ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Ανάπτυξη εφαρμογής ηλεκτρονικού παιχνιδιού με τη μηχανή Unreal Engine

Electronic Game Development with Unreal Engine

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Μπακανιάρης Πέτρος

Επιβλέπων Καθηγητής:

Μιχαήλ Βασιλακόπουλος
Αναπληρωτής Καθηγητής

Συνεπιβλέπουσα Καθηγήτρια:

Παναγιώτα Τσομπανοπούλου
Αναπληρώτρια Καθηγήτρια

Βόλος, Σεπτέμβριος 2016

.....

Πέτρος Μπακανιάρης

Προπτυχιακός φοιτητής του Τμήματος Ηλεκτρολόγων Μηχανικών και

Μηχανικών Υπολογιστών, Πανεπιστημίου Θεσσαλίας

Ευχαριστίες

Στην οικογένεια μου για τη στήριξη όλα αυτά τα χρόνια, στους φίλους μου που δεν έλειψαν ποτέ, στον καθηγητή μου κύριο Βασιλακόπουλο για την καθοδήγηση του, και στα μέλη της κοινότητας Unreal Engine Developers Community για την αφιλοκερδή βοήθεια που προσφέρουν σε νεαρούς Game Developers

Περίληψη

Η ανάγκη για διασκέδαση είναι έμφυτη στον άνθρωπο. Ανέκαθεν οι ανθρωποι προσπαθούσαν να βρούνε τρόπους για ψυχαγωγία και διασκέδαση. Με τη δημιουργία και εξέλιξη των ηλεκτρονικών συσκευών, η ανάγκη αυτή βρήκε ακόμα έναν τρόπο για να ικανοποιηθεί. Τα ηλεκτρονικά παιχνίδια. Τα ηλεκτρονικά παιχνίδια έκαναν την εμφάνιση τους στα μέσα του προηγούμενου αιώνα. Από τότε, η εκθετική ανάπτυξη της τεχνολογίας έχει οδηγήσει στην δημιουργία μιας ολόκληρης βιομηχανίας γύρω από τα ηλεκτρονικά παιχνίδια. Σήμερα, με την διάδοση των οικιακών υπολογιστών, όπως επίσης και με την καθολική χρήση των smartphone, σχεδόν ο κάθε άνθρωπος μπορεί να έχει πρόσβαση σε κάποιο ηλεκτρονικό παιχνίδι.

Ενώ αρχικά η ανάπτυξη ηλεκτρονικών παιχνιδιών γινόταν με απλά προγράμματα, η ανάγκη δημιουργίας πιο σύνθετων παιχνιδιών οδήγησε στην δημιουργία πιο σύνθετων προγραμμάτων με περισσότερες δυνατότητες. Σήμερα, για τη δημιουργία ενός ηλεκτρονικού παιχνιδιού συνήθως χρειάζεται μια ομάδα από προγραμματιστές, στην οποία υπάρχουν ορισμένοι ρόλοι. Ένας από αυτούς τους ρόλους είναι ο game programmer, ο οποίος αναπτύσσει τον κώδικα για το παιχνίδι.

Στη συνέχεια της εργασίας αναπτύσσεται ένα ένα υποτυπώδες ηλεκτρονικό παιχνίδι που ενσωματώνει αρκετές από τις λειτουργίες τις οποίες αναπτύσσει ένας game programmer.

Στοιχεία των λειτουργιών που θα αναπτυχθούν είναι τα εξής:

- Σύντομη παρουσίαση του εργαλείου Unreal Engine
- Σύγκριση με άλλες μηχανές ανάπτυξης παιχνιδιών
- Βήμα προς βήμα της διαδικασίας ανάπτυξης της εφαρμογής, με επεξήγηση των μεθόδων, λειτουργιών και κλάσεων που

χρησιμοποιήθηκαν

- Αξιολόγηση της εφαρμογής, προτάσεις για βελτίωση και γενικότερα συμπεράσματα.

Η εφαρμογή διατίθεται για ηλεκτρονικούς υπολογιστές, με δυνατότητα εξαγωγής σε άλλες συσκευές παιχνιδιών όπως Playstation, Xbox κτλ.

Λέξεις-κλειδιά:
Game Development, Unreal Engine, C++, Blueprints, ηλεκτρονικά παιχνίδια

Abstract

The need for entertainment is inherent to the human being. All along, people have been trying to find new ways of joy and entertainment. With the creation and evolution of electronic devices, people found one more way to satisfy said need. Video games. Video games debuted in the middle of last century. Ever since, the exponential growth of technology has led to the creation of a new sector of industry around electronic games. Today, with the extended use of personal computers, as well as the almost universal use of smartphones, almost everyone can have access to video games.

While at first the development of video games was made with simple programs, the need to develop more complex games led to the creation of better programs with more capabilities. To develop a contemporary video game, usually a group of developers are needed, as well as other people to work in other posts. Each of these developers has a specific role, and two or more developers can have the same role. One of these roles is the game programmer, who is tasked with creating the code for the game.

During this thesis, we will develop a rudimentary video game which incorporates some of the functionalities a game programmer develops.

Elements of these functionalities include:

- A short presentation of Unreal Engine
- Comparison with other game engines
- Step by step development of our application, explaining the methods, functionality and classes we used.
- Evaluation of our application, pros and cons, suggestions on what could be done better, and general conclusions.

The application is developed for Android devices and Windows computers, with the capability to be deployed in other devices, such as Playstation and

Xbox

Keywords:
Game Development, Unreal Engine, C++, Blueprints, Video Games

Content Table

Chapter 1
Introduction

1.1. What is a Game Engine

To put it simply, a game engine is a program used to develop video games. Before game engines, video games were developed as singular entities[1]. When arcade games met their peak of success and popularity, in an era known as the Golden Age of Arcade Games[2] the technological innovation that followed brought about the creation of the first 2D game engines.

The term "game engine" arose in the mid-1990s, especially in connection with 3D games such as first-person shooters (FPS), like Doom (1993) and Quake III (1996). Due to the high success of these games, third-party developers licensed the core components of the games and modified them, building their on games around them. Games developed later that decade, such as Unreal Tournament (1998), were built with this approach in mind, that the game mechanics and game concept developed separately. Therefore, the engine used in this assignment, Unreal Engine, was created.

1.2. Development in recent years

In the following years, a number of factors contributed to the exponential growth of unreal engines. The most important one was the growth of the gaming industry, as well as the competition between industries that sparked the flame of innovation. Another important factor was the evolution of computer hardware, giving potential for growth. Threading is taking on more importance due to modern multi-core systems and increased demands in realism. Typical threads involve rendering, streaming, audio, and physics.

Nowadays, game engines are some of the most complex applications written, often featuring dozens of finely tuned systems interacting to ensure a precisely

controlled user experience. Furthermore, the continued evolution has created a strong separation between the jobs a developer has, such as rendering, scripting, artwork and level design. Game engines are also built upon higher level languages, such as C++, C#, Java, Lua Script, .NET and Python. This happens because most games are GPU-limited (i.e. limited by the power of the graphics card), the potential slowdown due to translation overheads of higher level languages becomes negligible, while the productivity gains offered by these languages work to the game engine developers' benefit. As game engines become more user-friendly, the creation of video games is becoming easier has led to the creation of independent video games, commonly referred as indie games. An indie game is a video game that is created without the financial support of a publisher. In the later half of the 21st century, indie gaming has seen a critical rise, due to newer of development tools, new online distribution methods, and new methods of crowdfunding, such as Kickstarter. While surveys show that most Indie games don't make significant profit [3], there are cases in which indie games compete against published video games, and even surpassed them. Suffice to mention are Minecraft and League of Legends, both of which started as indie games. The former one was originally developed by a single programmer, later later developed and published by Mojang. As of June 2016, over 106 million copies had been sold, with more than 40 million unique players each month across all platforms, making it the best-selling PC game to date and the second best-selling video game of all time. The latter, League of Legends, is a free to play multiplayer online game released in 2009. The game supports microtransactions, and as of January 2014, the game has 67 million active monthly players, 27 million per day. Today, these numbers have increased to 100 million active players per month. Riot games, the company responsible for its development, is centered around this game, with no other games under its wing. The company has offices in 18 big cities around the world [4].

Another important recent trend is the purpose of game development. During the last years, the application of game engines has broadened in scope and is no longer for pure entertainment. Serious Games, as they are called, are designed for and used in a number of other purposes. To name a few, serious games are used in visualization, training, education, scientific exploration, health care, management, city planning, engineering, politics, religion and military

training[5]. The scope of application of game engines is expected to further expand in later years, opening jobs and opportunities for more game developers to enter the game development industry.

As of today, there is a plethora of game engines, most of which are free for learning and non-profit development. Unreal Engine for example, which will be used in this assignment, is free for non-profit development, as well as learning, has no monthly subscription or purchase price, but requires a 5% royalty on released games. Unreal Engine is developed by Epic Games, first released in 1998. Currently, it is in its fourth edition, named Unreal Engine 4.

1.3. Future Trends

Due to hardware advancements, gaming technology greatly mirrors the advancement of technology overall. Therefore, the future is expected to be interesting and full of promise. Generally, there are some features that are expected to be seen in the future, and although we can't really predict what is to come, each of these features has a general concept, yet we don't know how much we will deviate from the intended use of the original creation.

One of these concepts is cloud gaming, a concept encouraged by the increased internet speed. To put it simply, with cloud gaming one will not need to download a game, but simply maintain a stable internet connection. Games will also be more easily accessible through more than one devices, such as computer and tablet. In recent years, Sony has been working towards this direction as it purchased Gaikai the world’s largest and most widespread cloud gaming service for $380 million dollars.

Another concept is the one of augmented reality. On top of seeing the actual environment, in augmented reality we are fed with more information. For example, an app that communicates with google maps allows us to see nearby places of interest through our phone camera. This can be applied to gaming, and we may eventually not need to hold a device or controller, but playing the game will feel like we are inside the gaming realm.

Last but not least, is the introduction of immersion, or virtual reality. Perhaps the most anticipated feature of all, Immersion can be described as having a physical presence in a non-physical world. The idea around it is to stimulate the user with images, sound and other stimuli to create a total virtual environment.

Although virtual reality is already here, the degree at which it resembles reality isn't great yet, this is expected to change in the future as technology progresses. The first devices for virtual reality, such as the Oculus rift, have already made their appearance on the market, in relatively low prices too. Immersion, combined with augmented reality, is expected to change the way video gaming is known today. These concepts can be applied in numerous other cases, some of which were stated before.

Unreal Engine, the engine we will be working on in this project, supports development for all virtual reality devices released so far, such as Samsung Gear VR, Google VR, Oculus Rift VR and Steam VR. Although the project is still in development, there are high hopes for it in the future [6].

1.4. Deduction

To sum up, we can say that game development is a rich technological area, progressing as fast as technology itself. The video game industry encompasses dozens of job disciplines and employs tens of thousands of people worldwide. It is a rapid growing industry, with a worldwide budget estimated at 93 billion dollars [7], and despite the world's economical status, it continues to grow. The future is also expected to be bright, both due to new technological breakthroughs and due to game development's employment for other purposes other than entertainment, as we stated before. Moreover, the continuous growing of independent game development opens up opportunities for people who would like to open their own companies. Given all these facts, we can assume that a job in the game development industry is a good and promising job to have.

During the rest of the assignment, I will present different ways of developing video games, compare them and note the advantages and disadvantages of each one of them, and then develop a video game using one of these game engines (Unreal Engine). The video game will include most of the function a game should include. More attention will be given to coding and scripting, and graphic development, which is a task of a completely different role, will be almost completely left out. Finally, the application will be evaluated and tested, and improvements that could be made to it will be pointed out.

