelerations, drivetrains and sizes.

a. Vehicle Selection

In the main menu, the user has a button that sends him to the vehicle selection screen. This is a new scene, where the user can select the vehicle using the navigation arrows, left or right, to cycle through all the vehicles available. On the left bottom corner, there is a button to select the currently visible vehicle and go back to the main menu. There is also a small panel that shows the currently visible vehicle's characteristics, such as dimensions, power and drivetrain.

b. Vehicle Handling

The vehicle handling depends on many parameters, especially the tire friction curves. These curves determine the vehicle's tires capabilities and limits, in what comes to grip by identifying the slip values and force needed to make the wheels spin/slide, both sideways and forward. Each wheel has two friction curves, corresponding to the sideways and the forward grip.

The suspension system is also part of the wheels. Each wheel has a suspension component that can be fully configured, including the suspension height, suspension damping and spring force. By adjusting these values, the vehicle's stability and response to bumps changes, making it stiffer of softer on a bumpy road.

The vehicle's drivetrain is also a very important part, because it determines whether the vehicle is front wheel drive, rear wheel drive, 4x4 or all-wheel drive and depending on its differential, the vehicle assumes different driving strategies and behaviours. For instance, the front wheel drive (FWD) vehicles tend to suffer from understeer, while the rear wheel drive (RWD) tend to suffer from oversteer. The 4x4 or 4WD may suffer from both, but can easily find a balance to maintain maximum performance on corners and also have smaller acceleration times, due to the fact that the 4 wheels are pushing the vehicle. Just for reference, understeer is when the front wheels of a vehicle exceed the slip that provides maximum friction. When this value is exceeded, the friction values decrease meaning the vehicle starts sliding instead of cornering. On oversteer the same happens, but on the rear wheels, making the rear end lose forcing the driver to counter steer, to get back to control. The 4x4 is the midterm, because usually the differentials that connect the front and rear axles, attempt to distribute engine torque to the axel that needs it most and, therefore increase stability. A good example of a mechanical differential, is the Audi Quattro system that has a clutch that moves forward or backward, depending on the rotations of the axels.

If the front wheels are spinning faster than the rear wheels, the rotational force of the differential pushes the clutch towards the rear axle, making the torque provided from the engine, go in greater percentage to the rear wheels. The inverse happens when the rear wheels have a greater rotation then the front wheels.

c. Vehicle Engine

The engine is responsible for producing the torque that is applied to the wheels, by means of a mechanical system. First of all, the engine has to be turned on. To turn on the engine the driver rotates the key that activates an electric system, capable of rotating the engine pistons and creating a spark on the ignition chamber. When the pistons start rotating, they force the intake valves and the fuel valves to open and close, depending whether the piston is up or down respectively. With the piston up and the valves open, the fuel enters the ignition camber and ignites due to the spark and the pressure, making the engine start running on its own. It runs on its own, because the force of the fuel explosion/combustion forces the piston down, making other pistons go up and the cycle repeats. This is a 2 stroke engine, because it only has 2 phases, the ignition and the exhaust phase. A four stroke engine, which is the most regular on vehicles, is capable of producing more power and efficiently, because it introduces 2 more phases, the admission and the compression. A four stroke engine, starts by sucking fresh air into the pistons (admission), while the piston goes down and, when it comes up, the compression phase occurs, because the air gets compress. When the compression is maximum, the fuel is ignited (ignition) and the piston goes down with the force of the explosion. When it goes down, the dirty air inside the camber, which is the result of the explosion, is decompressed and, due to the other pistons ignition, this piston returns up forcing the dirty air into the exhaust system (exhaust phase). The valves are mechanical systems that are connected to the engines camshaft and are fine tuned to perfectly control the entrance and exit of air and fuel. Some vehicles also have a turbo. A turbo is a system connected to the exhaust system that receives and takes advantage of the dirty air, to rotate a fan that has a tube connected to the admission system, to eject fresh air at a much larger rate. The more fresh air inside the chamber, the best the force of the explosion, therefore the better the performance. Another system to eject air into the cambers is the supercharger. A supercharger is usually found on American muscle cars and is outside the front bonnet. Inside, a supercharger has a fan that is directly connected to the camshaft, rotating at the same speed than the engine and sucking air from the outside at a rate as large as the engine's RPMs. This kind of systems, may not be adequate for non-muscle cars, because a muscle car is known for not being fuel efficient, and since the supercharger is connected directly to engine, it drains out a bit of horse power, making the engine push harder, increasing fuel consumption. On the other hand, the turbo rotates due to the pressure of the air that comes out of the engine and, therefore does not drain any power from the engine.

With the pistons moving, the camshaft rotates. To move the vehicle, there needs to be a drivetrain connected to the camshaft. This is done by using a belt that is connected to the camshaft

and the drivetrain. The power is then transferred to the gear box. At the beginning of the gear box, right before the gears, there is the clutch and the fly wheel. The flywheel is always rotating at the same rate has the engine, while the clutch is controlled by the clutch pedal. When the clutch is pressed, it makes the engine run freely, by separating the connection on the flywheel. When the clutch is released, it makes contact with the fly wheel and starts skidding until it reaches the same speed. This is why vehicles stall. If the clutch pedal is released to soon, the clutch does not have time to gain enough rotation and the system blocks, because the transferred power is not enough to rotate the wheels. Since the engine is always running, just as the flywheel, the pistons are forced to slow down to a rate that the engine cannot enter in a cycle, therefore the vehicle stalls. With the clutch engaged, the power starts being transferred to the gear box. A gear box, is a complicated and complex system that consists on multiple gears and can be manual or automatic with double or single clutch. Inside the cockpit, the driver selects the gears. Each gear, has its own ratio. This ratio is used to multiply the torque that comes from the engine. By multiplying the torque, it divides the maximum speed the vehicle can reach. This is, the bigger the ratio, the bigger the torque, but the smaller the speed the vehicle can reach. By this logic, the first gear is the one with a bigger ratio, therefore it is used to start moving the vehicle. After the gearbox, the torque is transferred to the differentials of the traction wheels. The differential, makes possible that the wheels of an axel, spin at different velocities, making cornering and getting out of slippery surfaces much easier. The differential selects the power that needs to be transferred to each wheel, depending on the type of differential.

This was just to explain the basis on how a vehicle works and to better understand how the vehicle scripts is coded, because it attempts to follow the closest as possible the above procedure.

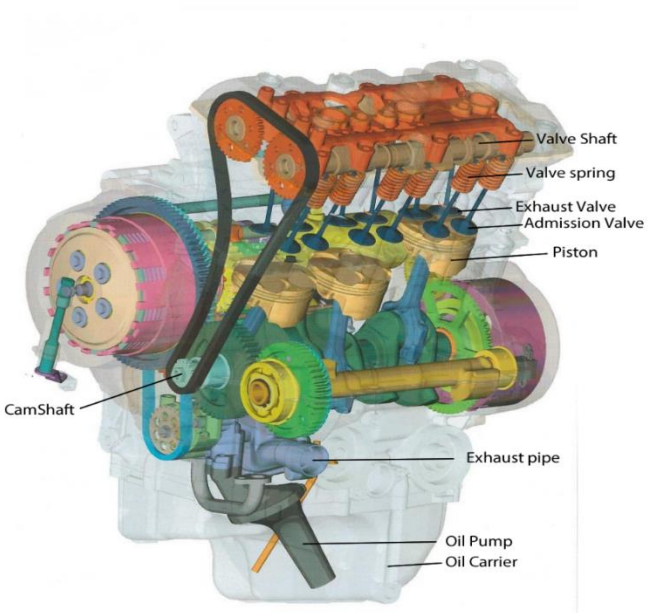

Figure 36 - Triumph Daytona 675 - Naturally Aspirated Engine (Adapted from [101])
675cc 12v in line triple four-stroke capable of 123 horse power

Fuel engines have multiple parts creating frictions and have recovery times from low to high throttle, or vice versa, and additional torque created by the turbo system that enters the mix, in a non-seamless way, therefore these engines generate a torque curve that is not constant. On the other hand, electric engines produce almost a constant torque curve, providing almost full torque at the entire RPM range, because ,not only they are made of a single rotating piece, but also because the system generates maximum electric intensity and voltage almost instantly when the pedal is fully pressed.

The torque curve, in this project, is represented on a file inside "CarCurves" and consists on an association RPM-Torque. What this means, is that for a certain RPM of the engine, it generates a determined amount of torque. The intermediary values of torque are interpolated.

The vehicle script loads this torque file and stores this information on lists. It has also defined the gear box ratios, max and idle engine RPM, brake distribution, brake force, handbrake force, drag constants for air and roll resistance, centre of mass, clutch engage time, maximum steering speed, maximum steer angle and so on.

First, the algorithm calculates the drag forces that depend on the velocity, to further apply. Then, it manages the gear box. If the engine's RPM are close to the maximum, then it is time to shift up, on the other hand, if the RPMs are close to the idle, then it is time to shift down for increased power. When the gear shifts, the clutch is applied and the vehicle stops accelerating for as long as the clutch needs to fully engage.

After the gearbox, the algorithm calculates the engine's RPMs based on the motor wheels rotation. It makes an average of the motor wheels RPMs and multiplies it by the ratio of the current gear and by the differential ratio. If these rotations are below the idle RPM value of the engine, then they must be forced to this value, otherwise the vehicle stalls.

When the algorithm knows the RPMs, it calculates the torque for the given RPMs. Remember that the torque values were loaded into lists, so the algorithm interpolates between the RPM and torque values, to get the current torque.

Known the torque, it is time to apply the drag forces that should point backwards the vehicle.

The RPMs should also be below the maximum engine RPM. If they are above this value, then their value has to be forced down. On real life, when this happens the engine struggles to keep pushing, but then, the electronic system comes in and drops them down to avoid any damage. To simulate this, a few RPMs are decreased (around 500) from the actual RPMs.

As the driver presses the accelerator pedal, the valves open accordingly and the engine starts producing power. In Unity, the pedal input is a value between zero and one, therefore the power applied on the wheels is equal to the current torque, times the pedal input. The torque is only applied to the motor wheels.

d. Vehicle Brakes

The vehicle brakes have an important role in stopping the vehicle, in the shortest distance possible. To decrease the stopping time, the brake system has to be well balanced, taking in account the vehicle's mass and the tires characteristics. Nowadays, there are electronic systems that allow to reduce the braking distance on different road conditions, such as ABS. Anti-lock braking system, makes sure the wheels do not lock. When the wheels lock, the vehicle starts skidding, increasing the stopping time and not allowing the vehicle to corner. With this system installed and active, the wheels will never lock, regardless of the road conditions, improving safety, stability and cornering.

A braking system is operated by the brake pedal, which compresses a fluid that circulates inside pipes that are connected to the brakes. Most of the cars have disk brakes, but some of them still have drum brakes in the rear wheels. The disk brakes, have a calliper that when the fluid pressure increases, pushes the pistons inside the callipers against the disk, creating a massive friction. The drum brakes work differently. They have and external part, called drum and inside it there are pedals and a spring. When the fluid pressure increases, the pedals are pushed against the walls of the drum. The disk brakes are better, because the disk is refreshed by the cold air coming from the front when the car moves, on the other hand the drum brakes do not breathe so much, so their temperature increases more. The temperature directly influences the way the materials generate friction. Usually, the better balance of temperature and performance is in the mid-range of the minimum brake temperature and the maximum temperature they can reach. Therefore, when using the brakes too much, they can overheat and lose performance, increasing the braking distance. On the other hand, if the brakes are cold, their efficiency is reduced, but not so much as if they were hot.

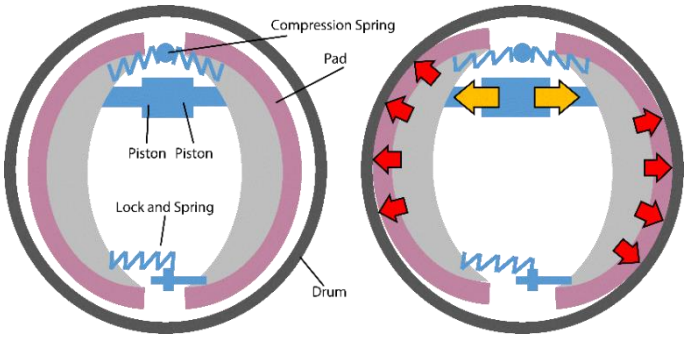

Figure 37 - Drum Brake Parts (Adapted from [102])

No brake pressure on the left, almost full brake pressure on the right

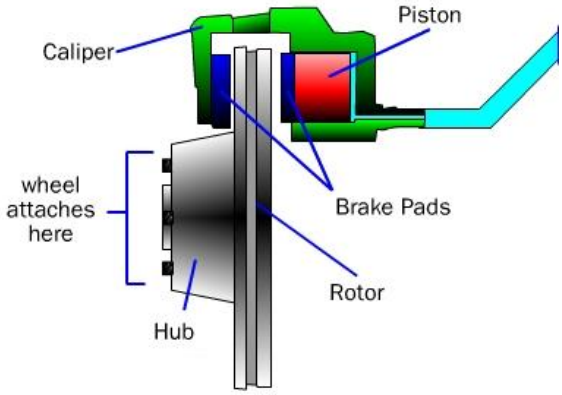

Figure 38 - Disk Brake Parts (Adapted from [103]**)**

Also, to further decrease the braking distance, it is possible to configure the brake distribution. Brake distribution allows to distribute the pressure of the fluid on the front and rear axles. For better efficiency, the brake distribution is usually 60% in the front wheels and 40% on the rear. When the vehicle brake the suspension in the front gets compressed due to the inertia, while the rear suspension gets decompressed, making the centre of mass move forward. To compensate the movement of the centre of mass, the front wheels need to have more braking power sense they are sustaining most of the mass of the vehicle while braking. When the rear wheels brake more than the front wheels, the vehicle may suffer oversteer and spin.

This project does not feature the factor of the braking temperature, but features the brake distribution, which is predefined to 60 in the front and 40 in the rear. The total braking force is defined by multiplying the brake force by the brake pedal input that ranges from zero to one. Then, this force is applied to the front and rear wheels, multiplying by the respective distribution in percentage.

e. Vehicle Steering

Vehicle steering is responsible for steering the steerable wheels on the direction the driver wants to go. To do so, there is a shaft connected to the steering wheel that rotates a gear. This gear is connected to the steerable wheels axel that moves on the opposite direction of the wheel rotation. By moving in the opposite direction, the wheels will turn to the side the steering wheel is pointing.

This system is usually very heavy, requiring a lot of effort from the driver, especially with the vehicle stopped. To overcome this problem, the vehicles come with a systems installed called power steering that can be hydraulic or electric. This system aids the driver by generating the necessary force to rotate the wheels, while the driver rotates the steering wheel.

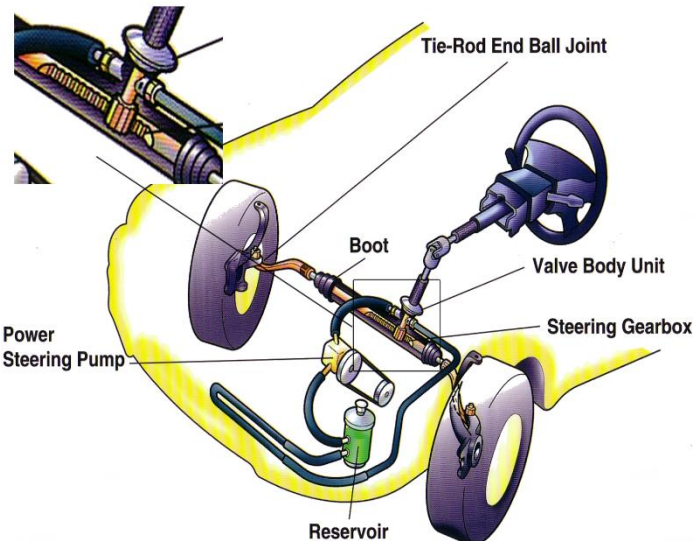

Figure 39 – Hydraulic Power Steering System (Adapted from [104]**)**

Non-power steering is exactly the same, but without the pump

This project has a system called adaptive steering that makes the vehicle steer less the greater the speed. The script has predefined values that define the maximum steer angle with the vehicle stopped, the maximum angle at the top speed and the maximum steer speed. To apply steering to the front wheels, the algorithm simply calculates a percentage of steering, equal to the current speed over the maximum steer speed. Then, the steer angle is calculated by using a function that interpolates the minimum and maximum steer angles with the percentage of steering. After, this value is multiplied by the steering input that ranges from zero to one and is applied to the front wheels.

f. Vehicle Illumination

Vehicle illumination is what makes driving at night possible and ensures road safety by increasing forward visibility and making our own car visible to the others. Some other lights are just warning lights, such as day running presence lights, brake lights and tail lights, others are direction change lights, such as the indicators and the last type is the visibility lights, which are the low and high beams ahead, fog lamps on some vehicles and reverse lights that act as warning lights as well.

In this project, the vehicles have the full visibility light set and braking lights. For now, the indicator lights are not required.

The headlights follow strictly the rules of vehicle illumination, having the correct shape and distance. The low beams cover around 30 meters of road ahead and have an inclination of around 35º on the right side. The high beams cover over 100 meters of road ahead and do not have any

shape restriction. None of the vehicles have intelligent or automatic headlights, such as Matrix LED or Laser.

Figure 40 - Shape that the Headlights Projection should have

Used on the low-headlight (included on the project)

The braking lights are turned on when the vehicle is braking and the tail lights are on when the low beams are on as well.

g. Vehicle Cameras

All vehicles have 7 cameras attached to them and all of them provide a different angle of vision. This cameras are in the cockpit, in the middle of the car's interior, on the roof, on the bonnet, on the left front wheel, on the right front wheel and behind the car, not counting the Oculus Rift.

All the cameras render image effects that allow to enhance the image quality. Since they operate in deferred mode, they can render a larger range of image effects than the cameras in forward mode, as well as supporting HDR.

The vehicles have a script attached that allows the user to change between cameras.

Also, the vehicle has rear and side view mirrors. This mirrors are basically cameras that render their image in a low resolution texture, for better performance that is used on a material. This material is then applied to the game object where we want the mirror to be, in this case, the side and rear view mirrors. Usually, the vehicles have a common mesh for the entire cockpit, meaning that it cannot be split in Unity, only on an external 3D modelling program, therefore applying a material to the rear view mirror will affect the entire cockpit. For better rendering and maintenance of the mirrors, a small polygon was created on the mirror's position that barely overlaps the parts where the reflection should be. The material is then applied to this polygon, to sustain that nothing on the cockpit gets ruined.

h. Oculus Rift Integration

An important aspect of a driving simulator is the capability of immerging the user on the virtual world. This is often achieved by using surround sound, natural looking visuals, appellative details or action and so on.

With Oculus Rift it is a different story. Oculus is capable of immerging the user so well that he actually feels that he is inside the virtual world, it is almost inevitable to attempt to touch things with your own hands and not to react to objects that come towards the camera. Having the Oculus Rift working in this project, inside the cockpit, is a massive contribution to the user immersion and experience.

When the user feels immersed in the virtual world, he forgets about the real world. He will react to the virtual world and get distracted by virtual objects that are present in the scene. In this driving simulator, using Oculus, it is not only a massive improvement to the immersion, but also a way to calculate how much time the user is not paying attention to the road or gets distracted by billboards or other appellative objects.

The Oculus integration is only available on the cockpit view of the vehicles, because it is where the driver is sitting. The Oculus provides a free integration package for Unity 4, which works perfectly on Unity 5. To make Oculus work on the project, the only thing needed was to drag a camera rig to the position of the cockpit camera and disable this last one. When the scene runs, the Oculus game object works like a normal camera. In fact, there are two main cameras, one for each eye, whose images are merged using a special shader. On the main menu, the user can select whether to use Oculus or not. If he does, when the vehicle is instantiated, right after the scene generation, all the cameras in the scene are disabled and the Oculus camera rig game object is created on the position of the cockpit camera. This guaranties that no more than one camera is active at the same time as the Oculus, as well as disabling the camera selection. To better use the Oculus, the graphical quality is automatically reduced increasing the frame rate. On lower end computers, it is recommended to lower the graphical quality manually on the start screen.

The Oculus HMD (head-mounted display) acts like a normal screen that is connected to the computer, therefore it should be used as one. It also has two display modes, the direct and the extended. In direct mode, the computer automatically selects to use the Oculus when an Oculus application runs, but has a slight drop on quality. On the other hand, on extended mode, Oculus acts as an extension of the main screen, meaning that everything that is dragged with the mouse to the next screen, will be displayed in the Oculus. Also, the extended mode enhances the quality of the head movements and tracking.

This project was built and tested using Oculus on extended mode. In order to change the Oculus mode the user needs to go to the Oculus runtime definitions that come with the installer of the service.

When the scene is loading, the project window needs to be changed to the extended screen, so it can be displayed on the HMD, to do so on Windows simply press Win+Shift+Right or Left,

depending on the position of the extended screen, always configurable on the Screen Resolution menu of Windows.

i. Vehicle Damage

The user can choose on the main menu what type of damage the vehicle can have. The options are no damage at all, visual damage only or visual and mechanic damage.

No damage at all, disables any kind of body deformation and mechanical malfunctions.

Visual damage only, enables body deformation, but without any kind of mechanical malfunctions. This kind of damage allows the vehicle to be freely deformed upon impact with no limits.

The mechanical damage, consists in creating malfunctions on the vehicle's mechanical parts. The main parts are the wheels that can get misaligned and ripped off, the steering can get unaligned, the suspension can break down and the engine can lose power. Other components such as gearbox, brake pressure, handbrake or fluids temperature were not considered for simplification.

The visual damage is applied to the vehicle's mesh, upon a collision it is possible to retrieve the collision point and the collision force by means of a collision event and thus it is possible to calculate the deformation range and apply it to the closest vertices of the vehicle mesh.

The mechanical damage is applied to the vehicle's parts. Having the collision points and force, the algorithm gets the closest wheel, decides whether to misalign the wheel or not and decreases the wheels "health" by a certain value until the wheel reaches a "health" equal or below zero, making it ripped off. Also when the impact is big, the suspension of the closest wheel to the contact point, may break its suspension, making the vehicle unstable.

3.5.17 Visual Effects

In Unity 5, it is now possible to have physically based shading, which improves the image a lot, enhancing the natural feeling that all the scenes need. Physically based shading is a type of object shade that allows the materials to interact with light, instead of just using it for themselves calculations. The light that reaches an object can be reflected and refracted according to the material's properties and interact with the multiple objects on the scene spreading colour and brightness. Although, the appearance is much enhanced and the graphical overhead is minimum.

Another important effect is the high dynamic range (HDR). When a scene is rendered in low dynamic range (LDR), the values for the RGB are between 0 and 1. This is the standard rendering mode, but it does not give the most realistic graphics/visuals. In order to improve the visual quality, a new type of rendering was created, HDR, which is capable of supporting values for the RGB out of the zero to one range, now it is possible to have values larger than one. The screen does not support values out of the zero to one scale, so the HDR will be converted to LDR again,

but in the middle, there is a processing step that will scale the HDR values to fit on LDR values. What this does is, to enhance the bright areas, making the darker areas even darker and the brighter even brighter, similar to the human eye. When staring at a light, the remaining field of view gets almost back and it is hard to distinguish anything, but turning of the light the eyes need time to adapt to the environment light (not so bright). Over time, it starts being possible to see the surroundings, along the entire field of view. HDR attempts to simulate this effect and the eye adaptation time can be configured, as well as the value scaling properties, to create different types of visuals.

To work with HDR implies that the cameras use the deferred rendering path, which has different capabilities then the standard mode, such as supporting HDR and any kind of image effects.

Another visual effect is the bloom. To use bloom it is mandatory to use HDR. Bloom processes any RGB values over a certain threshold, usually the values over one and blurs them. The blur only affects values above the threshold, which are the brightest areas, making them spread to the surrounding pixels, giving a very shiny look. This effect allows to create so shiny sun reflection that limit the user visibility. For instance, when the sun is down at the horizon, when the user is looking at it he can only see an extremely bright light and very little of the road ahead, like in real life.

Another effect is the lens dirtiness. This effect simulates lens dirtiness when the light entering the camera reflects and refracts on the dirty. It creates a beautiful effect of bright lights and reflections.

The other effect used is the Vigneting and Chromatic Aberration. They darken and blur the corners of the screen on a circular shaped vignette. Acting so, creates the feeling that the camera is focused on the centre of the screen, because the corners get blurred decreasing their relevance to the user.

The last image effect is the fundamental processing needed for the HDR, which is the tone mapping. This effect is responsible for scaling HDR values back to LDR values and creating the eye adaptation effect as well.

Any of these effects is a post-process, meaning that all the effects are applied only after the entire scene is rendered at each frame, in image space. Also, each one of this effects takes a processing time, slightly dropping the frame rate. For lower end computers, this effects can be a real pain on the frame rate, so it is possible to disable them in start menu by reducing the graphical quality. On higher end computers the frame rate drop is almost not noticed.

3.5.18 Procedural Scene Export

One of the goals of this project, is to facilitate users the ability to configure serious driving game scenes. In order to do it, it allows to export the scene to a 3D file format, in this case ".obj". The script used comes from the Unity Community Wiki, but it only allowed to export objects on the Unity editor, because it uses methods, to retrieve the materials and the textures of the objects, that belong to the UnityEditor namespace, which is never compiled when exporting the project to an executable. This does not mean it throws an error when compiling, the compiler simply ignores anything that belongs to the UnityEditor namespace. Therefore, a few modification had to be done. The first one was to figure out a way to retrieve the materials and textures without using the UnityEditor namespace. After a few research, there was no other method found, even on Unity forums there were people claiming that it was impossible. In fact, doing things like this is, but to overcome this problem I found a strange solution. First of all, all the textures had to be moved to a folder called "TextureLibrary". Then, instead of the algorithm using the UnityEditor methods, these lines of code were replaced by methods that search on the "TextureLibrary" folder for the desired textures. This fixed the problem, but on the other hand all the textures are visible and on the same folder, as well as they have to follow along the executable, so the user must have maximum caution with this folder. Another change was changing the name of the export function to "ExportScene" and make it static. This function receives an array of Transforms that are the children of the scene game object.