Chapter 2
Game Engines

As stated before, the continuous expansion of the gaming industry has given birth to a plethora of game engines. While many game development studios use their own proprietary game engine there’s still a huge market for indie developers and even larger studios needing a great game engine to help them create their game. Every game engine can be classified by some characteristics. Most notable are the primary programming language, scripting language, it's 2D/3D orientation, the number of gaming platforms it targets and its license. A list of the most notable game engines can be shown in the table below:

V · T · E	Game engines (list)	[hide]
Source port · First-person shooter engine (list) · Tile engine · Game engine recreation (list) · Game creation system		
Free and open-source	2D	Adventure Game Studio · Beats of Rage · Cocos2d · Flixel · Jogre · KiriKiri · libGDX · Moai · OHRRPGCE · OpenFL · ORX · Pygame · Ren'Py · Stratagus · Thousand Parsec · VASSAL · Xconq
	2.5D	Aleph One · Cube
	3D	Away3D · Blender Game · Cafu · Crystal Space · Cube 2 · Delta3D · Dim3 · Goo Create · GLScene · Horde3D · HPL 1 · Irrlicht · id Tech 3 · id Tech 4 · JMonkey · OGRE · Open Wonderland · Panda3D · Papervision3D · Platinum Arts Sandbox Free 3D Game Maker · PlayCanvas · PLIB · Quake · Quake II · RealmForge · Retribution · Torque 3D
	Mix	Allegro · Construct Classic · Godot · Lightweight Java Game Library · Spring · Wintermute Engine
Proprietary	2D	Construct 2 · Corona · Clickteam Fusion · GameMaker: Studio · GameSalad · M.U.G.E.N · NScripter · RPG Maker · Southpaw · Stencyl · UbiArt Framework · Vicious · Virtual Theatre · V-Play · Zillions of Games
	3D	4A · Amazon Lumberyard · Anvil · Bork3D · C4 · Chrome · Clausewitz Engine · Creation · CryEngine · Crystal Tools · Diesel · Dunia · EAGL · EGO · Elflight · Enforce · Enigma · Essence · Flare3D · Fox · Frostbite · Geo-Mod · GoldSrc · HeroEngine · HydroEngine · HPL 2-3 · id Tech 5 · id Tech 6 · Ignite · IW · Jade · Kinetica · LS3D · LithTech · Luminous Studio · LyN · Marmalade · Mizuchi · MT Framework · Outerra · Panta Rhei · PhyreEngine · Q · Real Virtuality · REDengine · Refractor · Riot · RAGE · SAGE · Serious · Shark 3D · ShiVa · Silent Storm · Source · Titan · TOSHI · Truevision3D · Unigine · Unity · Unreal · Vision · Visual3D · XnGine · X-Ray · YETI · Zero
	Mix	Gamebryo · Hybrid Graphics · Kaneva Game Platform · Metismo
Historical interest		BRender · Build · Dark · Doom · Game-Maker · GameMaker · Garry Kitchen's GameMaker · Genesis3D · Genie · Filmation · Freescape · INSANE · Jedi · MADE · Pie in the Sky · RenderWare · Sim RPG Maker · Sith · Voxel Space · Wolfenstein 3D
Proprietary middleware (list)		Euphoria · Gameware · GameWorks · Havok · iMUSE · Kynapse · Quazal · SpeedTree · Xaitment · FaceGen

Picture 2.1

While some open source engines are very good, and some promising ones are under development (Blender Game), it is generally accepted that proprietary game engines include the best and non-outdated engines. This is due to the fact that game engines are some of the most complicated programs there are, therefore development takes up a lot of resources. On the bright side though, most proprietary engines, and the most powerful ones too, are either free for non-profit usage, or include a free version covers the needs of most individual developers.

Among all game engines, the two most popular game engines are Unity 3D and

Unreal Engine, with Cryengine following close by. Choosing between these two is pretty hard, and the choice depends on more than one reasons, the most important being the developer's personal preference.

After we give a brief summary for each of these engines, we will argue on our preference. Unreal Engine

2.1. Unreal Engine

Unreal Engine was developed by Epic games and was first showcased in 1998 in the first-person game Unreal. Over the years it has been used in a variety of other game genres, such as MMORPGs (Massive Multiplayer Online Role Playing Games), stealth, puzzle and vehicle games among others. Its code is written in C++, allowing for great exeution spped and a high degree of portability. The current release is Unreal Engine 4, designed for Microsoft's DirectX 11 and 12 (for Microsoft Windows, Xbox One, Windows RT); GNM (for PlayStation 4); OpenGL (for OS X, Linux, iOS, Android, Ouya and Windows XP); Vulkan (for Android); Metal (for iOS); and JavaScript/ WebGL (for HTML5 Web browsers) [8][9][10]

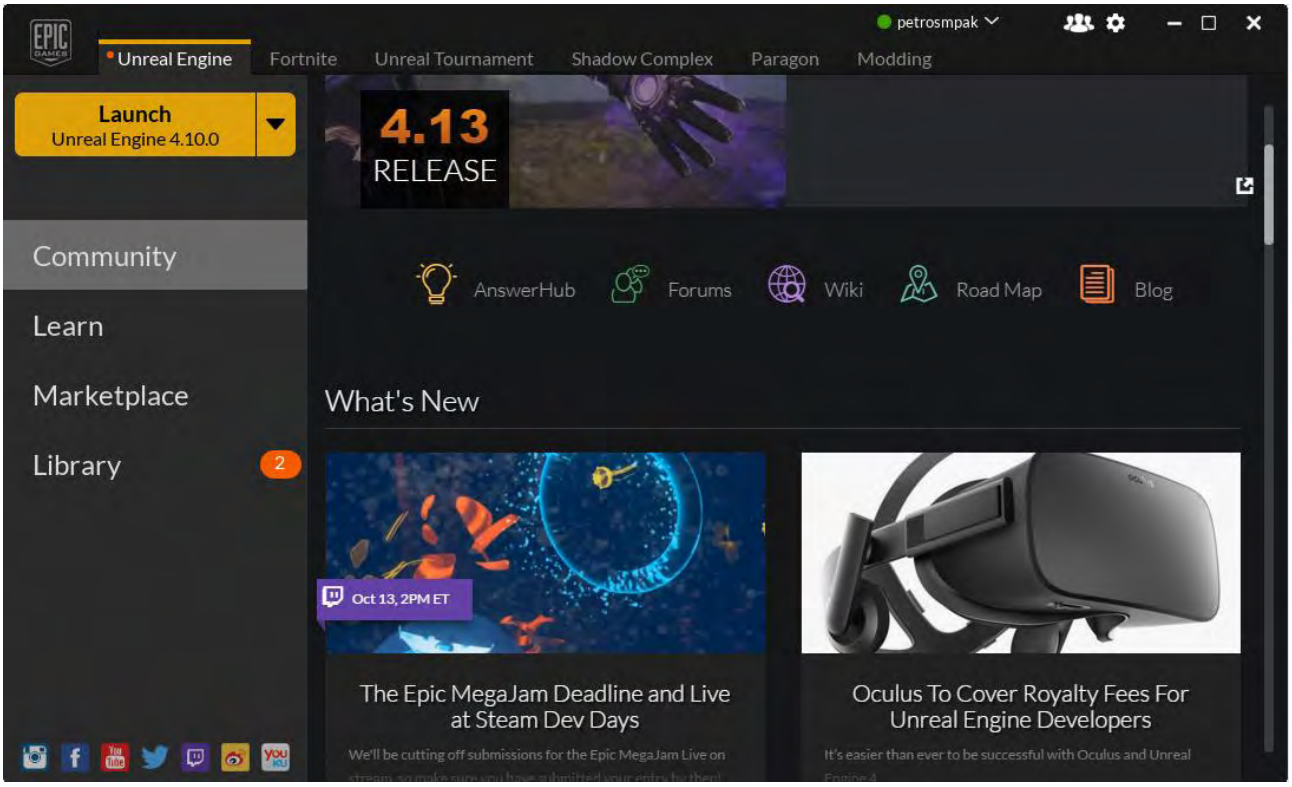
2.1.1. License

While Unreal Engine gives access to its source code, it is not open source, as there are certain limitations in its license of use. For example, distribution of the source code to non-licensed users is not allowed, as well as posting more than 30 lines of code on the internet is too. This doesn't affect developers much though.

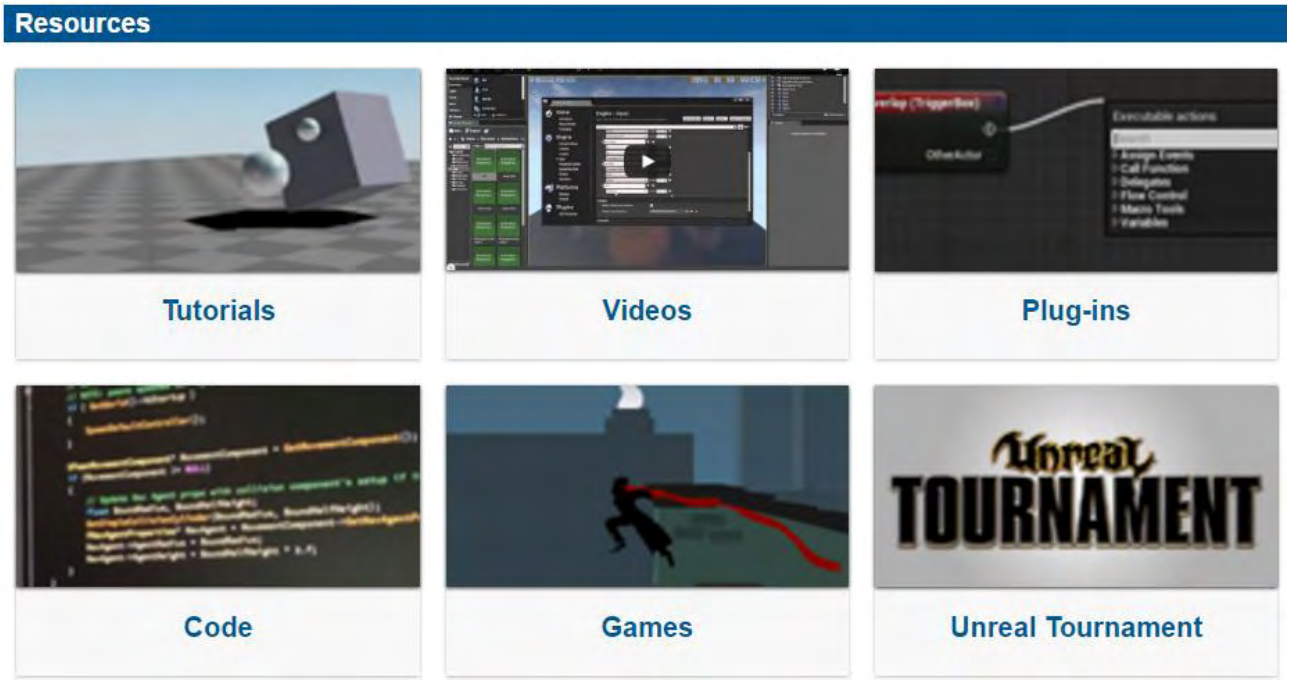
When it comes to pricing, which is one of the most important aspects a developer has to consider, Epic Games has changed their policy quite a few times over the years, each time towards a more free and open software. In the past, Unreal engine came with a monthly subscription, or free for students only, but as of today, it comes free for everyone and ony includes a royalty free for released games [11]. The fee is at 5% of the game's earnings, per calendar quarter (including microtransactions and in-game advertisements), and applies after the first 3 thousand dollars earned. This means that for developers who won't earn more than 3000$ per game or film, Unreal Engine is completely free.