In order to export the entire scene, all the objects that belong to it have to be placed as child of the entire scene game object.

Now, to export the scene, the user needs to bring the pause menu by pressing the Escape button. Then the user clicks "Export Scene" button that, first of all, finds the game object called "scene", which is the entire scene parent and then calls the "ExportScene" method from the export class called "MyObjectExporter", because this method is static. This method will receive all the children (direct and indirect) of the "scene" game object.

The scene usually takes a long time to export, depending on its size and complexity and usually generates very large files. This files go to the "ExportedObj" folder, along with all the textures used and can be imported by any 3D modelling application.

3.6 Summary

As seen in this chapter, this is an ambitious project with lots of modules and interactions to implement. Unfortunately, due to personal setbacks it was impossible to work for a time period of around a month, so the work was stopped and very delayed, but as soon as possible it was resumed start working pedal to the metal and everything proposed is now complete.

The implementation is fully detailed, in order to help any one that wants to continue extending and enhancing this project by providing extremely detailed information on how the modules interact and how the algorithms work.

Thanks to the first semester it was possible to start making prototypes and exploring possibilities that leaded to the abandonment of the SUMO integration, because it is impractical on this context. Therefore, regardless of any delays it was much easier to develop the rest of the project by following regular and fixed work hours and methods. It was not an easy task, there were moments of victory, failure, happiness, frustration, sadness and so on, but it was a journey where a lot was learned, ties where created and develop and the human factor was enriched.

Chapter 4

Results

This chapter describes all the information related to the results of the implementation section in Chapter 3. It is divided in multiple sections, each one corresponding to the respective section of the implementation. All the imagery was gathered from the final version of the project and represent the actual quality of the final product. Nevertheless, the images were gathered on the Unity Editor, therefore they have reduced graphical quality compared to the exported executable.

Although this was a very ambitious project, all the proposed goals were completed and the results are very much promising. They might not be perfect, but they do the job and establish the beginning of an important tool that can be further enhanced and expanded as mentioned before. The images only show a few possible scenarios, but there is a lot more content to see than what these images suggest.

4.1 Main Menu

Figure 41 - Main Menu

The main menu is simple and easy to use as seen on the picture above. It provides access to the options, the vehicle selection screen and the start button that only works if the input box is filled with a valid world location by address or coordinates.

4.2 Main Menu Possible Configurations

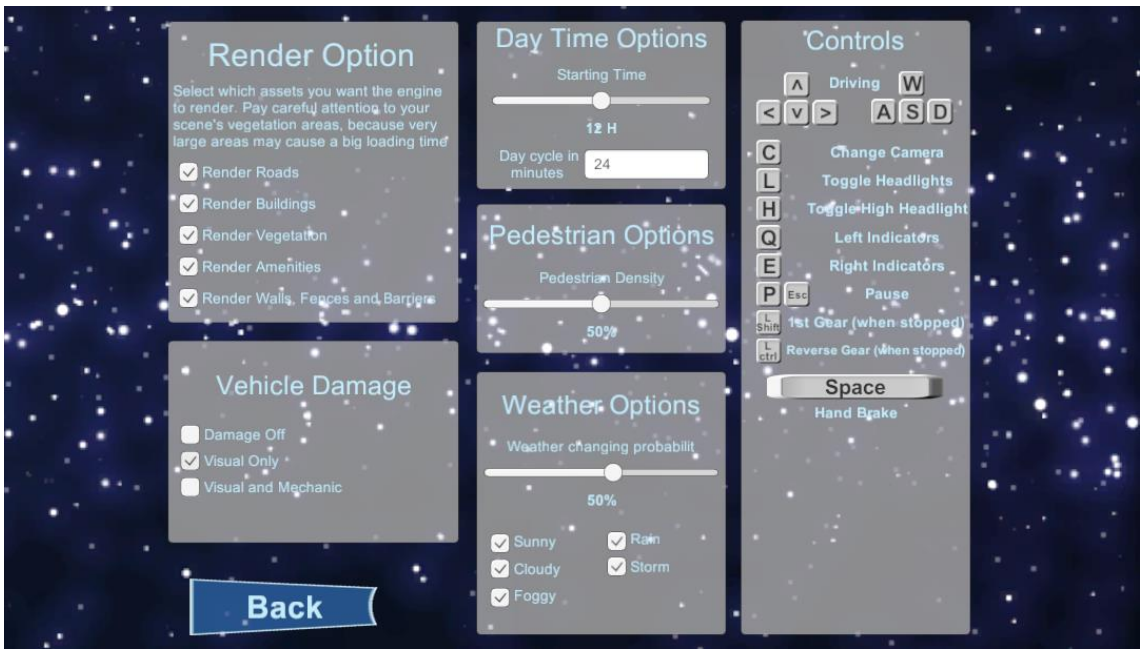

Figure 42 - Advanced Options Screen

This screen is the advanced options menu, accessible on the main menu. It provides a decent amount of possible configurations on a stylish layout. It allows the user to select the render options, the type of vehicle damage, the weather conditions that can occur, the weather changing probability, the pedestrian density, the time of day and its duration and on the right all the controls are displayed. The controls can be changed before the application start, although any alteration will not affect the current controls displayed.

4.3 UnitySlippyMap

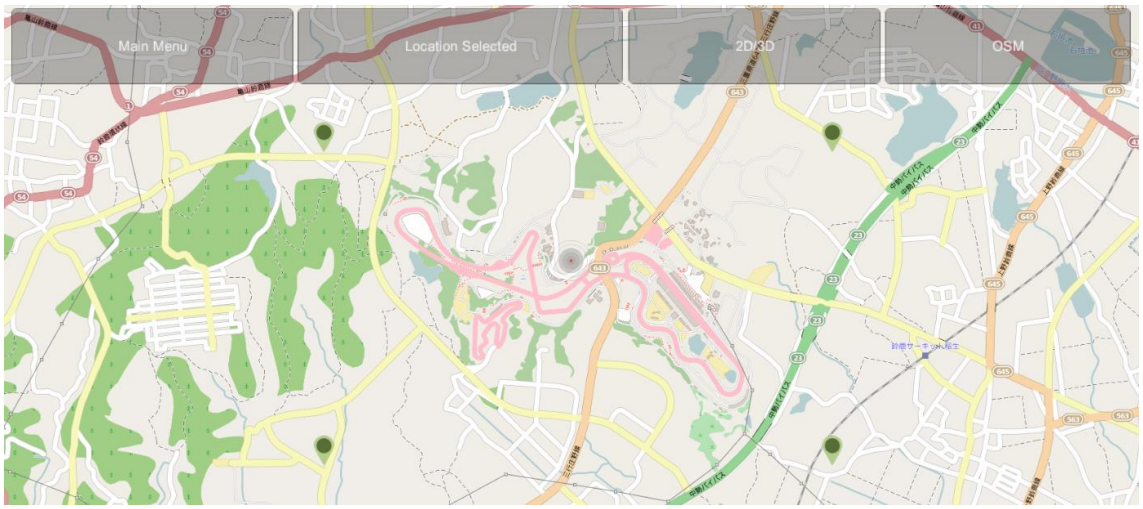

Figure 43 – UnitySlippyMap

Four points define de bounding box around Suzuka Race Tack

This interface is a completely new scene that display on location typed on the input box of the main menu. This interface allows the user to navigate the world map by zooming in or out and dragging the map with the left button of the mouse and select a bounding box with the right button of the mouse. To select a location the user simply uses the target in the middle as a reference to place the initial marker by right clicking. The second right click places the second marker and two more markers that will close the bounding box. The first and the second markers represent the diagonal of the box. If the user is not satisfied the next right click will place a new first maker and overwrite the previous selection. On the top of the image there are four buttons. The first gets the user back to the main menu, the second selects the location inside the bounding box and advances to the next step, the 2D/3D button allow to switch between the top-down view and the 45º view and the last button changes between different tile providers.

Although the interface is simple the bounding box selection may not be very intuitive. But this is because all the code was adapted from an initial code without any documentation, making

it hard to understand the flow of the scripts and how to integrate it on the context of this project. Nevertheless, the integration made, is solid and functional, providing the possibility to select any world location.

4.4 Elevation Data

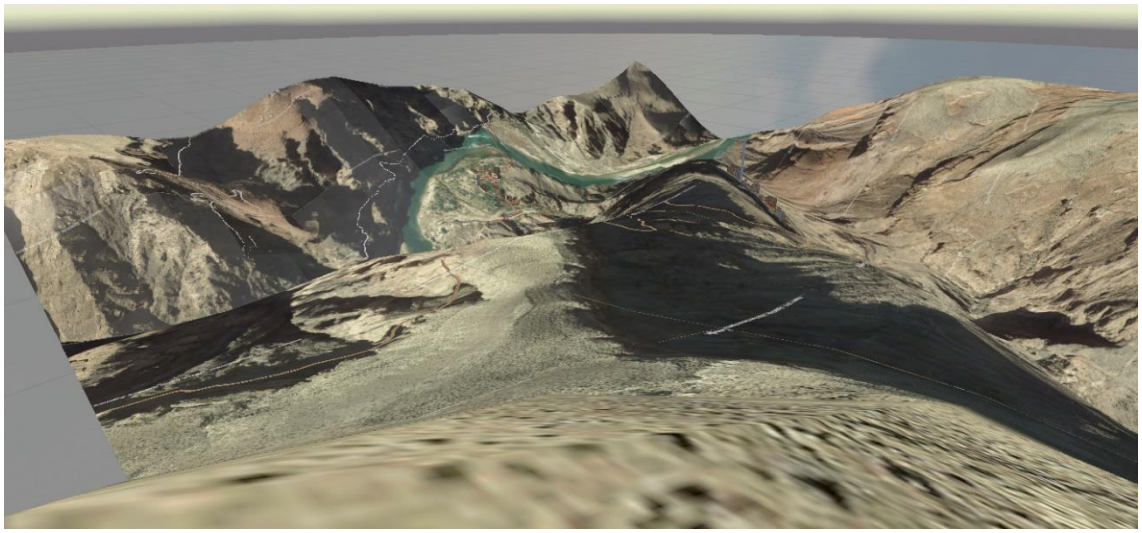

Figure 44 - Elevation Data
Small portion of the Grand Canyon, USA

The elevation data used has a 90 meter resolution on both longitude and latitude, but this has a disadvantage, because the elevation values belong to an exact point of the Earth. Also, the Earth is not a perfect sphere, it is a spheroid, which means that the distance covered by 5º will not be the same across the entire surface. This means that the elevation data and the real world data have different scales, therefore the real elevation is not apart 90 meters across the entire surface of the Earth. This introduces an offset, causing the elevation to be misaligned with the real elevation. As you can see on the image above, the river starts climbing the mountain. Obviously, this offset is noticeable, because rivers always go down, otherwise they make their way through. This couldn't be corrected, because there was no time to, but in certain locations the elevation offset is not noticeable, rather because it is too small or there is not a great elevation variance.

Regardless of the offset problem, the elevation model looks fairly realistic and approximate to the real world.

4.5 Location Data Parsing and Satellite Image Gathering

Figure 45 - Bird Eye View of the Suzuka Race Track, Japan

The location data parser is responsible for organizing all the information so it can further be procedurally generated.

By the image above, it is possible to see all the generated models such as roads, footways (red), paths, vegetation and the walls of the buildings between others.

The satellite imagery is gathered from Google Maps, as it contains the Google mark on the bottom right corner. In most of the cases the imagery and the generated meshes are well aligned. On other cases there may be an error of around 5 meters, which is not major.

4.6 Terrain Elevation

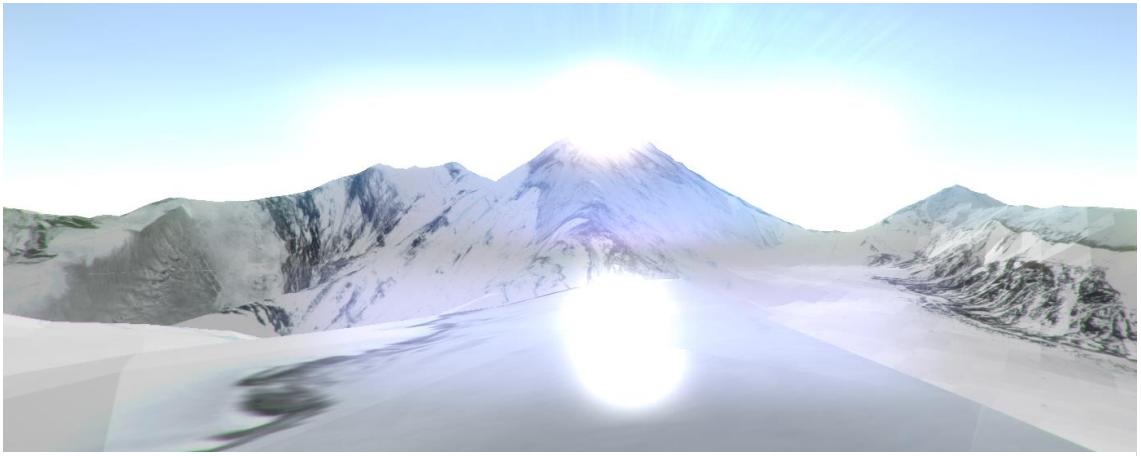

Figure 46 - Highest peak of the Mount Everest, Himalayas

The terrain elevation suffers from the problem of the differences of scale between the location data and the elevation data, causing a more or less small offset between the position of the elevation points and the real world. Despite this problem, the elevation system can generate realistic results especially having all the satellite imagery aligned.

4.7 Procedural Roads

Figure 47 - Procedural Roads with Street Lights

The procedural road produce a very smooth and satisfying result. They perfectly adapt to the ground, always following the path and not producing any kind of sharp corners that are not smooth or seamless. Also, they can preserve their width along the entire path. The sidewalks and the sidewalk steps go along with the road, always keeping the correct height, width and shape.

The only problem with the roads is the fact that different roads overlap. What this means is that on the intersections, two or more roads take the same place and the sidewalks go into the middle of the other roads. This project does not support intersection handling, for now.

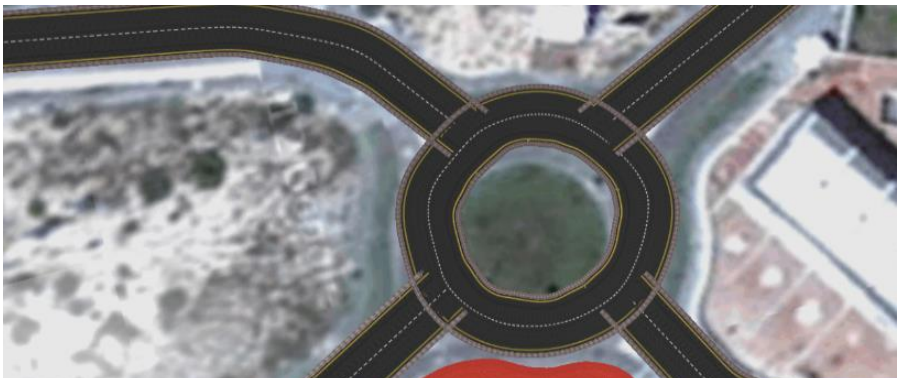

Figure 48 - Procedural Roads Overlap Problem

4.7.1 Road Enlightenment

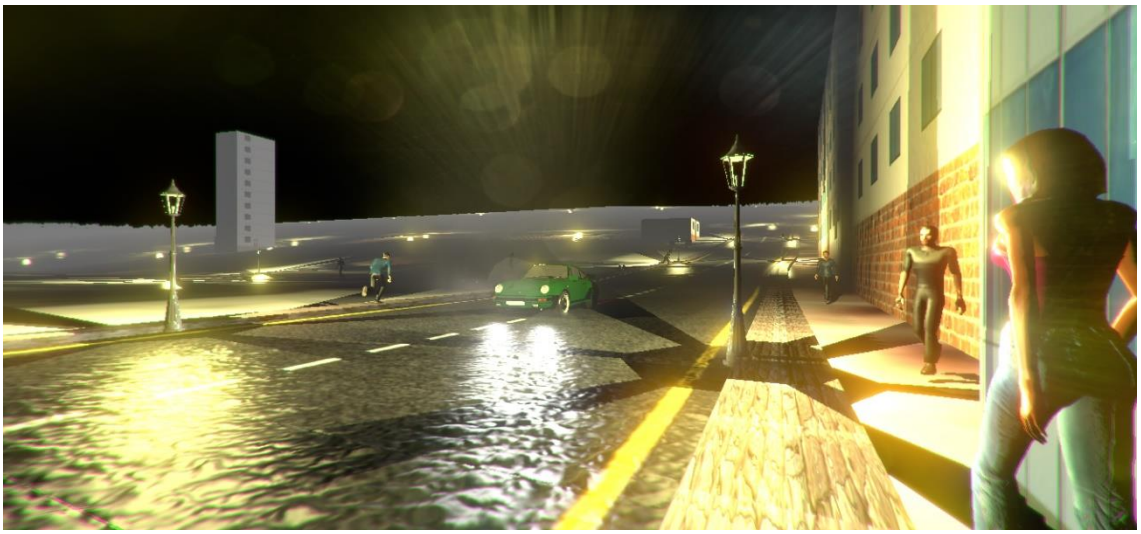

Figure 49 - Procedural Roads Night Illumination

As expected, all the street illumination is placed along the enlightened road, at a fixed distance on each side, providing enough light to see the pedestrians and the road itself. Without the illumination the user could only rely on the vehicle's lights, which may not be enough for an environment where pedestrians are walking. They turn on and off when needed, in this case when it gets darker or brighter respectively. Regarding performance, having dozens of lights casting dynamic shadows at the same time causes the frame rate to drop. To counterbalance this fact, it is possible to remove the shadow casting on the initial screen before starting the project by reducing the graphical quality.

4.8 Pedestrians

Figure 50 - Pedestrians Walking and Getting Hit By Vehicle

Pedestrians populate the procedural scene making it harder to drive around without making any casualties. As expected, the pedestrians are capable of walking around on the sidewalks and on the footways and are physically interactive with the vehicle by applying rag-doll physics. Also, they attempt to avoid obstacles in front of them, including each other. When they are off a sidewalk or a footway, they try to run to the safety of the closest sidewalk or footway. In addition, when they cross over another walkable surface, they decide whether to change their path or not. After implementing the LOD controller, there was a massive improvement on the performance of the application. By using a LOD controller, the pedestrians disappear according to the distance between them and the main camera. By just showing a few pedestrians instead of showing the further away, the frame rate gets a very big improvement.

When the pedestrians get hit, they never damage the vehicle, regardless of the force of impact. This is because they are careless, although they analyse the path in front of them, they have no notion of what surrounds them, so they do not react or predict the vehicle's movement in order to avoid it. Although the AI is simple and functional, it still needs a lot more work in order to achieve a much more complex and smart intelligence.

4.9 Procedural Buildings

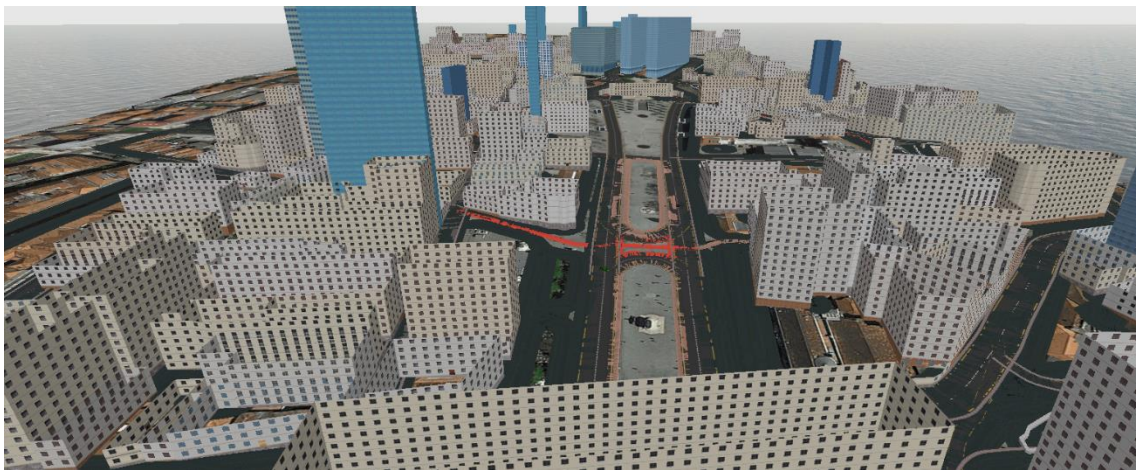

Figure 51 - Procedural Buildings, Reconstructing Avenida dos Aliados, Porto, Portugal

As mentioned before, buildings have no rooftops. This is a driving simulator, everything happens on ground level, so, to save resources, there is no need to implement the rooftops. This picture was taken on the editor, with a free camera and, therefore, it can be moved anywhere. To show the potential of the building generation procedure, the picture was taken from a bird eye view. By analysing the picture, the results seem very good, each building is well suited to the ground with a basement, multiple floors, a large number of windows and a different material selected depending on its perimeter. It is also possible to see different kinds of buildings, the regular ones and the skyscrapers. Skyscrapers are buildings over 15 floors and do not have window meshes, instead they have a texture of glassy windows.

Unfortunately there is lack of information to generate accurate buildings. The file taken from OpenStreetMap barely contains information regarding the existence of buildings or their characteristics. This is why it is possible to see buildings on the satellite images that were not generated.

4.10 Procedural Barriers

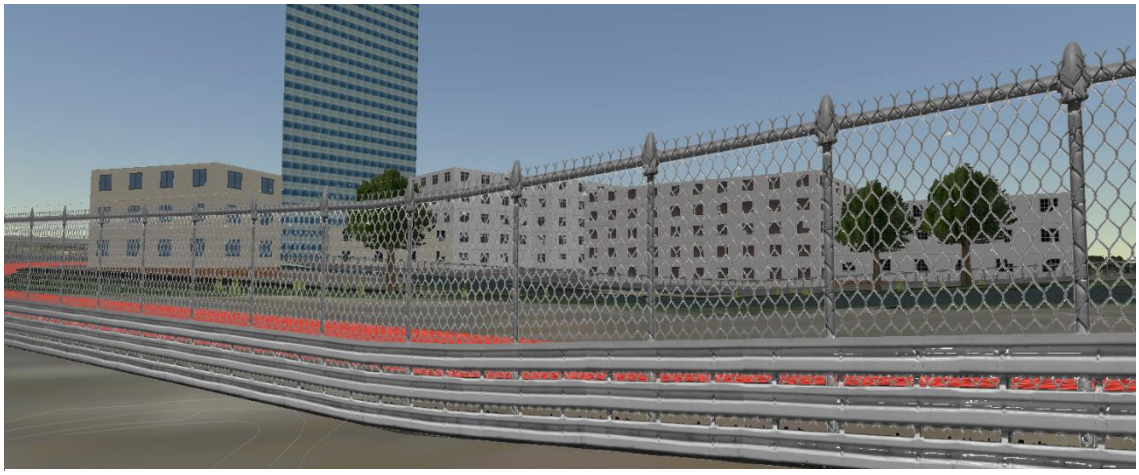

Figure 52 - Procedural Barriers - Guard Rails

The image above, shows a guard rail of a race track with a fence on top of it. These kind of meshes adapt to the ground elevation just like the road, because they are built using the same principle. The other variant are the walls that instead of being a structure with no thickness, they have two walls closed by a top cover.

4.11 Procedural Land Uses

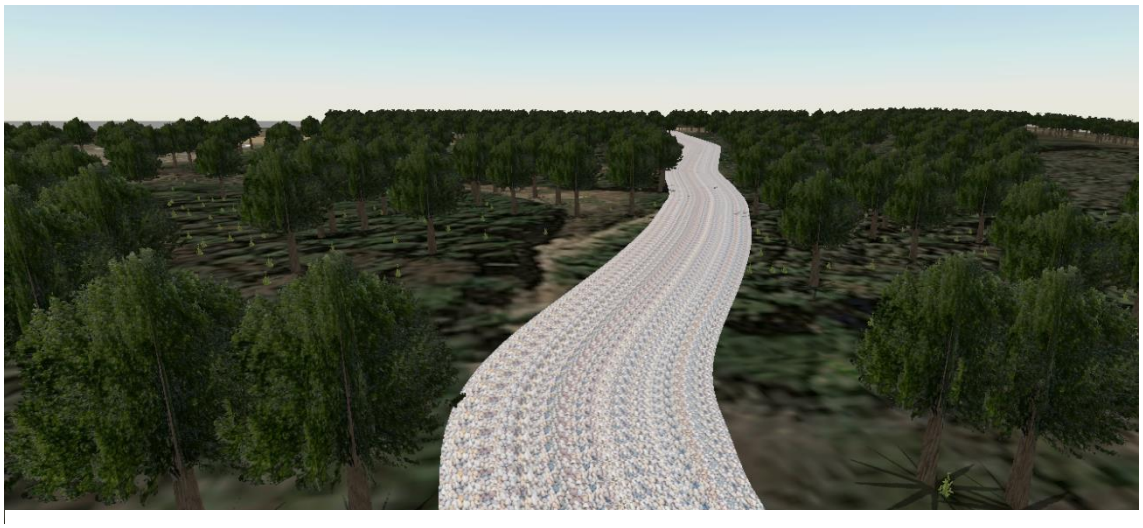

Figure 53 - Procedural Vegetation - Gravel Path Going Through a Dense Forest

As expected, the vegetation covers the respective areas by placing very low polygon trees and bushes. Each tree is an individual mesh that casts a shadow and has a transparent material to simulate the leaves. Notice that there is no vegetation over the gravel path, they are all placed

around it. This is because the road is generated first and then, the trees and the bushes are placed one by one, to avoid overlapping any road. By doing so, it is possible to create beautiful scenarios, like the one in the picture above. All the trees have a collision mesh, while the bushes do not test for any collision.