2.1.2. Ease of Use

Another issue developers have to consider, especially young aspiring ones, is how easily an inexperienced user can start working with Unreal Engine. For those familiar with game development, Unreal Engine comes with an easy User interface, making it easy to utilize its functionality. Furthermore, there is an online guide with step by step instructions and tips about how to switch from unity to unreal. This guide requires a basic knowledge of Unity 3d, the second most popular game engine, but is perfect for developers who seek to take full advantage of unreal's next generation graphics.
For new game developers, there is also an enormous number of guides, tutorials and resources online, an official youtube channel with over 800 tutorial videos. Besides that, there are also many independent developers posting their tutorials, some of whom live stream their work through twitch. In the official Unreal engine site there is also a documentation, including every detail a developer needs to master game development with unreal engine. Complementary to what is mentioned above, there are also numerous communities in every social media, counting tens of thousands on members each, willing to help fellow developers progress. In the picture 2.1 below, we can see a screenshot of the "learning tab" from the Unreal Engine Launcher. The five icons on top contain links for the "AnswerHub" [12], "Forums"[13], "Wiki" [14], "Road Map" and "Blog"[15]. The AnswerHub is a place where developers can post their question regarding development. Users can also search for questions. With hundreds, maybe even thousands of questions posted, chances are most questions are already there. The second icon points to the forums, where the user can interact with the Unreal community. The third points to the Unreal Engine Wiki, where an enormous amount of resources can be found and downloaded. As seen in picture 2.2, videos, tutorials, source code, and even full games can be found here. Last but not least, the Unreal Engine Blog contains news regarding epic games and Unreal Engine.



Picture 2.2



Picture 2.3

The Learn tab also contains a structured collection of tutorials and resources, covering virtually every aspect of the engine.

When it comes to functionality, Unreal's programming language is C++, a

widely known programming language, and as some people say, a must-know language for programmers, and a programming language many developers are already familiar with. Unreal also includes its own visual scripting language, called Blueprints. Although there are many limitations and disadvantages in the use of Blueprints compared to C++, visual scripting is something anyone can learn, making it possible for people with no programming background to develop games.

2.1.3. Asset store

Unreal Engine also comes with its own marketplace. Although lots of resources can be found on the internet, each of which has its own license, yet many are completely free, the marketplace has some of the top quality resources available, which come at a cost in most cases though. The marketplace features 18 categories, each of which has lots of resources. These resources are submitted by individual developers, therefore creating and selling resources is another job a developer may choose.

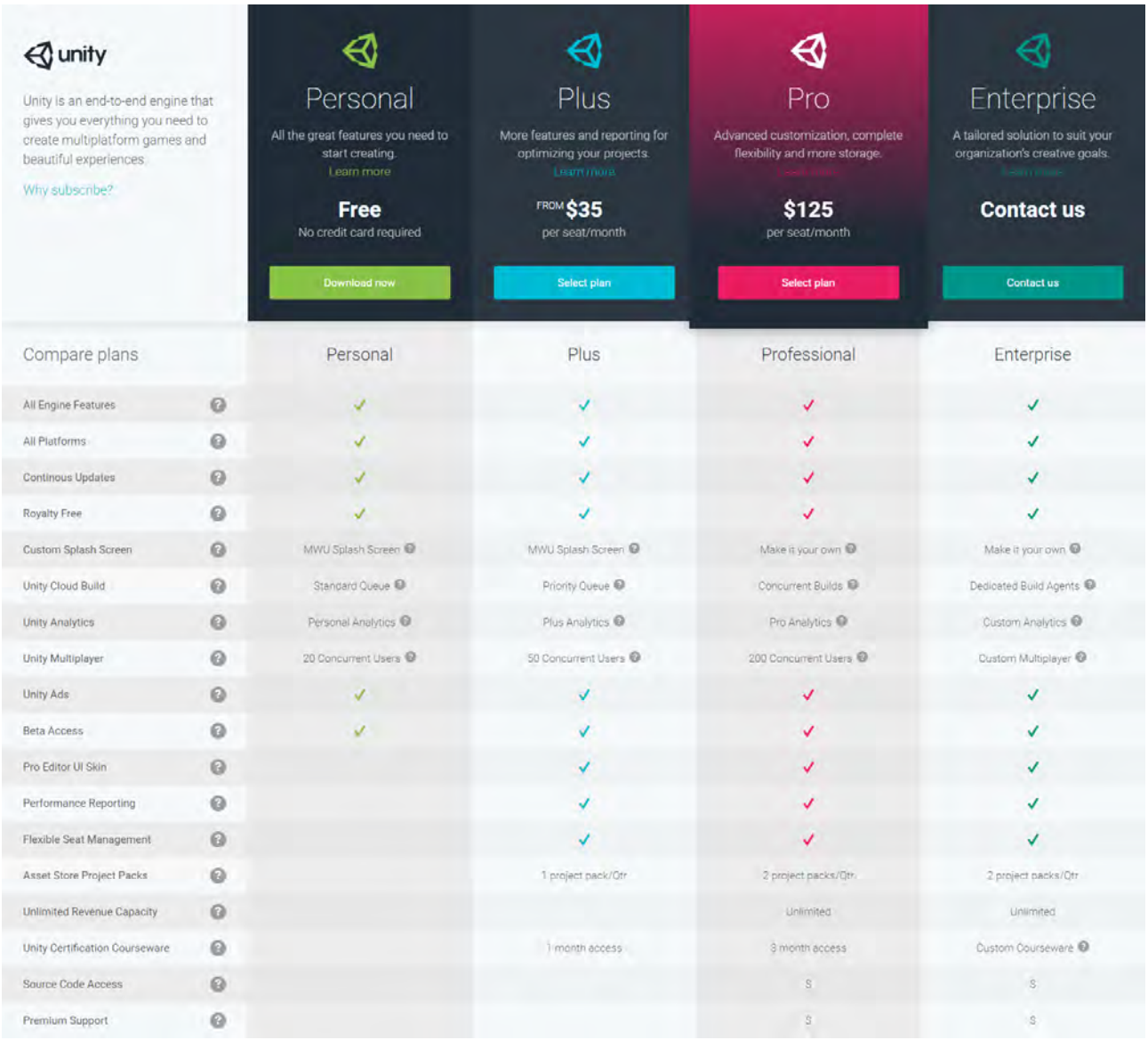
2.2. Unity Engine

Unity is a cross-platform game engine developed by Unity Technologies and used to develop video games for PC, consoles, mobile devices and websites. First announced only for OS X in 2005, at Apple's Worldwide Developers Conference, it has since been extended to target 21 platforms. Nintendo developers are provided with a free Unity 5 license, along with software development kits (SDKs) for Nintendo consoles. Unity allows the developer to choose the main programming language, offering he choice between C# and Javascript.

Like Unreal, Unity emphasizes on portability. Therefore, it targets a variety of APIs, like Direct3D for windows and Xbox, OpenGL on Mac, Linux and windows, OpenGL ES on Android and iOS, as well as proprietary APIs on video game consoles. Within a game project, the developer has control over delivery to mobile devices, web browsers, browsers, desktops and consoles. Unity is the default software development kit (SDK) for Nintendo's Wii U video game console platform, with a free copy included by Nintendo with each Wii U developer license [16]

2.2.1. License

Unity comes with a proprietary license, which comes with quite a few options. These options range from a free personal use, to an enterprise version aimed for big companies and corporations. More specifically, there are four subscription options, which are shown in the below image. Generally, the subscription method has received mixed reactions from developers, with desktop developers having the most negative reactions.



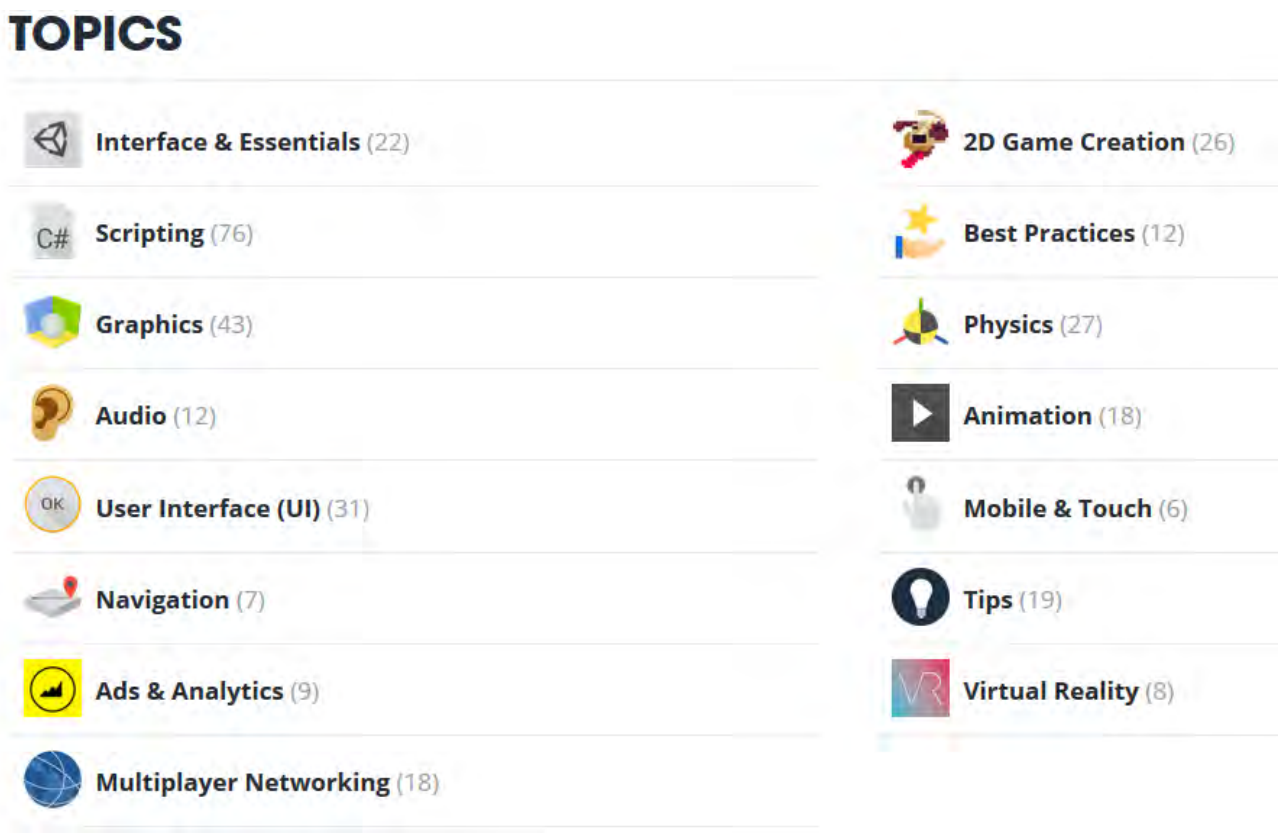
Picture 2.4

As we can see, there are quite a few differences between the licensing between Unreal and Unity, which we will explore in the end of this chapter.

2.2.2. Ease of Use

One of the most important things, especially for young developers is the ease of use. Unity 5, the current engine version, comes with an easy to grasp user interface. Generally, Unity is considered by many to be the easiest game engine for people with no development experience or coding background. As stated before, Unity's programming language can be either C# or javascript, allowing the developer to choose the language of preference.

Regarding content, Unity has the largest database of tutorials, assets, guides and resources than any other game engine. As seen in the picture 2.4 below, this content is split into fifteen categories, and includes virtually anything a Unity developer will need to know about the engine. More tutorials exist on youtube, custom sites, and online teaching services such as Udemy.



Picture 2.5

Other segments in the learning tab, besides tutorials include:

The Documentation, which includes a manual that helps developers learn how to use the Unity Editor and its associated services [17].