4.12 Procedural Amenities or City Furniture

Figure 54 - Trash Bin Next to a Street

As explained on the implementation section, the only amenities considered were the trash cans, because there are hundreds of different objects and the trash bins are frequent and are a simple 3D model. The image above show a trash bin placed on the scene. They are not placed randomly, but instead on the nodes identified as so, on the .osm file. Also, they behave has a physical body with collision simulation and weight.

4.13 Day and Night Cycle

Along this chapter, you can find that the images have different lighting conditions. This is because of the dynamic time of day. As expected, the project can simulate a full day-night cycle, whose duration and starting hour can be configured by the user, on the main menu. Also, the time of day influences the global illumination, thus the visibility conditions and the state of the road lights (on or off).

4.14 Weather Cycle

Figure 55 - Heavy Rain and Storm

One of the goals was to create a dynamic weather cycle. As shown in the image above, it is heavily raining decreasing the visibility and adherence conditions of the road. On the main menu, the user can select which conditions he wants to happen and the weather changing probability. The duration of each weather condition depends on the changing probability and on the duration of the day. The bigger the day's duration the longer the transitions between different conditions. Any condition can have the addition of fog to the mixture creating an even harder driving environment.

The current weather conditions available are sunny, cloudy, rainy, stormy and foggy.

Also, when it is raining the user can hear the rain falling and when a lightning strikes he can hear the thunder sound and an intense bright light. This increases the user immersion on the experience by capturing his attention to the smaller details created by the sound.

4.15 Vehicle

At the moment, the project features two extremely different vehicles. One is a sports car and the other a 4x4 off-road pickup truck. This section is divided in subsection to better separate the project's features regarding the vehicles.

4.15.1 Vehicle Selection

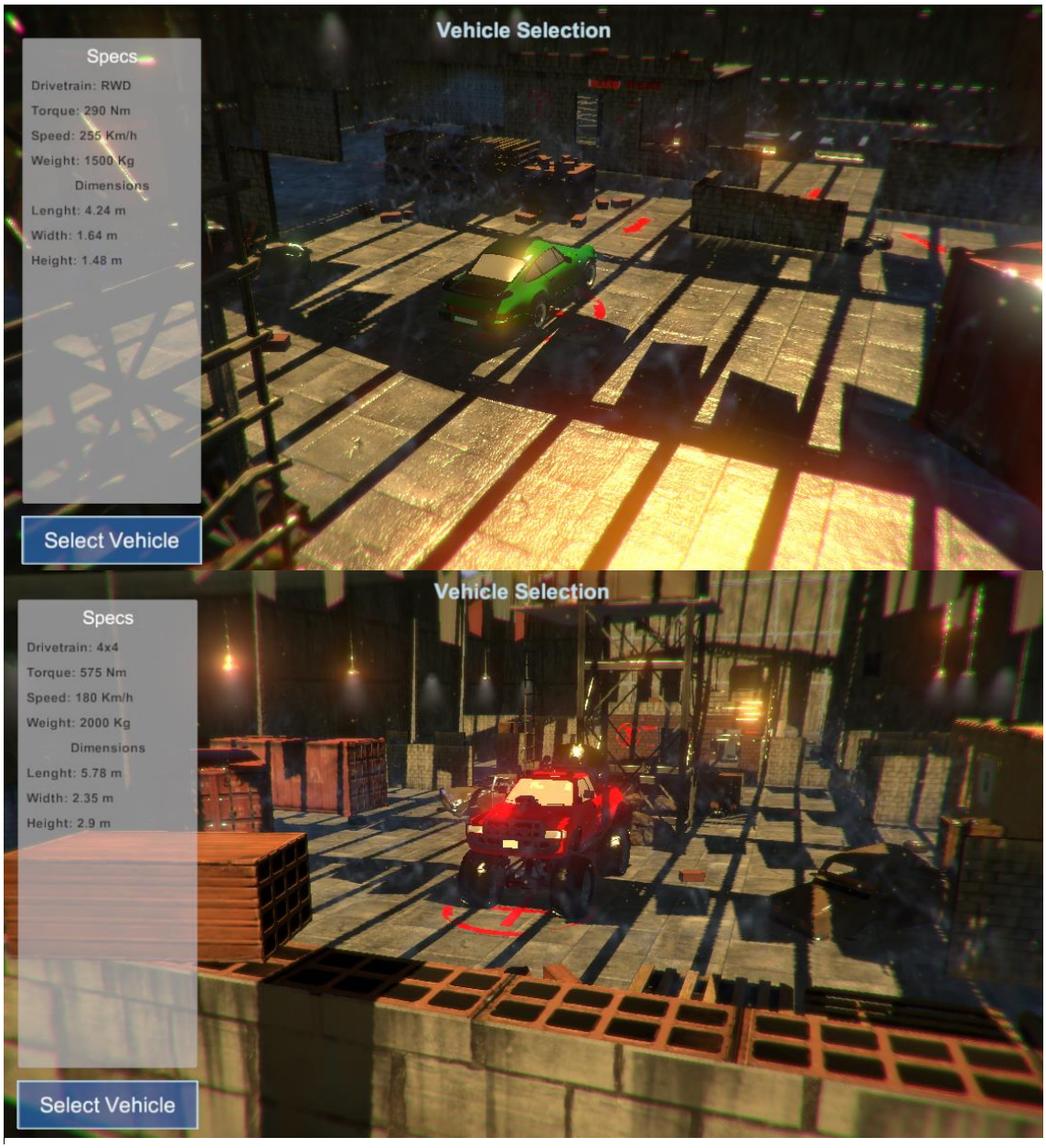

Figure 56 - Vehicle Selection Screen

The vehicle selection screen allows the user to select the vehicle he wants to use. It is accessed on the main menu.

On the left side, the user can see the vehicle's characteristics and find a button to select the currently visible vehicle. This is a simple scene, where the environment is extremely detailed. Also, this scene allows the user to change the vehicle using the navigation arrows, zoom in and out the camera and rotate it around by dragging the mouse.

4.15.2 Vehicle Dynamics

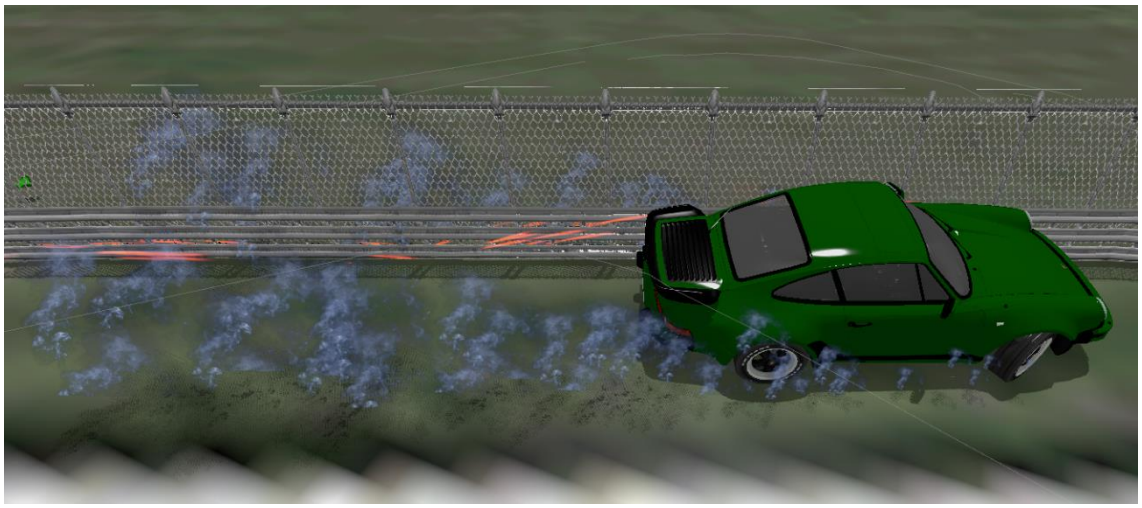

Figure 57 - Vehicle Rubbing a Guard Rail

One of the most important parts of a driving simulator is the vehicle dynamics, the way they drive and feel. This project attempts to simulate the closest as possible the physical behaviour of a real vehicle. Of course, this is an extremely complex topic and Unity is not the most advanced physics engine there is, so there is always room for improvement on this topic.

There are plenty cameras to choose from, to guarantee that every user finds a comfortable view of the road.

When the vehicles skid the tires release smoke particles.

The vehicles are very easy to drive, their suspension is configured to give them stability at higher speeds, the steering is speed adaptive, the faster the vehicle goes the less the wheels turn, the engine's torque is interpolated from a torque curve with realistic values for every vehicle, the braking system takes in account the brake distribution for the front and the rear brakes to guarantee a more stable braking.

In short, it is hard to describe the vehicle's behaviour, it's easier to simply test it instead, because it is a matter of opinion.

4.15.3 Vehicle Illumination

Figure 58 - Vehicle Headlights

High beams on the left and low beams on the right

The vehicles feature the full set of traffic lights, which are the low and high beams, rear night lights, brake lights and indicators. The most important on this context are the low beams, because they are regularly used at night and are not supposed to flare the other drivers. The low beams follow the international headlight shape that can be found on the implementation section and have a reach of 30 meters. So, on the image above, on the right we see the low beams of the vehicle and on the left the high beams that reach up to 100 meters. When driving at night, if the user is in a location with no street lights he really needs to turn the headlight on, because the night is really pitch black as shown on the image above.

4.15.4 Oculus Rift Integration

The integration with Oculus Rift, is the most important part on what comes in immerging the user into the experience. The results of this integration are very impressive, the user really feels inside the vehicle. Thanks to the head tracking, the user can rotate his head and move side to side, backward and forward and the camera inside the application goes along that movement creating the sensation of freedom to move inside the vehicle.

Oculus Rift integration package keeps in constant change along Unity, so updates will be required and a few of them change the Oculus API, forcing the developer to modify the project.

Regardless of the updates, the Oculus provides a system that allows the possibility to be extended in ways that can be used to control and measure the user's movements and the time he is not paying attention to the road, for instance, and thus collecting valuable metrics that may contribute to road safety and driver awareness.

Unfortunately, by the time this section was written there was no access to the Oculus equipment, therefore it was not possible to take a screen shot of them working.

4.15.5 Vehicle Damage

Figure 59 - Vehicle Damage System

Mechanical Damage on the Left and Visual Damage Only on the Right

The image on the left, is an example of the mechanical damage, the deformation is the same as on the visual damage, but the wheels flew off not allowing the vehicle to move anymore. When the vehicle hits something, collision debris come out on the colour of the vehicle and a collision sound is produced. There are 5 different collision sounds, and the sound that plays corresponds to the respective collision force. The debris amount depends on the force of the collision as well. Also, when the vehicle scraps objects, sparks are produced on the collision point and their emission rate depends on the velocity of the vehicle.

The damage looks extremely good, because the vehicle has no deformation cap, meaning that it can fully deform without any limitation.

4.16 Visual Effects

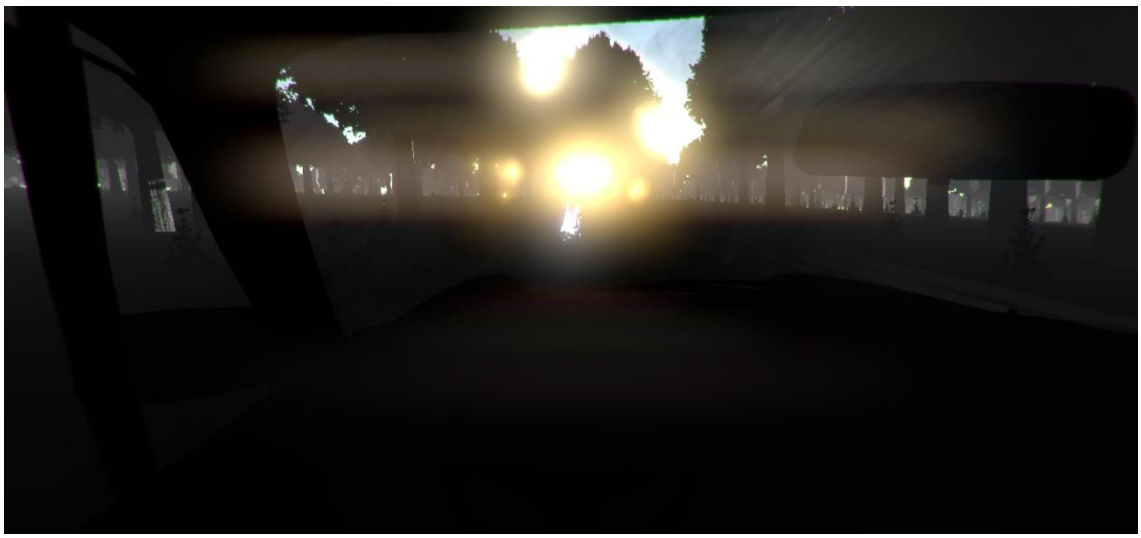

Figure 60 – Beautiful Effect of the Sun Setting Behind the Trees Using Cockpit View

Visual effects massively contribute to the user immersion on the experience, but they come with a cost, the drop on the frame rate. Nevertheless, Unity includes some extremely optimized visual effects that work as a post process effect applying their calculations to the final image gathered by the camera.

As seen along this chapter, on the pictures there are multiple visual effects taking place. For instance, on the image above the sun is setting behind the trees, creating a very bright bloom that spreads across the dirty lens of the camera. As the sun light is right in front of the user's eyes he can barely see the road ahead, just like in real live he gets flared. When running with the high performance graphics card, the drop on the frame rate is almost unnoticeable, but with the integrated graphics card the visual effects drop the frame rate a bit. When decreasing the visual quality down, these effects turn off granting a much smoother experience, especially with Oculus Rift where the frame rate needs to be high to avoid any motion sickness.

The visual effects used in this project, make it look like a high end product with AAA graphical quality.

4.17 Procedural Scene Export

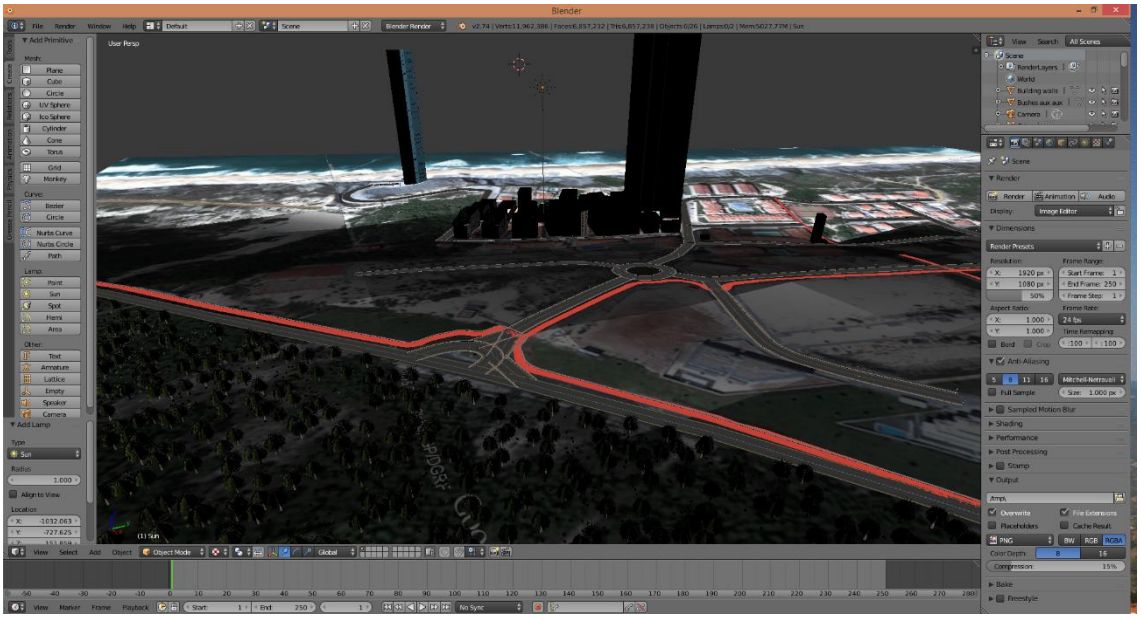

Figure 61 - Exported Scene Edition in Blender

To export a scene, the user simply has to press Escape button to pause the application and click on export button. The exported scene goes under “ExportedObj” on the project folder.

The resultant file of the export operation is a .obj 3D file that can be opened and edited by an external application, such as blender seen on the image above. Then, the artist can modify any structure on the scene, since they are grouped on a hierarchy and separated by their types and from each other, any existent object is accessible independently from the others.

The 3D model can then be exported to a different file type, or not, and be used on any other application for any kind of use.

The .obj file exported, is usually a very big file for a small scene (around 1GB), and even larger for a bigger scene, obviously. The problem is that the application takes a huge amount of time to export the entire scene and the user may think that the application is blocked. This type of file format is usually very large for even simple meshes. The application could export the scene to a different smaller extension, but the .obj extension is much easier and faster to implement, due to the time available to develop all the features of this project. Still, the application never blocked so far on the export process, so if the user is patient the scene is successfully exported and can then be used as desired.

4.18 Summary

From this chapter, we can conclude that the results look very appellative and adequate, especially taking in account the time available to develop all the features.

All the features of the project are solid and work well with no major bugs or glitches, meaning that the application is functional. About practicality, there are parts that could be more accessible to the user in terms of ease of use, such as the UnitySlippyMap bounding box selection and the export process that makes the user think that the program blocked while exporting.

Although it is solid and functional it is not perfect, there is always room for further improvement and expansion, but that is also a goal of this project, distribute freely the source code so any one can expand and modify the application their own way, to suit their needs.

The next chapter describes the possible future work and enhancements.

In conclusion, this project creates the beginning of a new open source tool that is capable of reconstructing real world locations from anywhere, populating them with pedestrians, allowing the user to drive a vehicle, simulating different day and night conditions and weather cycles and allowing to export the entire scene to be externally edited by another program.

Chapter 5

Conclusion and Future Work

Procedural modelling is a very easy concept, with understandable and easy to implement algorithms. The hard part is the vertex manipulation, because the magic of generating a beautiful mesh is not the algorithm itself, but the way it manipulates vertices. To create such meshes, the vertices need to be merged and used by each other, making this a very hard procedure to implement. This project attempts to create the better meshes possible taking in account the time available to develop all the features. The procedural modelling algorithms can still be extensively expanded and enhanced. Nevertheless, the results are very good and convincing, generating an appellative and realistic environment, populated with pedestrians, which can be exported and edited by an external application.

Having so, it is possible to configure many parameters that will allow to create different driving conditions, to conduct different kinds of studies. Also, if the user is not satisfied, the project can be easily extended to suit any kind of needs.

5.1 Goal Satisfaction

All the main goals of this project are fully satisfied, with the exception of the traffic simulator that was abandoned, because it is impractical to integrate an application that runs in real time with an average of 30 fps with an application that does not have a continuous time step, through a TCP channel. The overhead would be so large that the application would be unusable. On the other hand, the road system allows the implementation of a traffic simulator within Unity, without recurring to external applications. Although, this was not done, because it would take too much time. Only the vehicle AI would take around 2 months to be completed, in order to be fully

functional and intractable with the physics engine. This project has too much content to care of that there was no time at all to create an internal traffic system. On the other hand, it includes a pedestrian system that makes pedestrians move around on the sidewalks and footways, avoiding objects and each other and allows them to interact with the vehicle, using ragdoll physics. What this means is that when the vehicle hits them, they are thrown away like a puppet. Every other proposed feature is fully implemented and functional. There is the procedurally generated terrain with satellite images as a texture, the satellite images are downloaded in runtime, there are procedural roads with sidewalks that adapt to the terrain elevation, the roads have different materials, different types and characteristics such as width, surface, the existence of sidewalks and street illumination, the procedural buildings have varying textures, heights, windows and a basement that adapts to the terrain, the vegetation spreads around the entire scene, a full day and night cycle simulates different light conditions, street illumination that turns on or off depending on the time of day, the full weather cycle that simulates sun, clouds, rain, fog and storm with lightning effects that cast a very bright light and noise, two different vehicles that can be selected on the selection screen with different torque curves, dimensions, suspension setting and drivetrains, the vehicle selection screen is a well detailed warehouse, full vehicle damage that can be switched off, visual only or mechanical where the wheels fly off and get misaligned or the suspension breaks, vehicle collision particles including sparks and flying debris, vehicle collision sounds depending on the force of the impact, there is also a map on the main menu where the user can select the location he wants to download, the elevations data is downloaded in runtime, if not already existent in the database, the project offers the possibility to export the generated scene (by pressing Escape for the in-game pause menu) to .OBJ 3D file, to modify the scene on an external 3D application, there are also a few amenities such as garbage bins around the map, procedural guard rails, walls and fences that enrich the scene, there is a water simulation system that simulates the waves on the ocean and provides the best looking water as possible, there are a good amount of image effects applied to every camera such as bloom, vigneting, chromatic aberration, lens dirtiness, HDR camera rendering and linear colour space, there is the Oculus Rift integration that allows the user to drive the vehicles using the cockpit camera with the Oculus Rift on extended mode and finally there is an options menu where the user can configure parameters such as the pedestrian density, the weather changing probability, the weather conditions that can occur, the rendering options that define what objects are going to be procedurally generated, the vehicle damage type and the day and night cycle duration and starting time.

This is a very complex project that attempts to generate serious driving games scenes and possibilities of configurations, so the user can simulate any kind of conditions and situations possible. It marks the beginning of a new tool to quickly generate scenarios of the real world that can be exported to be used in other projects or applications and it can be enhanced and further expanded to fulfil the desired needs. All the features are fully implemented and functional, providing a complete ecosystem of world location generating tools. By means of procedural

modelling, the process of generating an entire scene is made fast, simple and free, because everything is available to anyone who wants to expand or enhance the project.

5.2 Future Work

Although all the goals have been implemented, it is possible to improve some of them and implement new ones.

Regarding the procedural elevation tiles, maybe not now, but in the future, the source of the elevation data can be changed to a source that offers better data resolution. The same applies to the satellite imagery, for now the zoom level used is the maximum possible, but in the future eventually Google will have HD imagery.

Another thing about the elevation data, is that the elevation data provided from CGIAR-CSI follows a square grid that is constant around the entire earth. On the other hand, the earth is an ellipsoid and both OpenStreetMap and Google Maps use an ellipsoid model. Therefore, there is an offset between the elevation presented on the elevation tiles and the real elevation. The solution is to somehow find the relation between the different sources and apply a negative offset to compensate the differences.

Procedural roads were a complicated task, due to the fact that they have to follow a predefined path, so they have to be built along them. They still have a problem, which is road overlap. When two or more roads intersect with each other, they overlap, making the sidewalks go over the intersection. It is possible to avoid this, but it would require a lot of effort and time.

Procedural buildings do not have neither a roof nor interior. Since this is a driving simulator, everything happens on ground, but it can still be expanded to a flying simulator and then, the rooftops will be needed. But once again, for the purpose of this project the rooftops are not needed, although they can easily be implemented. Regarding the interiors, it comes to the same basis, the interiors are not required, but let's say other developer needs to expand the project for firefighting simulation. Then the doors need to open, there needs to be interior decoration, stairs and functional windows.

Unfortunately, OpenStreetMap does not have information of all the buildings of a location, only the most relevant. So, an interesting improvement could be to attempt to extract the rooftops from the satellite imagery to then render the missing buildings.

The vegetation itself is very simple, there are just 2 different models that are instantiated. It was done like this to save the maximum computer resources as possible. Nevertheless, the project is configured to work with an unlimited number of different 3D models for the vegetation, this function is only deactivated for the resource saving. Although, it can still be activated easily. There are shaders and other workarounds that can do this, but more time was needed to fully investigate.

The number of different amenities can also be increased by finding different 3D models for the different OSM tags that constitute the amenities and somehow group those when needed.

The pedestrians are not very smart, although they run from the road to the sidewalk, change paths, avoid each other's and the obstacles and follow along the sidewalks, they do not have much awareness of the world around, they only see in front of them, meaning that they do not attempt to run away from the car. So, their AI still needs a few weeks of work, but it is not bad at all.

The project features no traffic system. One major improvement would be to take advantage of the well-known vertices of the road and create a traffic system. This may seem easy, if we want a simple block that represents the vehicle to follow a path, but when the vehicles have wheels and follow a full physics simulation, the work is much harder, because the input for the brakes and gas pedal needs to be controlled based on the surrounding environment, as well as the steering input. Although it is not impossible, only very complex, so, maybe more than a month is needed.

The list of future work above, is just a small example of maybe the most important features, but this kind of project has the possibility to be expanded in millions of ways, because, not only it is supposed to be a tool to help conduct experiments, but also because it was built on a powerful game engine.

References

[1] J. Gregory, *Game engine architecture*. 2009.

[2] I. Millington, "Game Physics Engine Development," *J. Vis. Commun. Image Represent.*, vol. 18, p. 481, 2007.

[3] GameWorks, "GameWorks PhysX Overview," 2014. [Online]. Available: https://developer.nvidia.com//gameworks-physx-overview. [Accessed: 26-Nov-2014].