Knowledge base, which provides information on anything associated with

Unity, from licenses to supported platforms among others.

Support, a service available to those with a Premium License [18] Live training, where live seminar style sessions are held by experts who take users through a project or a topic, or even a number of topics.

The last two categories include the resources, which we covered before, as well as the Certification service, which offers developers the chance to certify their knowledge.

2.2.3. Asset store

Unity's asset store is available through its main site, and hosts a lot of the content found online, both free and purchasable. Unity has an impressive asset store, with thousands of assets to be found. The biggest category is 3D models, currently counting over 15 thousand models. Other categories are significantly lower in numbers, the second biggest category counting 4 thousand assets. Unity also has the biggest community of all game engines. For example, the Unity facebook group has about 25 thousand members, while the Unreal Engine group has 22 thousand. While this isn't such a big deviation, one that could convince a young developer to choose one engine over the other, the gap between the two seems to close, as there seems to be a shift to Unreal since the release of Unreal engine 4.

2.3. CryEngine

CryEngine is the third most popular game engine. While it does have quite a few perks and advantages of its own, there are quite a few disadvantages that prevent developers, especially indie developers, from using it. To name a few, CryEngine's tools are a bit outdated, despite the fact that there are more than enough for every need, there are lots of performance issues compared to the other engines, and lacks proper support. On the other hand, CryEngine has one of the best graphic engines in the industry, equal only to Unreal Engine 4's current engine.

Regarding new developers, it is questionable whether CryEngine has enough tutorials for complete beginners, or whether a developer needs to be familiar with game engines.

In the next Chapter, we will compare Unreal Engine and Unity, point out where

each engine is better, or preferable, and give an idea of the application designed.

Chapter 3
Unreal vs Unity

In the previous chapter we briefly presented the most popular game engines in the industry, Unity and Unreal. In this chapter, we will compare these two engines, point out each ones' strong and weak points, and finally decide which one to use for our application. To do that, we will compare some of their characteristics, as well as their potential and performance in specific areas. The engine characteristics include features unrelated to the development itself. A few examples of such features include the license of each engine, the ease of use, number of resources and tutorials found, as well as the size and price of each asset store. Performance has to do with the effectiveness and capabilities of each engine, such as the graphic level, the availability of tools, etc.

3.1.1. License and pricing

One of the most important issues a game developer has to think about, is the license of the engine they use. Especially for indie or new developers, this may become the most important issue, or even an endgame if the license requires payment upfront. Luckily, both Unity and Unreal have a way to deal with this problem, each with its own method. Unity, as we said before, comes with a free to use license, and includes a royalty fee of 5% for games with n income over than 3000$ per calendar quarter. This means that a new developer does not have to consider expenses before even staring to develop a game or animated movie, and will have to pay a fee only when (and if), their game becomes quite successful. However, this may not be worthwhile for big companies that make thousands of dollars, as a 5% fee may cost more than Unity's, or any other engine's for that matter, license.

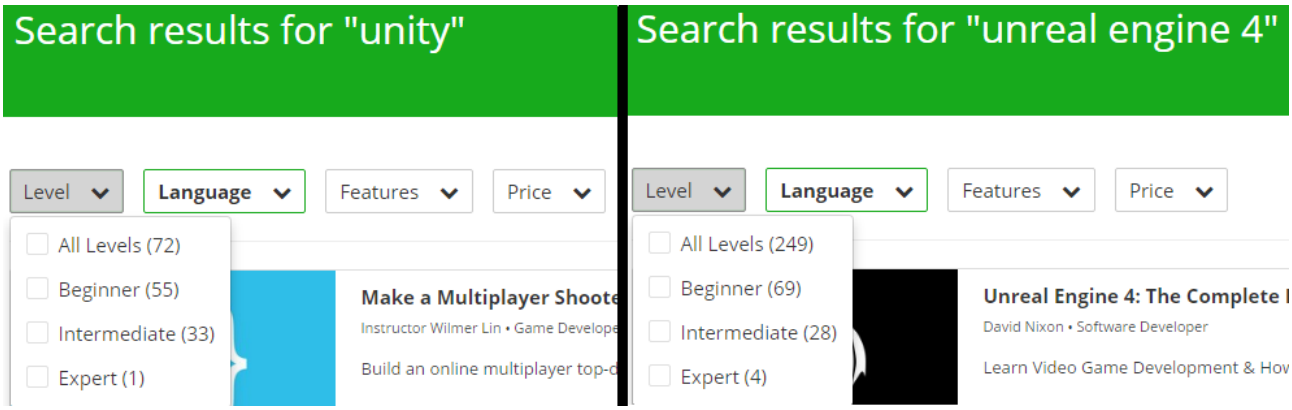
On the other hand, Unity has four licenses as mentioned in the previous chapter. One of these licenses is free, aiming to be used by junior developers, who will in turn expand their membership to plus, pro or enterprise, should they have additional requirements. While this is definitely a good way for a developer to

enter the business, there are a couple of problems that emerge. The first one is that while the free version covers the majority of the needs a junior developer might have as it includes the full version of the engine, it doesn't include some excellent tools like beta access, game performance reporting, customizable splash screens and more. These are included in the professional edition, which requires a monthly fee. Second and more important, even small or middle-sized companies may find the license to be expensive, as the cost (35-125) is per seat, meaning the license is charged for every developer working for the company.

Therefore, regarding licensing and pricing, we have come to conclude that for game developers who wish to have an engine that gives its full potential from the get-go, and has a fair pricing which depends on the game's success, Unreal is the best choice.

3.1.2.Learning

As stated before, both engines have impressive and active communities with lots of developers and professors creating tutorials, courses and other content. Moreover, both have great documentation, step by step tutorials and wikis. While Unity seems to have a larger amount of content online, the gap between the two seems to lessen after the release of the fourth edition of Unreal Engine. In some platforms, Unreal engine seems to have more content, as seen in the picture below, taken from the website Udemy, which offers online courses on a variety of subjects, covering every scientific and business sector. Udemy offers more than 40 thousand courses, and as of now offers 161 courses for Unity and 350 courses for Unreal. In the picture below, the search results are shown, as well as the level of the courses offered.



Picture 3.1

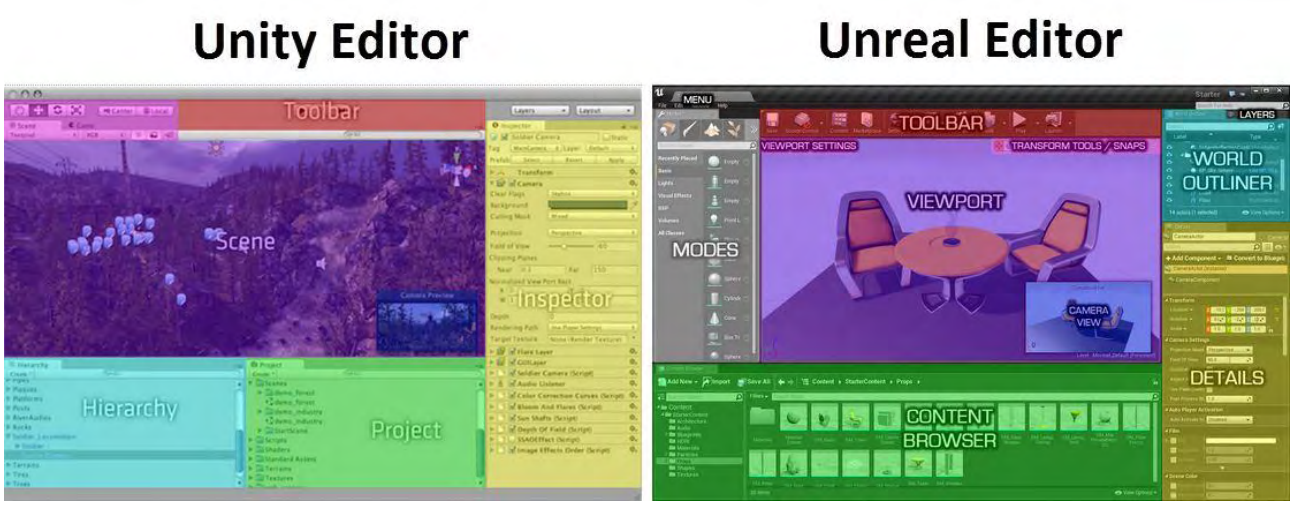
Regarding which engine is easier, I found it difficult to find a definitive answer. Both engines use programming languages with lots of tutorials, as well as have a plethora of tutorials themselves. The small difference is virtually unimportant, since the official tutorials of each engine are more than enough. In both cases, as well as in every other game engine, some, if not a lot of difficulty is expected to be encountered, since game engines are some of the most complicated programs there are, with lots of functionality. With the help of tutorials, documentation, a structured search engine and a community to guide, a developer can most definitely master any of these two engines. To sum up, it really comes down to personal preference regarding this aspect.

3.1.3. Ease of Use

Unity has always been known for their easy to use interface where beginners can jump right in and start making games. Though Unreal Engine 4 was a major improvement, they still take second place behind Unity in the area of user experience.

Both interfaces are very similar, with toolbars and settings within resizable & movable windows. Unreal's user interface is still quite bloated and complex. Everything takes longer and is more complicated than it should be. Assets take a long time to import and save, and simple tasks require extra, unnecessary steps. Unity 3D is fast, and the interface is quick and responsive. It's so light that it can run on Windows XP (SP2), while UE4 requires at least Windows 7 64-bit. Though the final product can look nicer in Unreal than in Unity, getting there takes longer and much more effort, especially for beginners.

In the picture below, there is a screenshot of the two engines, showing their user interface.



Picture 3.2

Unity seems to be the winner here, since its user interface is clearly friendlier than Unreal's, especially for junior game developers.

3.1.4. Code

Essentially this comes down to the developer's personal preference. Unity has the advantage of choosing between many programming languages (Javascript and C#), while Unreal uses only C++ and blueprints. Blueprints is a type of visual scripting, and works with nodes. For developers who have no programming background and are not interested in delving into the world of programming, visual scripting with blueprints is the solution. There are many limitations to Blueprints' potential, so coding in C++ or C++ and blueprints is the best choice, both in terms of performance and flexibility.

3.1.4. Asset Store

Both Unity and Unreal come with their own marketplace, allowing for the buying (and selling) of different game assets like characters, props and even things like sounds and particle effects. However, Unity really comes out on top in terms of the size of their asset store. Offering everything from intuitive animation and rigging tools to GUI generators and motion capture software, Unity hosts more than 15 thousand assets. Unreal's marketplace is much newer, and thus smaller with more expensive assets. However it is growing rapidly and all assets are of high quality. Developers can also sell assets they created in

these stores, and both stores split the profit 70/30.

3.1.5. Supported platforms

Both Unity and Unreal have a wide range of supported platforms. Unity seems to be the winner here, as it supports 24 platforms, while Unreal supports 17. More specifically, Unreal supports Android, iOS, Arcade, Microsoft windows, Playstation 4, Playstation Vita, Wii U, Xbox One, HTML 5, Linux/Steam OS, OS X, Mac OS X, Oculus Rift, HTC Vive, Project Morpheus, Playstation VR, Samsung Gear VR[19]

Unity on the other hand supports iOS, Android, Windows Phone, Tizen, Windows Desktop, Mac, Linux/Steam OS, WebGL, Playstation 4, Playstation Vita, Xbox One, Xbox 360, Wii U, Nontendo 3DS, Oculus Rift, Google Cardboard, Steam VR, Playstation VR, Gear VR, Microsoft Hololens, Android TV, Samsung smart TV and tvOS [20].

It is clear that Unity has better support for platforms generally, while Unreal focuses on selective ones. However, this may not be a problem for most developers, as both Unity and Unreal support all the most popular platforms.