[4] Havok, "Product Overview | Havok," 2014. [Online]. Available: http://www.havok.com/products. [Accessed: 26-Nov-2014].

[5] H. Bao, *Real-time graphics rendering engine*. Springer Science & Business Media, 2011.

[6] Mechanical Simulation Corporation, "Driving Simulators (CarSim and TruckSim)," 2014. [Online]. Available: http://www.carsim.com/products/ds/index.php. [Accessed: 13-Nov-2014].

[7] Toyota Motor Corporation, "Toyota Global Site History of Toyota," 2014. [Online]. Available: http://www.toyota-global.com/innovation/safety_technology/safety_measurements/driving_simulator.html. [Accessed: 13-Nov-2014].

[8] I. 2007: 6th I. Conference and ... S., *Entertainment Computing - ICEC 2007: 6th International Conference, Shanghai, China, September 15-17, 2007, Proceedings*. Springer Science & Business Media, 2007.

[9] J. Slob, "State-of-the-Art driving simulators, a literature survey," *DCT Rep.*, no. August, 2008.

[10] I. Doron Precision Systems, "Doron Precision Systems |," 2011. [Online]. Available: http://www.doronprecision.com/. [Accessed: 28-Dec-2014].

[11] IRacing, "Online Racing - What is iRacing | iRacing.com," 2014. [Online]. Available: http://www.iracing.com/. [Accessed: 11-Nov-2014].

[12] iRacing, “See what the profesionals are saying | iRacing.com,” 2014. [Online]. Available: http://www.iracing.com/testimonials/. [Accessed: 11-Nov-2014].

[13] Slightly Mad Studios, “New Project CARS Weather Previews – WMD Portal,” 2014. [Online]. Available: http://www.wmdportal.com/projectnews/new-project-cars-weather-previews/. [Accessed: 12-Nov-2014].

[14] Slightly Mad Studios, “Inside Project CARS Seta Tire Model – WMD Portal,” 2014. [Online]. Available: http://www.wmdportal.com/projectnews/inside-project-cars-seta-tire-model/. [Accessed: 12-Nov-2014].

[15] Slightly Mad Studios, “Game Info - Project CARS,” 2014. [Online]. Available: http://www.projectcarsgame.com/game-info.html. [Accessed: 12-Nov-2014].

[16] Forward Development, “City Car Driving 1.3 Description.” [Online]. Available: http://citycardriving.com/products/citycardriving. [Accessed: 12-Nov-2014].

[17] D. Krajzewicz, G. Hertkorn, C. Rössel, and P. Wagner, “Sumo (simulation of urban mobility),” in *Proc. of the 4th Middle East Symposium on Simulation and Modelling*, 2002.

[18] D. Toffin, G. Reymond, a. Kemeny, and J. Droulez, “Influence of steering wheel torque feedback in a dynamic driving simulator,” *... Driv. Simul. ...*, vol. 2003, no. 1, pp. 1–11, 2003.

[19] T. R. M. Coeckelbergh, W. H. Brouwer, F. W. Cornelissen, P. Van Wolffelaar, and A. C. Kooijman, “The effect of visual field defects on driving performance: a driving simulator study.,” *Arch. Ophthalmol.*, vol. 120, no. 11, pp. 1509–1516, Nov. 2002.

[20] P. R. Desai, P. N. Desai, K. D. Ajmera, and K. Mehta, “A Review Paper on Oculus Rift-A Virtual,” *Int. J. Eng. Trends Technol.*, vol. 13, no. 4, pp. 175–179, 2014.

[21] E. Zeeb, “Daimler ’ s New Full-Scale , High-dynamic Driving Simulator – A Technical Overview,” *Driv. Simul. Conf. 2010 Eur.*, pp. 157–165, 2012.

[22] G. Kotusevski and K. Hawick, “A review of traffic simulation software,” vol. 13, pp. 35–54, 2009.

[23] L. S. Passos, R. J. F. Rossetti, and Z. Kokkinogenis, “Towards the next-generation traffic simulation tools: a first appraisal,” *6th Iber. Conf. Inf. Syst. Technol. (CISTI 2011)*, pp. 1–6, 2011.

[24] T. V. Mathew and K. Rao, “Microscopic traffic flow modeling,” Mumbai India, 2007.

[25] T. V. Mathew, “Lecture notes in Traffic Engineering And Management,” 2014. [Online]. Available: http://www.civil.iitb.ac.in/tvm/1111_nptel/535_TrSim/plain/plain.html.

[26] A. Wegener, M. Piórkowski, M. Raya, H. Hellbrück, S. Fischer, and J.-P. Hubaux, “TraCI,” *Proc. 11th Commun. Netw. Simul. Symp. - CNS ’08*, p. 155, 2008.

[27] Namdre and Dkrajzew, “Sumo at a Glance,” 2014. [Online]. Available: http://sumo.dlr.de/wiki/Sumo_at_a_Glance.

[28] N. T. Ratrout, S. M. Rahman, and S. Arabia, “a Comparative Analysis of Currently Used Microscopic and Macroscopic Traffic,” *Science (80-.).*, vol. 34, no. 1, pp. 121–133, 2009.

[29] W. Burghout, “Hybrid microscopic-mesoscopic traffic simulation,” 2004.

[30] W. Burghout and H. Koutsopoulos, “Hybrid Traffic Simulation Models: Vehicle Loading at Meso--Micro Boundaries,” *Int. Symp. Transp. Simul.*, 2006.

[31] D. Ni, “2DSIM: A prototype of nanoscopic traffic simulation,” *IEEE IV2003 Intell. Veh. Symp. Proc. (Cat. No.03TH8683)*, pp. 47–52, 2003.

[32] M. Miska and M. Kuwakara, “Nanoscopic Traffic Simulation with Integrated Driving Simulator to Investigate the Sag Curve Phenomenon,” *SEISAN KENKYU*, vol. 63, pp. 153–158, 2011.

[33] R. M. Smelik, T. Tutenel, R. Bidarra, and B. Benes, “A Survey on Procedural Modelling for Virtual Worlds,” *Comput. Graph. Forum*, vol. 33, no. 6, p. n/a–n/a, Sep. 2014.

[34] J. Rossignol, “Interview: Codies on FUEL,” *Rock, Paper, Shotgun*, 2011. [Online]. Available: http://www.rockpapershotgun.com/2009/02/24/interview-codies-on-fuel/. [Accessed: 29-Dec-2014].

[35] M. Bertuch, *“Klötzchenwelten” [Worlds of little blocks] in c’t Magazin issue 04/2009*. Hannover: Heise Zeitschriften Verlag GmbH & Co. KG, 2009.

[36] GamePedia, “Seed (level generation) - Minecraft Wiki,” *Curse, Inc*, 2014. [Online]. Available: http://minecraft.gamepedia.com/Seed_%28level_generation%29. [Accessed: 29-Dec-2014].

[37] Hello Games, “About | No Man’s Sky,” 2014. [Online]. Available: http://www.no-mans-sky.com/about/. [Accessed: 30-Dec-2014].

[38] Game Informer, “A Behind-The-Scenes Tour Of No Man’s Sky's Technology,” 2014. [Online]. Available: https://www.youtube.com/watch?v=h-kifCYToAU. [Accessed: 30-Dec-2014].

[39] D. G. Green and T. R. J. Bossomaier, *Complex Systems: From Biology to Computation*. 1993.

[40] P. Prusinkiewicz, “Graphical modeling using L-systems,” *The Algorithmic Beauty of Plants*, pp. 1–50, 1990.

[41] K. Sp and K. Sp, “Language modelling’s generative model: is it rational?,” *Computer (Long. Beach. Calif).*, pp. 1–14, 2004.

[42] R. Berndt, D. W. Fellner, and S. Havemann, “Generative 3D models,” *WEB3D - Int. Conf. 3D Web Technol.*, vol. 1, no. 212, pp. 111–121, 2005.

[43] J. M. Snyder and J. T. Kajiya, “Generative modeling,” *ACM SIGGRAPH Computer Graphics*, 1992. [Online]. Available: http://www.generative-modeling.org/. [Accessed: 29-Jan-2015].

[44] R. M. Smelik, T. Tutenel, K. J. de Kraker, and R. Bidarra, “A Proposal for a Procedural Terrain Modelling Framework,” *EGVE Symp.*, pp. 1–4, 2008.

[45] S. Stachniak and W. Stuerzlinger, “An Algorithm for Automated Fractal Terrain Deformation,” *Proc. Comput. Graph. Artif. Intell.*, pp. 64–76, 2005.

[46] R. M. Smelik, K. J. De Kraker, S. a. Groenewegen, T. Tutenel, and R. Bidarra, “A survey of procedural methods for terrain modelling,” *3AMIGAS - 3D Adv. Media Gaming Simul.*, pp. 25–34, 2009.

[47] Consortium for Spatial Information, “CGIAR-CSI SRTM 90m DEM Digital Elevation Database,” 2004. [Online]. Available: http://srtm.csi.cgiar.org/. [Accessed: 10-Feb-2015].

[48] S. Schmitt, “World Machine Features,” 2015. [Online]. Available: http://www.world-machine.com/about.php?page=features. [Accessed: 29-Jan-2015].

[49] N. Doldersum, “WorldComposer |.” [Online]. Available: http://www.terraincomposer.com/worldcomposer/. [Accessed: 29-Jan-2015].

[50] H. Honda, “Description of the form of trees by the parameters of the tree-like body: effects of the branching angle and the branch length on the sample of the tree-like body.,” *J. Theor. Biol.*, vol. 31, no. 2, pp. 331–338, 1971.

[51] P. Prusinkiewicz, “Graphical Applications of L-Systems,” in *Graphics Interface*, 1986, pp. 247–253.

[52] J. Arvo, D. Kirk, and A. Computer, “Modeling Plants with Environment-Sensitive Automata,” *Proc. Ausgraph ’88*, pp. 27–33, 1988.

[53] N. Greene, “Voxel space automata: modeling with stochastic growth processes in voxel space,” *ACM SIGGRAPH Comput. Graph.*, vol. 23, no. 3, pp. 175–184, 1989.

[54] B. Benes and E. U. Millan, “Virtual climbing plants competing for space,” *Proc. Comput. Animat. 2002 (CA 2002)*, 2002.

[55] O. Deussen, P. Hanrahan, B. Lintermann, R. Měch, M. Pharr, and P. Prusinkiewicz, “Realistic Modeling and Rendering of Plant Ecosystems,” *Conf. Comput. Graph. Interact. Tech.*, vol. 98, no. 1, pp. 275–286, 1998.

[56] G. Kelly and H. McCabe, “A survey of procedural techniques for city generation,” *ITB J.*, pp. 87–130, 2006.

[57] J. Martin, “Algorithmic beauty of buildings methods for procedural building generation,” *Comput. Sci. Honor. Theses*, p. 4, 2005.

[58] Esri, “Esri CityEngine | 3D Modeling Software for Urban Environments.” [Online]. Available: http://www.esri.com/software/cityengine. [Accessed: 29-Jan-2015].

[59] M. Qi, J. K. Jia, Y. L. Xu, and K. Li, “Automatic Extraction of the Building Rectilinear Rooftops Based on Region over Growing,” *J. Comput.*, vol. 9, no. 10, 2014.

[60] R. G. Laycock, “Automatically Generating Roof Models from Building Footprints,” *Wscg 2003*, pp. 4–7, 2003.

[61] E. Science, “HEIGHT ESTIMATION FOR BUILDINGS WITH COMPLEX CONTOURS IN MONOCULAR SATELLITE / AIRBORNE IMAGES BASED ON FUZZY REASONING Mohammad Izadi and Parvaneh Saeedi,” pp. 4293–4296, 2010.

[62] J. Derrough, “UnitySlippyMap!,” 2014. [Online]. Available: http://jderrough.blogspot.pt/2012/12/unityslippymap.html. [Accessed: 30-Apr-2015].

[63] J. Bennett, *OpenStreetMap*. Packt Publishing Ltd, 2010.

[64] P. Neis, D. Zielstra, and A. Zipf, “The Street Network Evolution of Crowdsourced Maps: OpenStreetMap in Germany 2007–2011,” *Futur. Internet*, vol. 4, no. 1, pp. 1–21, Dec. 2011.

[65] G. Moritz and C. Dornhege, “Realistic Cities in Simulated Environments - An Open Street Map to Robocup Rescue Converter,” *Inst. fur Inform. Univ. Freibg.*

[66] D. Krajzewicz and G. Hertkorn, “SUMO (Simulation of Urban MObility) An open-source traffic simulation,” *... Symp. Simul. ...*, 2002.

[67] J. Macedo, G. Soares, Z. Kokkinogenis, D. Perrotta, and C. Algoritmi, “10 A Framework for Electric Bus Powertrain Simulation in Urban Mobility Settings: coupling SUMO with a Matlab/Simulink nanoscopic model,” *1st SUMO User Conf. 2013*, vol. 21, pp. 96–103, 2011.