3.1.6. Type of game

Another aspect of our game engine selection is the type of game we want to create. While Unity and Unreal can both create excellent games of any kind, there are some sectors where each engine is best. When it comes to 2D games, or mobile games, this is where Unity dominates, with most popular indie mobile games created with it. As it has become a must for mobile developers, lots of 2D features have been developed that make the engine easy to use for mobile development, and have made it easy to start creating such games. Unreal is trying to push their engine that way too, but is clearly behind when it comes to performance. The main issue is with rendering. In Unity, there are two options for the rendering process. Per Object and per pixel, while in Unreal there is only the per pixel option. This limitation explains why Unreal requires more hardware than Unity's default setup, therefore making development for mobile devices fall behind. There is a guide online about methods on how to increase performance [21] but even this is currently under development. It would

suffice to say, that for 2D and mobile development, Unity really stands out.

On the other hand, when it comes to 3D and PC development, nothing can match Unreal's next gen graphic capabilities. Despite the new version of Unity, Unity 5, with lots of new features, Unreal is still ahead in nearly every area of graphics: terrain, particles, post processing effects, shadows & lighting and shaders.

In an article published on not-lonely.com, a site aimed for game developers, hosting lots of resources, developer Vitaly Okulov created the same asset for both Unity and Unreal Engine, to show the difference in graphics. A few pictures are shown below, and many more can be found in the references.

Picture 3.3

In the next two pictures, we will show the same picture rendered in Unreal and Unity:

Picture 3.4

Picture 3.5

We can see that while both pictures look great, it is clear Unreal is one step ahead. [22][23]

3.2. Our Choice

After seeing each engine's characteristics and capabilities, we can assume one thing. That there is no perfect game engine, and the selection of each one depends solely on the developer's needs and preferences. While this choice is a tough one, as both engines are capable enough, and each has its own strengths and weaknesses, I chose to work on Unreal Engine. The most important reason is that Unreal Engine is one of the oldest engines in the field, yet Unreal Engine 4 is a fairly new edition, which explains many of its disadvantages, like the less content compared to Unity. This means that Unreal has a lot of potential to further develop and fix whatever weaknesses it has. What led me to this decision was mostly the graphic capabilities of Unreal. While the game developed in this project does not focus on graphics but on coding, a big part of the gaming (and animated) industry relies on the quality of graphics. Especially in the future, when virtual reality will be developed, graphics will play a more important role than ever. Moreover, C++ is a must-know lprogramming language, and since I have had some experience with Javascript, I took the chance to become familiar with C++ too.

The next chapter will be about the application, the concept of the game, what game mechanics it presents, as well as step by step instructions on the process of its development.

Chapter 4
Game Development

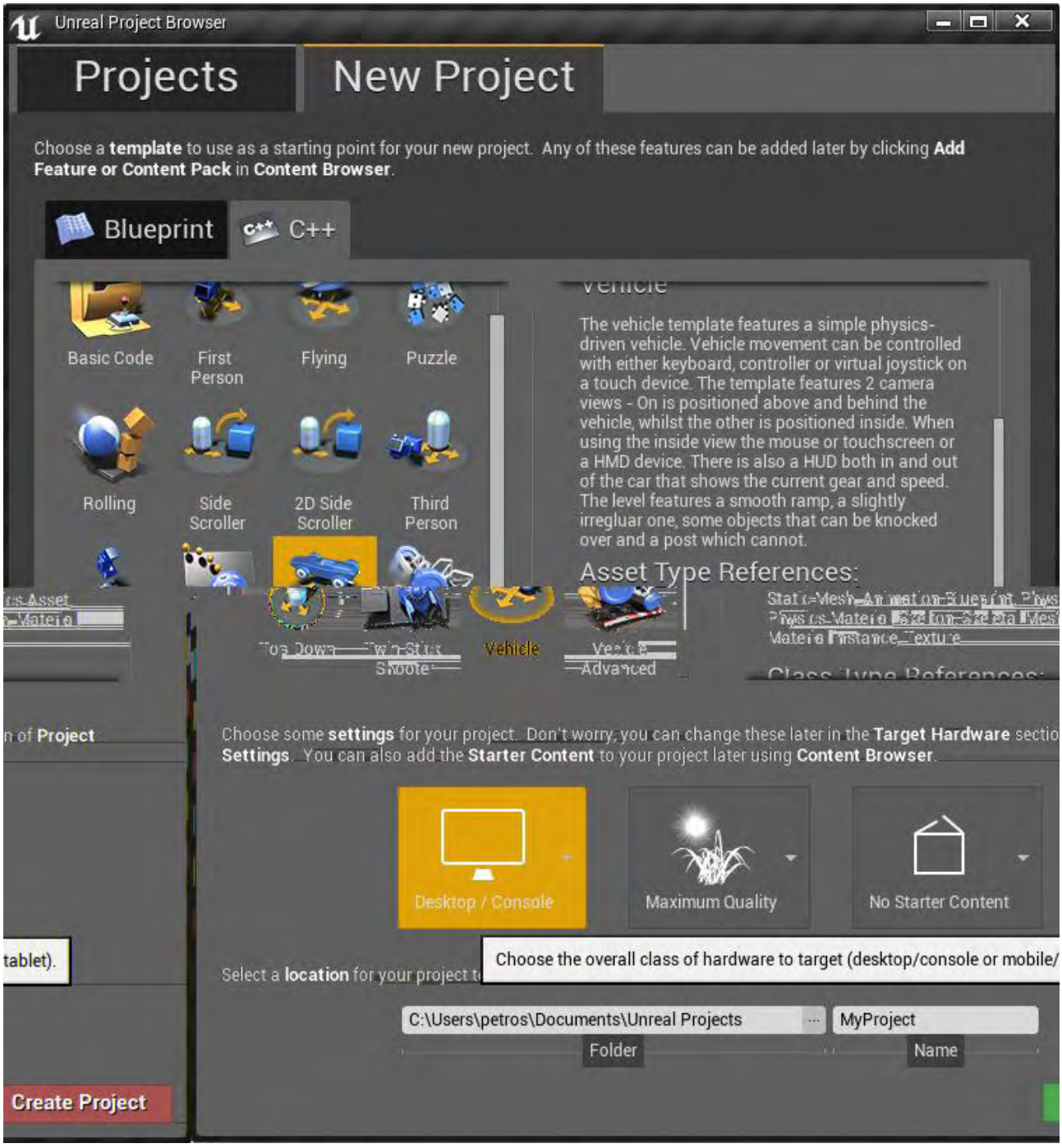
Our game will essentially be a simple game, with little to none graphic content, as graphics is a completely different job in high-end games. In this game, we will focus on the coding part, mostly with C++, which, as we said before, is the most efficient option too. However, we will work with blueprints too, in collaboration with C++, for best performance.

4.1. Creating the base level

To get started, we open the Epic Games launcher. From inside the launcher, we then launch Unreal Engine. Note that the launcher includes a few more features, mainly video games made by epic, or tools to create mods for some of these games. Once the Engine has launched, we select the new project tab, and a new screen appears.

On this screen, there are quite a few more options. Let's take a look at them one by one. Right below the new project tabs, there are two few sub-tabs, one called Blueprint and the other C++. These describe the kind of project to be created. Ours will be a C++ project. Inside this tab, there are twelve templates to choose from. While these templates will create a starting project, everything can be modified later once the project has been created. Our application will be a vehicle game. Other templates, as seen in the picture below, include first person game template, third person, side scroller, flying, puzzle, and basic. Below are some more settings for the project, which help improve game performance. The first choice to make is to choose whether the project is targeted towards Desktop and consoles, or for mobile and tablet devices. This is important to figure out the settings the game should have to be able to run efficiently in the targeted devices, without any problems. For example, in mobile devices the engine “will know” to drop graphic quality in exchange for less hardware consumption, while in consoles it will do the opposite. On the right is the next choice, where we choose the overall quality of the game. Furthermore, choosing tablet or mobile will also add virtual controls on the screen, which we will see later. Regardless of what we chose in the previous menu, here for example we may choose to make the game be able to run in older computers in exchange for

some drop in quality, and vice versa. Lastly, we can choose whether to have starter content or not. While development is easier with starter content, Choosing no starter content will decrease the size of the game, and the developer can later choose which of this content is required and add it manually. Finally, we select a path and name for our project, and press Create Project on the bottom right of the window. Below is a picture of this first screen where we created the application.



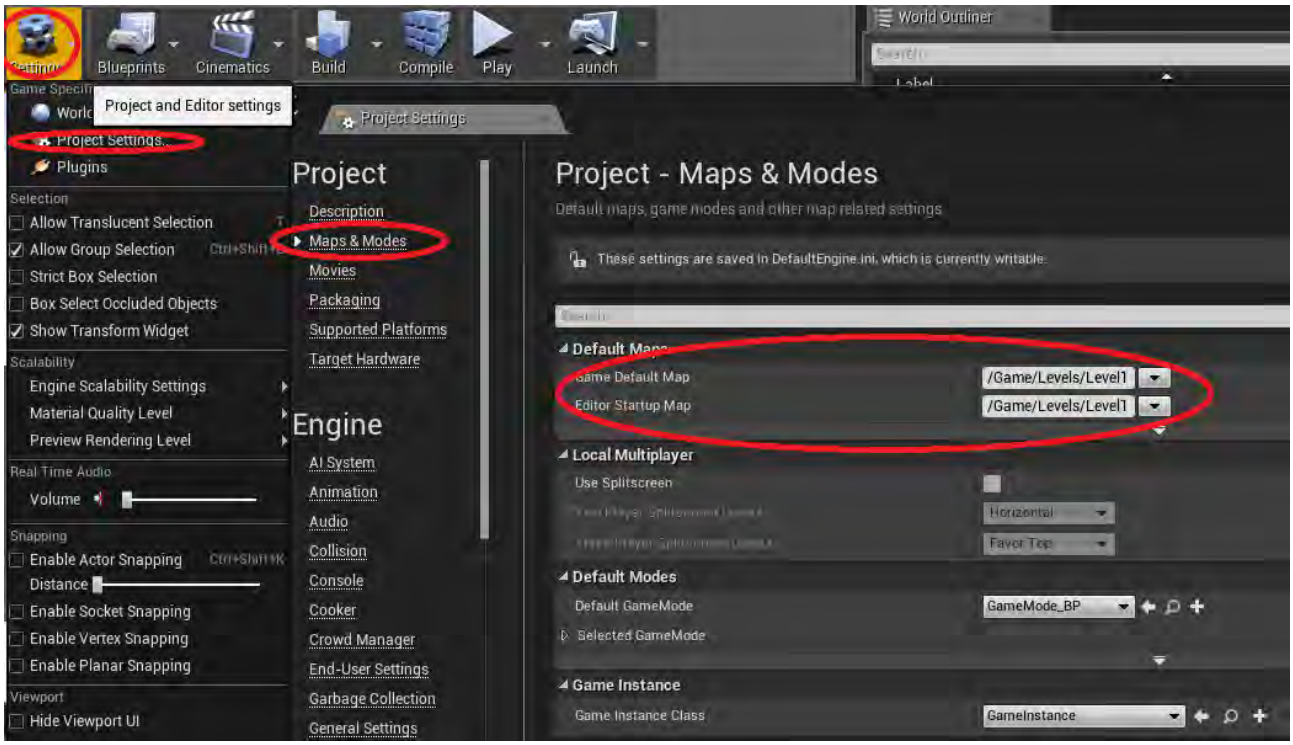
Picture 4.1

Before we continue, we should note that when working on C++, Unreal Engine requires a third party program to work on the code. In this project, as recommended by many sources, we used Visual Studio.
In the game created, we chose the mobile/tablet hardware to target. Despite this, we can export the game for windows as well, since as said before everything in the game is customizable.

Building the game can take some time, depending the hardware we are using. Once it is complete, the project opens. In the level window, as seen below, we can see that a sample vehicle already exists, as well as a default level. The vehicle has all the basic functions, like movement, collision etc.

Picture 4.2

Since we don't need the terrain to show the functionality we want, we create a new level, set it to default, and then delete this one. To do that, we go to file → New Level → Basic. This level is very similar to the previous one, but missing the terrain. Next, we want to set the new level to the default level, since a projet can (and most likely will) have many levels. So we go to Settings→ Project Settings → Maps and Modes → and we see a section called Default Maps. Here we can change the game default map, which is the first map to load when the game begins, and the editor startup map. We set these two variables to the new level. A better description is shown in the picture below:



Picture 4.3

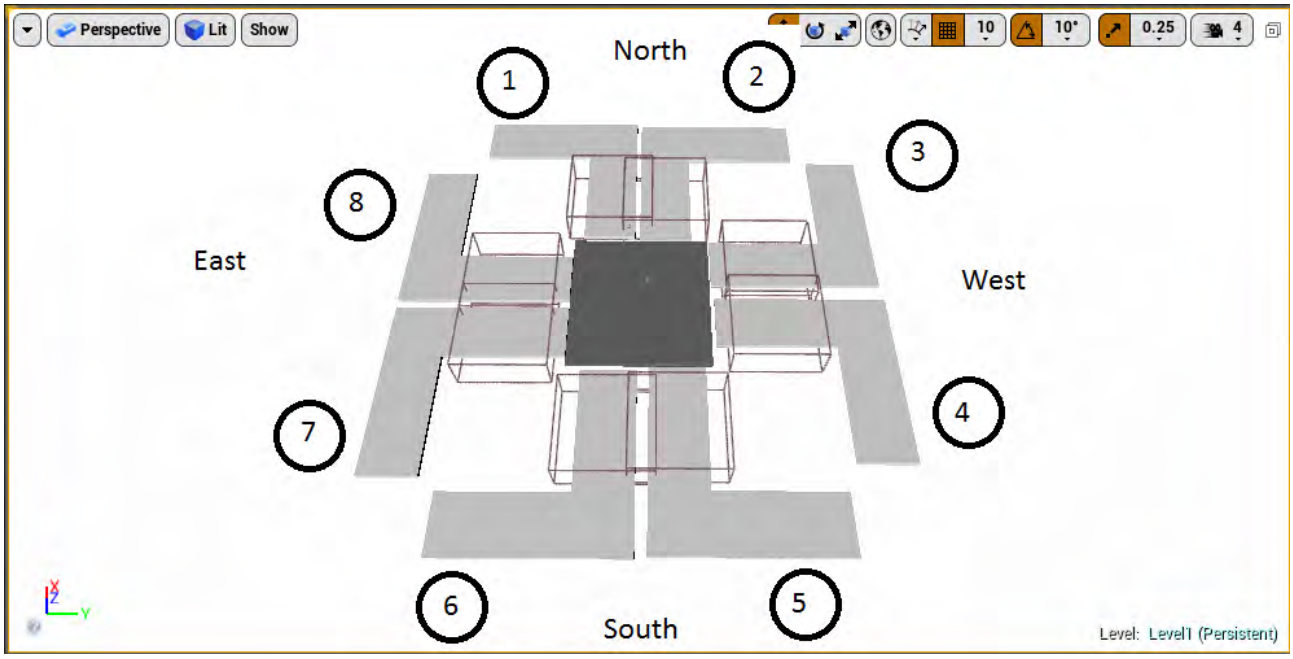
4.2. Creating the Road_Segments

The game will include a road that will spawn as the vehicle moves on top of it. Each road segment will have a random rotation, so that the road will be completely random and differ each time we run the game. A score will be increased as the player keeps moving, and when the player reaches a specific score, they win. On the other hand, if the player falls off the road, they lose.

4.2.1. Concept and ideas

Let's start with the road. First of all, we need to explain what the road will be, and what each “thing” is here. Generally, everything that has some functionality is a class. More specifically, whatever has a physical representation to our level is called an actor. The Actor class is already implemented, and when we create something new to be physically represented, we make it a child class of the Actor class. So now, we on the top left corner of the content browser, we select the Add New button, and from the drop down menu we select C++ class. In the window that appears, we select Actor as the parent class and name our class Road_Segment.

Now while this is a class, it doesn't have a physical representation yet on the level. But we will get there. Before we do that, we have to do some thinking about how to implement this. First, we will create eight subclasses, children of Road_Segment. Each subclass will represent a turn, as seen in the picture below.



Picture 4.4

As seen above, we will have eight “turns”, each one represented as a road segment. This picture is taken after the segments were created, yet this was the original idea from the beginning. When the game begins, the vehicle is on the central square, facing north. Therefore, the road direction we should choose is either 1 or 2, meaning the next segment to spawn will be either Road_Segment1 or Road_Segment2, randomly chosen. For each segment, we can deduce which two segments will be the next to spawn, depending on the direction it faces. For example, Segment 1 faces east, so the next segment will either be 7 or 8. For road segment 2, which faces west, it will either be 3 or 4, and so on. Briefly, for road segments 1 and 6, the next will will either be 7 or 8, for segments 2 and 5, 3 or 4, for segments 3 and 8, 1 or 2, for segment 4 and 7, 5 or 6. One of the problems that may occur now is overlapping, meaning that if a circle occurs, one road segment will spawn over the old one. To prevent that, we need to destroy the old segments after some time. Also, we need a trigger, an event to signal the creation (Spawning) of the next segment. Finally, we need to have an endgame to our game, either win or loss. Therefore, we need to keep a score,

and create some play states depending on the score and other parameters. To sum up, until now we have some tasks to be done. These tasks include:

- Create the physical representation of the road segments.
- Create an event that will trigger the spawning of the next segments
- Spawn the segments
- Have a variable to keep score, which will increase each time the vehicle enters a new segment
- Create some game states, for the win/lose/playing game state.
- Display the current score and a custom message informing the player about win or loss situation

4.2.2. Creating the Road

Having made the class Road_Segment above, we already have a class that all different segments will inherit from. Road_Segment, like every other class, has two files, the header file Road_Segment.h and the source file Road_Segment.cpp.
Each class has 3 functions by default. One is the constructor, here it is called ARoad_Segment(). This function is used to initialize variables. Second is the Begin_Play() function, which is called only once when the component is created. In the component preexists, it is called in the beginning of the game. Last is the Tick(deltaseconds) function, which is called every (deltaseconds) seconds. If this is not initialized, the function will be called every frame, which may me something we might want, but in many cases it may cause performance problems, even get the whole game stuck. Thee functions exist in every class we will be using in this project. On the top of the header file, we include other header files we will need to access throughout the development.

Also, besides C++ code, we will be using Blueprints, which is a visual scripting language, in collaboration with C++ for best performance.

The code we write for the road segments will be both in the Road_Segment class, and its child classes. Since most features are going to be common among the segments, lots of variables and code will be in the Road_Segment class. Below is a picture of the code in the Road_Segment class, and the code of one

of its children classes will follow to fully explain the functionality.

```
// Sets default values
ARoad_Segment::ARoad_Segment()
{
    // Set this actor to call Tick() every frame.  You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;
    RoadMesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("RoadMesh"));
    RootComponent = RoadMesh;
    CreationBox = CreateDefaultSubobject<UBoxComponent>(TEXT("CreationBox"));
    CreationBox->AttachTo(RootComponent);
    CreationBox->SetBoxExtent(FVector(1000,700,1000),false);
    haspassed = false;
}
```

Picture 4.5

This function is called at the beginning and used to set the default values as well as define certain things. Let's see what each line does:
`RoadMesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("RoadMesh"));`

This line creates a subobject of the type mesh. A mesh is actually the 3D model used to represent an actor. Here we define this variable, which will be linked to a model through the blueprints.

Secondly, we create an invisible box, attach it to the enter of the RootComponent, and give its dimensions. The box will be used later to check when the vehicle has overlapped (passed inside it). Note that physics and collison will be turned on for the roadmesh, so that the vehicle can stand on it and not fall through, while the creation box doesn't have physics turned on, allowing the vehicle to pass through it (overlap).

The next function as we said before is the BeginPlay function. It is called once as we said when the game starts or when the actor spawns.

Haspassed is a boolean variable used to enter the “lose” game state. We will see it later.

```
// Called when the game starts or when spawned
void ARoad_Segment::BeginPlay()
{
    Super::BeginPlay();
    SegmentIsActive = 1;
    UWorld* const World = GetWorld();
    World->GetTimerManager().SetTimer(DestroyHandle, this, &ARoad_Segment::Destroy_Segment, 10.0f);
}
```

Picture 4.6

The `Super::BeginPlay();` line executes the father's begin play function. As we said before, Road Segment inherits from Actor class.

Next we have a variable, which activates and deactivates the segment. This happens to solve a problem appearing later on. To put it simply, we only want the code inside Tick() to only happen once, and without this variable it happened for as many frames as the vehicle overlapped the creation box. The next two lines create a timer that destroys the class after 10 seconds. GetWorld() is a function that returns the a pointer to the game world. The next line creates a timer that after 10 seconds calls the function Destroy_Segment. DestroyHandle is a name for the timer to distinguish it, and this means that the Destroy_Segment function will be called for this specific instance of Road_Segment. In other words, every road Segment of any type created will be destroyed in 10 seconds of its creation.

```
// Called every frame
void ARoad_Segment::Tick( float DeltaTime )
{
    Super::Tick( DeltaTime );
}


void ARoad_Segment::Destroy_Segment() {
    if (haspassed == false) {
        ACrazywheelsGameMode* Mode = (ACrazywheelsGameMode*)GetWorld()->GetAuthGameMode();
        TestMode = Cast<ACrazywheelsGameMode>(Mode);
        if (TestMode) {
                TestMode->SetCurrentState(ECrazyWheelsPlayState::EGameOver);
        }
    }
    Destroy();
}
```

The last two functions are Tick() and Destroy_Segment() functions.

Picture 4.7

As we can see, the Tick function is empty as each kind of Road_Segment has different, although similar functionality.

Destroy_Segment though does the same things for all segment types. The thought behind it is simple. If it is my time to be destroyed, and the vehicle hasn't passed through here yet, then it fell off the road. Therefore the player loses. We will explain this further later.

Regarding the header file, as we said there is a number of classes included since we need to use them later on. Also, when we declare functions and variables, the macros UFUNCTION and UPROPERTY are used, to declare how these variables and functions are handled by the blueprints. A full explanation of the arguments ufunction and uproperty take can be found here [24].

Moving on to the segment types, they all have similar functionality, so we will examine Road_Segment1 and point out differences where they are.