[68] 7-Zip, “7-Zip.” [Online]. Available: http://www.7-zip.org/. [Accessed: 12-Feb-2015].

[69] I. Ahmed and S. Janghel, “3D Animation : Don ’ t Drink and Drive,” *Int. J. u- e- Serv. Sci. Technol.*, vol. 8, no. 1, pp. 415–426, 2015.

[70] M. Goetz and A. Zipf, “OpenStreetMap in 3D – Detailed Insights on the Current Situation in Germany,” *City*, no. 1, pp. 3–7, 2012.

[71] M. Goetz and A. Zipf, “Towards Defining a Framework for the Automatic Derivation of 3D CityGML Models from Volunteered Geographic Information,” *Int. J. 3-D Inf. Model.*, vol. 1, no. 2, pp. 1–16.

[72] N. Neubauer, M. Over, a Schilling, and a Zipf, “Virtual Cities 2.0: Generating web-based 3D city models and landscapes based on free and user generated data (OpenStreetMap),” *GeoViz2009, Hambg.*, pp. 1–4, 2009.

[73] C. Olaverri-Monreal and R. J. F. Rossetti, “Human Factors in Intelligent Transportation Systems,” *Trans. Intell. Transp. Syst.*, vol. 15, no. 4, pp. 1734–1737, 2014.

[74] J. Gonçalves, R. J. F. Rossetti, and C. Olaverri-Monreal, "IC-DEEP: A serious games based application to assess the ergonomics of in-vehicle information systems," *IEEE Conf. Intell. Transp. Syst. Proceedings, ITSC*, pp. 1809–1814, 2012.

[75] P. R. J. A. Alves, J. Goncalves, R. J. F. Rossetti, E. C. Oliveira, and C. Olaverri-Monreal, "Forward collision warning systems using heads-up displays: Testing usability of two new metaphors," *IEEE Intell. Veh. Symp. Proc.*, pp. 1–6, 2013.

[76] J. S. V. Goncalves, R. J. F. Rossetti, J. B. Jacob, J. C. Goncalves, C. C. Olaverri-Monreal, A. B. Coelho, and R. B. Rodrigues, "Testing Advanced Driver Assistance Systems with a serious-game-based human factors analysis suite," *IEEE Intell. Veh. Symp. Proc.*, pp. 13–18, 2014.

[77] J. Goncalves, J. S. V. Goncalves, R. J. F. Rossetti, and C. Olaverri-Monreal, "Smartphone sensor platform to study traffic conditions and assess driving performance," *IEEE Conf. Intell. Transp. Syst. Proceedings, ITSC*, pp. 2596–2601, 2014.

[78] R. J. F. Rossetti and S. Bampi, "A software environment to integrate urban traffic simulation tasks," *J. Geogr. Inf. Decis. Anal.*, vol. 3, no. 1, pp. 56–63, 1999.

[79] R. J. F. Rossetti, E. C. Oliveira, and A. L. C. Bazzan, "Towards a specification of a framework for sustainable transportation analysis," *13th Port. Conf. Artif. Intell.*, 2007.

[80] P. J. Bentley and J. Timmis, "Guest Editorial Special Issue on Artificial Immune Systems," *Trans. Intell. Transp. Syst.*, vol. 12, no. 2, pp. 309–312, 2003.

[81] R. J. F. Rossetti, J. E. Almeida, Z. Kokkinogenis, and J. Goncalves, "Playing transportation seriously: Applications of serious games to artificial transportation systems," *IEEE Intell. Syst.*, vol. 28, no. 4, pp. 107–113, 2013.

[82] I. J. P. M. Timóteo, M. R. Araújo, R. J. F. Rossetti, and E. C. Oliveira, "TraSMAPI: An API oriented towards multi-agent systems real-time interaction with multiple traffic simulators," *IEEE Conf. Intell. Transp. Syst. Proceedings, ITSC*, pp. 1183–1188, Sep. 2010.

[83] iRacing, "In-game screenshots | iRacing.com," 2014. [Online]. Available: http://www.iracing.com/screenshots/. [Accessed: 12-Nov-2014].

[84] Slightly Mad Studios, "Project CARS Community Gallery #78 – WMD Portal," 2014. [Online]. Available: http://www.wmdportal.com/projectnews/project-cars-community-gallery-78/. [Accessed: 12-Nov-2014].

[85] Forward Development, "SCREENSHOTS - City Car Driving." [Online]. Available: http://citycardriving.com/gallery/category/12-citycardriving?start=40. [Accessed: 12-Nov-2014].

[86] Logitech, "Logitech G27 Racing Wheel," 2009. [Online]. Available: http://thecoolgadgets.com/wp-content/uploads/2009/08/logitech_g27_gaming_wheel.jpg. [Accessed: 29-Jan-2015].

[87] PlaySeat, “Racing seat,” 2012. [Online]. Available: http://ecx.images-amazon.com/images/I/51VywKSHv8L.jpg. [Accessed: 29-Jan-2015].

[88] Vanhese, “Motion Base 301,” 2015. [Online]. Available: http://www.vanhese.de/wp-content/uploads/2015/02/301simulator2.jpg. [Accessed: 29-Jan-2015].

[89] Lexus, “Real Scale Driving Simulator.” [Online]. Available: http://www.lexus-int.com/design/images/virtual-reality/mainImg02.jpg. [Accessed: 29-Jan-2015].

[90] L. Oculus VR, “Oculus Rift DK2,” 2015. [Online]. Available: https://dbvc4uanumi2d.cloudfront.net/cdn/4.3.0/wp-content/themes/oculus/img/order/dk2-product.jpg. [Accessed: 29-Jan-2015].

[91] J. Leno, “Jay Leno tests the real car simulator.” [Online]. Available: http://images.thecarconnection.com/med/jay-leno-tests-the-real-car-simulator_100340696_m.jpg. [Accessed: 29-Jan-2015].

[92] ModDB, “FUEL,” 2010. [Online]. Available: http://media.moddb.com/images/games/1/14/13397/FUEL_2010-05-31_18-40-01-90.png. [Accessed: 29-Jan-2015].

[93] Minecraft, “Minecraft Voxel World,” 2011. [Online]. Available: http://nicolejeanette.me/wp/wp-content/uploads/2011/09/2011-09-16_22.05.35.png. [Accessed: 29-Jan-2015].

[94] Hello Games, “No Man’s Sky,” 2014. [Online]. Available: http://cdn2.vox-cdn.com/assets/4646831/no-mans-sky-gallery-06.jpg. [Accessed: 29-Jan-2015].

[95] Zapatopi, “Fractal,” 2012. [Online]. Available: http://vicdesotelle.com/wp-content/uploads/2012/09/fractalscalingsocialinnovation.jpeg. [Accessed: 29-Jan-2015].

[96] GML, “GML Gothic Window Thickness,” 2014. [Online]. Available: http://upload.wikimedia.org/wikipedia/commons/2/27/GML-Gothic-Window-Thickness_%281%29.jpg. [Accessed: 29-Jan-2015].

[97] S. Schmitt, “World Machine,” 2013. [Online]. Available: http://www.world-machine.com/images/terrain3.jpg. [Accessed: 29-Jan-2015].

[98] WorldComposer, “WorldComposer,” 2012. [Online]. Available: http://www.terraincomposer.com/wp-content/uploads/2012/12/F15E.jpg. [Accessed: 31-Jan-2015].

[99] BuildR, “Building.” [Online]. Available: https://d2ujflorbtfzji.cloudfront.net/package-screenshot/d7d55155-8ef8-4381-baaa-dd1fb07659eb_scaled.jpg. [Accessed: 29-Jan-2015].

[100] Esri, “Esri City Engine.” [Online]. Available: http://blenderartists.org/forum/attachment.php?attachmentid=299726&d=1396350264. [Accessed: 29-Jan-2015].

References

[101] TriumphRat.net, “Triumph Daytona 675 Engine Diagram,” 2011. [Online]. Available: http://www.triumphrat.net/attachments/street-triple-forum/43366d1321773455-technical-engine-drawings-triumph-675-motor-diagram.jpg.

[102] Blue Pigeons, “Drum Brakes,” 2014. [Online]. Available: https://pigeonsblue.files.wordpress.com/2014/04/drumbrakemechanismstageswa.gif.

[103] Howstuffworks, “How Disk Brake Works,” 2000. [Online]. Available: http://s.hswstatic.com/gif/disc-brake3.jpg.

[104] Genious, “How Power Steering Works.” [Online]. Available: http://s3.amazonaws.com/rapgenius/Steering.jpg.

Table of Contents

Introduction **1**
1.1 Motivations and Goals 1
1.2 Structure of the Dissertation 2
Literature Review **3**
2.1 Game Engines 3
2.2 Simulation Engines 4
2.3 Physics Engine 4
2.3.1 PhysX 4
2.3.2 Havok 5
2.4 Render Engine 5
2.5 Driving Simulators 5
2.5.1 Entertainment 6
2.5.2 Research 6
2.5.3 Training 6
2.5.4 The Latest Driving Simulators 7
2.5.5 Driving Simulator Peripherals 10
a. Steering Wheels 10
b. Vibration Chairs 11
c. Force Dynamic Simulator 11
d. Big Round Screen 12
e. Surround Sound System 13
f. Head Tracking 13
g. Real Vehicles 14
h. Extreme Experience 14
2.6 Traffic Simulators 14

2.6.1 Microscopic 15
2.6.2 Macroscopic 15
2.6.3 Mesoscopic 15
2.6.4 Hybrid 16
2.6.5 Nanoscopic 16
2.7 Procedural Modelling 17
2.7.1 Procedural Modelling and Games 17
19
2.7.2 Procedural Modelling Techniques 20
a. L-Systems 20
b. Fractals 21
c. Generative Modelling Language 22
2.7.3 Procedural Modelling Main Usage 22
2.7.3.1 Terrain 23
a. Height Maps 24
b. World Machine 25
c. World Composer 26
2.7.3.2 Flora 27
2.7.3.3 Buildings 28
a. Geometric Primitives 28
b. L-Systems 28
c. Building Generation Tools 29
2.7.3.4 Roads 29
a. Grid Layout 29
b. L-Systems 29
2.7.3.5 Cities 30
a. Esri CityEngine 30
2.7.4 Extracting Rooftops 31
2.8 Tools to be Used 32
2.8.1 Unity 5 32
2.8.2 UnitySlippyMap 32
2.8.3 Open Street Map 33
2.8.4 Google Static Maps API 33
2.8.5 SUMO Simulation of Urban MObility 33
2.8.6 CGIAR-CSI 33
2.8.7 7zip 34
2.8.8 Blender 34
2.9 Related Work 34
2.10 Summary 35

Methodological Approach and System Architecture 36

3.1 Problem to Solve 36
3.2 System Architecture 38
3.3 Unity3D + SUMO (Simulation of Urban MObility) 39
3.4 Development 40
3.5 Implementation 42
3.5.1 Main Menu 42
3.5.2 Main Menu Possible Configurations 42
3.5.3 UnitySlippyMap 43
3.5.4 Elevation Data 44
3.5.5 Location Data Parsing 45
3.5.6 Satellite Image Gathering 47
3.5.7 Terrain Elevation 48
3.5.8 Procedural Roads 50
a. Road Enlightenment 54
3.5.9 Pedestrians 55
3.5.10 Procedural Buildings 58
3.5.11 Procedural Barriers 61
3.5.12 Procedural Land Uses 62
3.5.13 Procedural Amenities or City Furniture 65
3.5.14 Day and Night Cycle 65
3.5.15 Weather Cycle 67
3.5.16 Vehicle 68
a. Vehicle Selection 68
b. Vehicle Handling 68
c. Vehicle Engine 69
d. Vehicle Brakes 72
e. Vehicle Steering 73
f. Vehicle Illumination 74
g. Vehicle Cameras 75
h. Oculus Rift Integration 76
i. Vehicle Damage 77
3.5.17 Visual Effects 77
3.5.18 Procedural Scene Export 79
3.6 Summary 79

Results 81
4.1 Main Menu 82
4.2 Main Menu Possible Configurations 82
4.3 UnitySlippyMap 83
4.4 Elevation Data 84
4.5 Location Data Parsing and Satellite Image Gathering 85

4.6 Terrain Elevation 85
4.7 Procedural Roads 86
4.7.1 Road Enlightenment 87
4.8 Pedestrians 88
4.9 Procedural Buildings 89
4.10 Procedural Barriers 90
4.11 Procedural Land Uses 90
4.12 Procedural Amenities or City Furniture 91
4.13 Day and Night Cycle 91
4.14 Weather Cycle 92
4.15 Vehicle 92
4.15.1 Vehicle Selection 93
4.15.2 Vehicle Dynamics 94
4.15.3 Vehicle Illumination 95
4.15.4 Oculus Rift Integration 95
4.15.5 Vehicle Damage 96
4.16 Visual Effects 97
4.17 Procedural Scene Export 98
4.18 Summary 99

Conclusion and Future Work 100
5.1 Goal Satisfaction 100
5.2 Future Work 102

References 104

Table of Contents 112