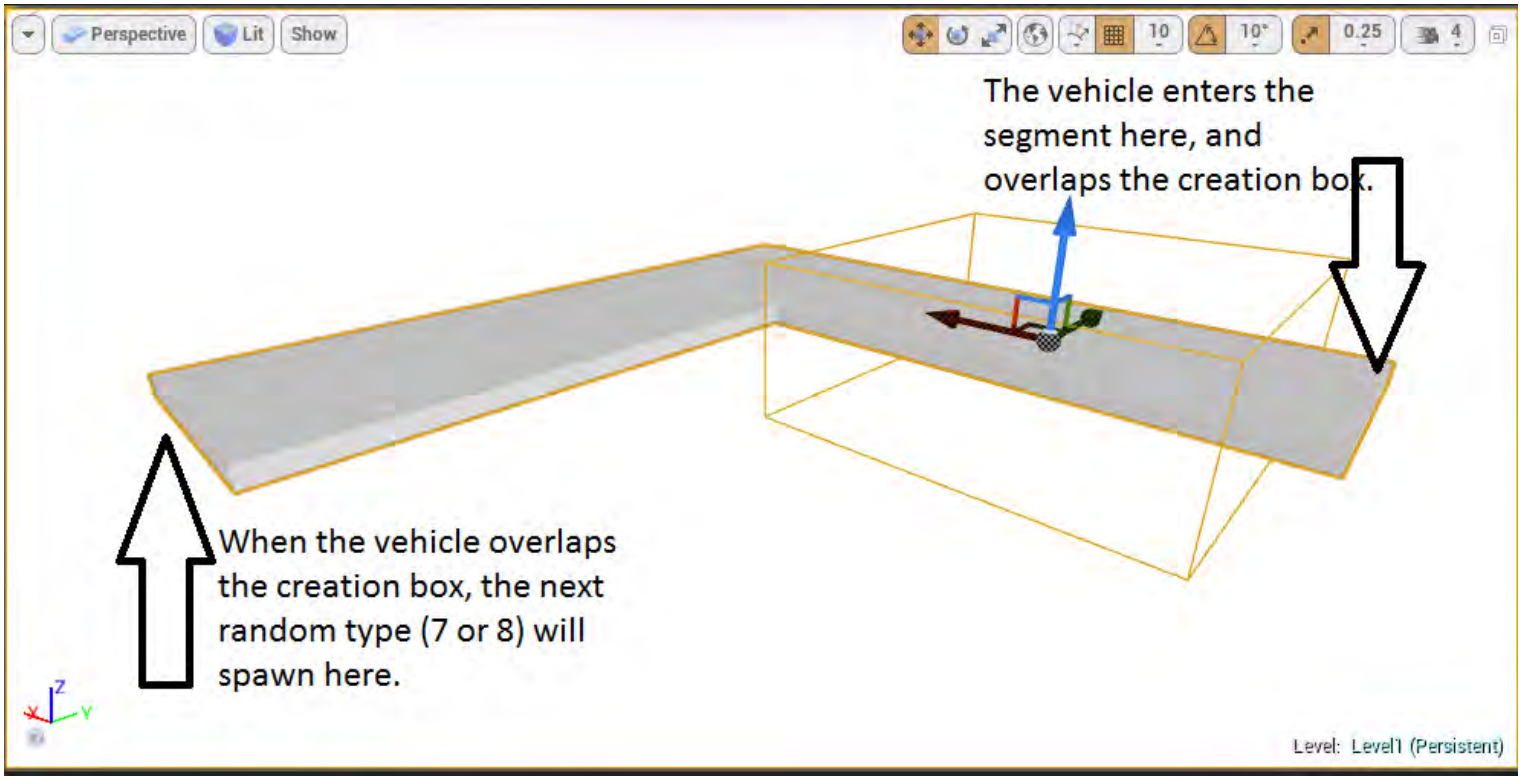
In these classes, most of the functionality is included in the Tick() function, which is presented below

```
void ARoad_Segment1::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);
    if (SegmentIsActive == 1) {
        //get all overlapping actors
        TArray<AActor*> CollectedActors;
        CreationBox->GetOverlappingActors(CollectedActors);
        //for each actor collected
        for (int32 iCollected = 0; iCollected < CollectedActors.Num(); ++iCollected) {
            //cast the actor to Pawn
            TestVehicle = Cast<ACrazywheelsPawn>(CollectedActors[iCollected]);
        }
        if (TestVehicle) {
            haspassed = true;
            UWorld* const World = GetWorld();
            FRotator SpawnRotation;
            SpawnRotation.Yaw = 0.0f;
            SpawnRotation.Pitch = 0.0f;
            SpawnRotation.Roll = 0.0f;
            FVector SpawnLocation = GetActorLocation();
            SpawnLocation[0] = SpawnLocation[0] + 950;
            SpawnLocation[1] = SpawnLocation[1] - 2250;
            Road_Choice = rand() % 2 + 1;
            if (Road_Choice == 1) {
                ARoad_Segment7* const Next_Segment = World->SpawnActor<ARoad_Segment7>(Spawn_Road7, SpawnLocation, SpawnRotation);
            }
            else if (Road_Choice == 2) {
                ARoad_Segment8* const Next_Segment = World->SpawnActor<ARoad_Segment8>(Spawn_Road8, SpawnLocation, SpawnRotation);
            }
            SegmentIsActive = 0;
            ACrazywheelsGameMode* Mode = (ACrazywheelsGameMode*)GetWorld()->GetAuthGameMode();
            TestMode = Cast<ACrazywheelsGameMode>(Mode);
            if (TestMode) {
                TestMode->IncreaseCurrentScore();
                if (TestMode->GetCurrentScore() >= TestMode->GetScoreToWin()) {
                    TestMode->SetCurrentState(ECrazyWheelsPlayState::EWon);
                }
            }
        }
    }
}
```

Picture 4.8

The first thing encountered in this function is an if statement, and all the other code is included inside it. We use this variable SegmentIsActive because we only want this code to run only once.

Next thing we want is to get all actors overlapping with Creation Box. Let's say we drive the vehicle, and it enters the segment as seen in the below picture:



Picture 4.9

Returning to the code, CollectedActors is an array of actor components. What the next line (`CreationBox→GetOverlappingActors(CollectedActors);`) does, is get all actors overlapping with the creationBox and putting them in the array.

At this point, let's mention that the purpose of the creationbox is to simply trigger an event when the vehicle overlaps it. So, when the vehicle enters the creationbox, the next segment will spawn, the score will increase by 1, etc. Before we get to that, we want to make sure that the vehicle has overlapped the creationbox. For example, let's say that another road segment overlaps the box, we wouldn't like to spawn the next segment. So, what we do here, after we collect all overlapping actors in the array, is cast each actor into CrazyWheelsPawn class (commonly referred as Pawn class, since Unity names it as “ProjectnamePawn). The pawn actor the base class of all actors controlled by the player [25]. So any child of it can be cast into Pawn. The result is saved into a TestVehicle variable. If the cast is unsuccessful, meaning the actor

overlapped is not player controlled, then TestVehicle = 0. Since the tick function is called every frame (or more sparsely if we choose so, here we did for performance reasons), it will search for overlapping actors until it finds one, or until the 10 seconds pass and it gets destroyed.

If the cast is successful, this means the vehicle has entered the creation box, and we want for some things to happen.

We set haspassed = true. We do this, because as mentioned before, and as seen in picture 4.8, haspassed "informs" if the vehicle has reached the creation box. If not, the pay state changes to Gameover.
The next numbers are for spawning the next segment. This is achieved with the SpawnActor() function [26]. The spawn actor command takes three arguments. The first one is the type of actor to spawn. Generally, we need to spawn the blueprint classes for each road segment, since the C++ class has no mesh to represent it on the field, nor physics. Creating the blueprint class will be explained later, but here, since C++ does not recognise Blueprint classes, we have declared it as a subclass (child class) of the classes Road_Segment 7 and Road_Segment8 (since blueprint classes do inherit from the c++ ones):

```
UPROPERTY(EditAnywhere, Category = "Spawning")
TSubclassOf<class ARoad_Segment7> Spawn_Road7;

UPROPERTY(EditAnywhere, Category = "Spawning")
TSubclassOf<class ARoad_Segment8> Spawn_Road8;
```

Picture 4.10

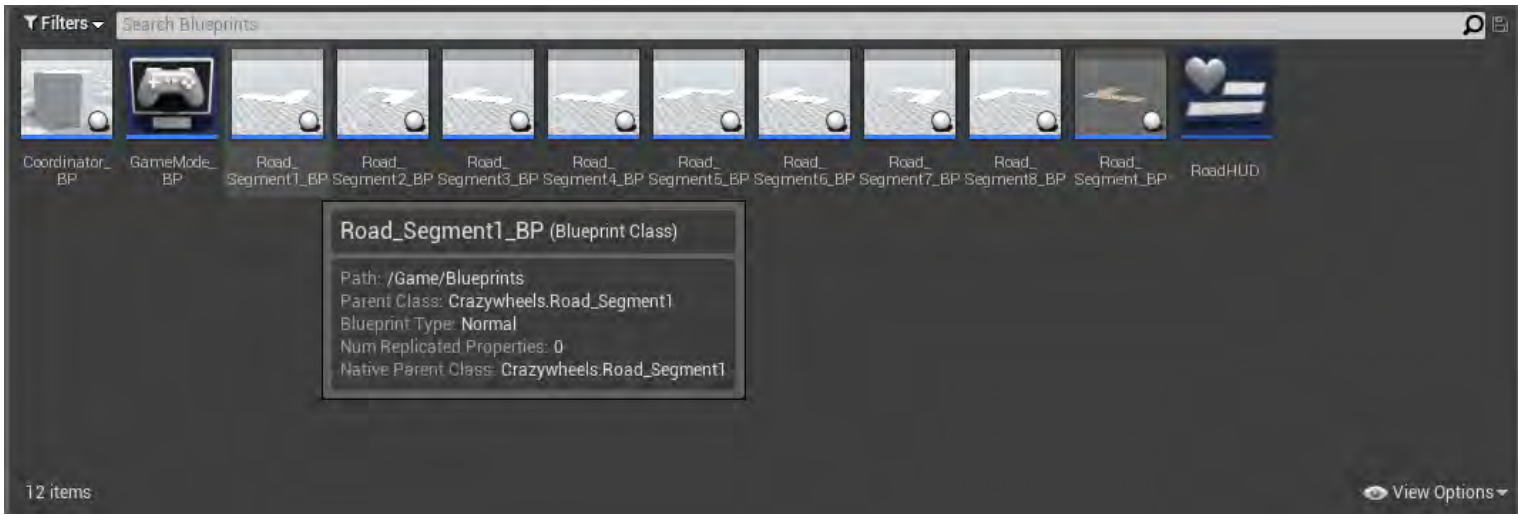
For each other type of segment, we choose the next ones as described in picture 4.5. Finally, we create a random value of 1 and 2, and choose the type of the next segment based on this. Before we spawn though, we need to intialize the other two arguments. The second argument is SpawnLocation, and its value tells us where to spawn the new actor. As stated in Picture 4.10, we want to spawn on the other end of the current segment. Therefore, we take the location of our current actor with the command GetActorLocation(), save it into an array, and then add to the array the offset to match the new actor on the other end. This offset was calculated by putting two actors side by side, and subtracting their locations. Of course this offset differs for each type of segment, but for the

same segment, it doesn't matter what our next choice will be. For example, if our current segment is Segment1, the offset is 950 pixels in the x axis, and -2250 in the Y axis, regardless of whether our next segment is of type 7 or 8. Lastly, we set the SpawnRotation to [0,0,0] since we dont want our actor to be rotated. After this, we set SegmentIsActive to 0, since in the next frame the vehicle and the creation box will still overlap, and we don't want to do this procedure again. Lastly, we increase the current score by one, and check whether we have reached the maximum score. If so, we change the game state to Ewon. We will talk about the game states later.

Now while the C++ code is ready, we still don't have a physical representation for each segment. As we said, to do that we need a blueprint class. We go in the content Browser, and from the Add new option, we select Blueprint Class. In the options menu that popped up, we select the parent class, as seen below:

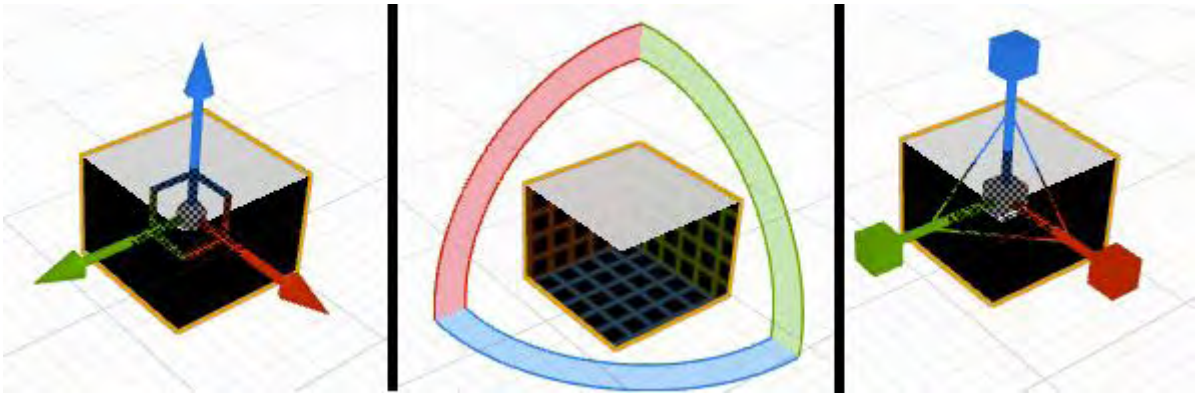
Picture 4.11

We search for our Road_Segment types in the search bar, and create one blueprint class for each segment type. Once this is done, the blueprint folder in the content browser will look somewhat like this:



Picture 4.12

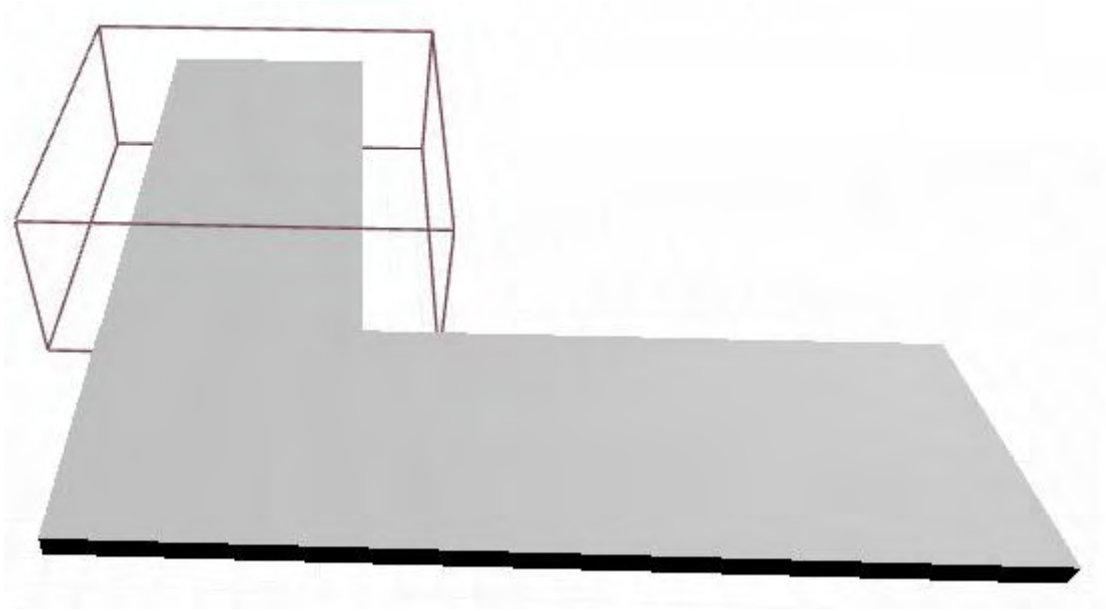
At this point, we need the static (because it won't have movement) mesh to represent each segment. To create this, we go to the modes tab on the left, drag two cube inside our level, and modify them to fit the dimensions we need. When we select a static mesh, by pressing R we can move it across the three axis, with E we rotate it, and with R we scale it.



Move (W) Rotate(E) Scale(R)

Picture 4.13

After we bring the cubes to the dimensions we want, we merge them. To do this, we go to Window on the upper menu → Developer Tools → Miscellaneous and we select merge Actors. Here we should note that the center of the new mesh is the center of the first old mesh selected. After we have merged the Cubes, the mesh that will represent all our Segments is created. It looks like this (besides the “invisible” box”):



Picture 4.14

All Road Segments use a rotated version of this mesh.

Returning to the blueprints, after we double click on Road_Segment1_BP (we named it that way) the Class Defaults tab appears. Here we can see the class data and edit their values (if allowed). Some of these data are the ones we wrote ourselves in C++, and they are visible here because the Blueprint class inherits from the C++ class. Here, lets remember the UPROPERTY macro we talked about before. This is useful here, as it determines the kind of access blueprints has on C++ variables. For example, if a variable is BlueprintReadOnly, Blueprints can't edit its value, while if it is BlueprintReadWrite, it can. A full

documentation of the arguments these macros take was provided before. Functions are not displayed here, but in the full blueprint editor, which we will access later.

To continue with the blueprint, some of the data presented is shown below:

Picture 4.15

These categories are the ones we are interested in for now. As we can see, there is another category named materials, but we won't work with it. From the static mesh dropdown menu, as seen above, we choose the static mesh to represent the road segment 1 on the level. This is inherited from the actor class, since as we said before our Road_Segment class inherited from it. In the spawning category, which we created here:

```
26      UPROPERTY(EditAnywhere, Category = "Spawning")
27      TSubclassOf<class ARoad_Segment7> Spawn_Road7;
28
29      UPROPERTY(EditAnywhere, Category = "Spawning")
30      TSubclassOf<class ARoad_Segment8> Spawn_Road8;
```

Picture 4.16

The category argument is the one that shows where this property will appear in the blueprints. At first, the value is set to none, as seen in the Spawn Road 7 above. Since Spawn_Road7 is a Subclass of Road_Segment7, the dropdown menu allows us to choose what to spawn between all of Road_Segment7's subclasses. We choose the blueprint class, as seen in Spawn_Road8. The same procedure is repeated for segments 2-8, with each segment spawning its respective next segments, and getting its own mesh.

Next is the actor Tick. The second value, Tick Interval, determines when the Tick function will be called. Its initial value is 0, which means Tick will be called every frame. This is unnecessary, and reduces performance (as in every Tick we check for overlapping and casting, which are “expensive” actions) so we changed this to 0.3, which will call the function a lot less times (a game may reach 60 frames per second, and even more) but isn't a big enough time space so that the vehicle passes through the creation box without triggering it.

The last is the shape of the box, and here we can override the values put in the C++ code, in the Road_Segment class.

Last but not least, in the same way we did the above, we create another class (named it coordinator) which will spawn the first road segment, and get destroyed within 10 seconds as well.

4.3. Score and Play States

4.3.1. Keeping a score

Another thing we need for the game, is to win or lose. What determines this is entirely up to the developer, so the concept behind this is that the player will win when the vehicle has “traveled a lot on the road”, and lose when the vehicle falls from the road.

For the score, we need some variables and functions:

- CurrentScore, to keep the current score obviously
- ScoreToWin, a variable to keep the score needed to win the game
- A float function to return the CurrentScore, GetCurrentScore()
- A float function to return the ScoreToWin, GetScoreToWin()
- A void function to increase the score, IncreaseCurrentScore()

As seen in the code of the class Road_Segment1, the score will increase when the vehicle enters a new segment. So, besides creating a new road segment as mentioned, the score will be increased there too. So, in the Road_Segment class, we include CrazyWheelsGameMode.h in order to have access to it. To increase the score, we implement the following code:

```
ACrazywheelsGameMode* Mode = (ACrazywheelsGameMode*)GetWorld()->GetAuthGameMode();
TestMode = Cast<ACrazywheelsGameMode>(Mode);
```

```
if (TestMode) {
	TestMode->IncreaseCurrentScore();
}
```
First we get the game mode of the game, then we cast it to CrazyWheelsGameMode (this helps prevent lots of errors), and if the cast is successful, we call the function IncreaseCurrentScore().

4.3.2. Implementing PlayStates

Play states will determine whether the player has won, lost, or paying the game. As it is a key element of the game, it will be implemented in the GameMode class.

The play states will have the form of an ENUM variable. In the CrazyWheelsGameMode.h file, we introduce the variable:

```
enumclassECrazyWheelsPlayState {
	EPlaying,
	EGameOver,
	EWon,
	EUnknown
};
```
Eplaying is the state at which the player hasn't won or lost yet, EGameOver is the state at which the player has lost, and EWon is the state at which the player has won the game. EUnknown is not used, we implemented it to have a default value in case something goes wrong.

The decision about when someone wins the game is fairly simple. When the CurrentScore increases, we check if it is equal or greater to the ScoreToWin. If so, we set the game State to EWon. The code that implements this follows:

```
if (TestMode) {
	TestMode->IncreaseCurrentScore();
	if (TestMode->GetCurrentScore() >= TestMode->GetScoreToWin()) {
		TestMode->SetCurrentState(ECrazyWheelsPlayState::EWon);
	}
}
```
Note that some of this code we showed on the previous page.

The way to determine when the player loses is a bit trickier. We said that the player loses when the vehicle falls off the road.

Picture 4.17

But how can we know that the vehicle fell off the road. One way to do that is to create another invisible box under each segment, and when that box overlapped with the vehicle, we would know that the vehicle fell. However, this would be too expensive in terms of resources, as overlapping and casting are very costly procedures, let alone when cast lots of times within a second. So, another thought is to implement the haspassed variable. This variable is set to false when the segment spawns, and is set to true when the vehicle overlaps with the creationbox. Also, we know that after 10 seconds, every road segment gets destroyed. Therefore, if the segment is about to be destroyed and haspassed is still false, this can only mean one thing. That the vehicle never overlapped with the creationbox, which means it fell off the previous road segment. So, inside the Destroy_Segment function of Road_Segment (the parent of all road Segments 1-8) just before we call the Destroy() command, we check if haspassed == false. If it is, we set the Play state to EGameOver. The code implementing this is shown in picture 4.8.

4.4. HUD

In this section of development, we will implement the HUD. HUD (head-up

display) is a method by which information is visually relayed as part of the game's user interface. Most usual examples are the character's health, items etc.

Within the engine, to display information on the HUD, we need to use UMG (Unreal Motion Graphics) [27]. Unreal Motion Graphics UI Designer (UMG) is a visual UI authoring tool which can be used to create UI elements such as in-game HUDs, menus or other interface related graphics you wish to present to your users. At the core of UMG are Widgets, which are a series of pre-made functions that can be used to construct your interface (things like buttons, checkboxes, sliders, progress bars, etc.). We will use a widget to display the score and a message. To do that, we need to change some dependencies in the project's build file. So we go to Visual Studio, in the solution explorer tab, and the file we are looking for is in the path Games → ProjectName → Source → ProjectName → ProjectName.Build.cs. Of course ProjectName is the name of each project, in our case it is CrazyWheels. Depending on the engine version and settings, this file may be different, so in order to have the full functionality, we need to add a few more dependencies. So, after opening the file, we see this line of code: `PublicDependencyModuleNames.AddRange(newstring[] { "Core", "CoreUObject", "Engine", "InputCore", "HeadMountedDisplay"});`

in which we add one more dependency `"UMG"`to enable it. However, UMG uses some Private Modules as well, so we add them right below it. These new modules are slate, and SlateCore: `PublicDependencyModuleNames.AddRange(newstring[] { "Slate", "SlateCore" });`

Also, for some templates it may be useful to change the project's header file as well. Since some projects don't use as many classes, they have `#include"EngineMinimal.h"`to increase performance. If this is the case, in order to use UMG, we have to change this to `#include"Engine.h"` and we are ready to use UMG

To use the UMG, we first need to declare it. So, we go to our Game Mode class, which is the class responsible for handling all the “rules” set in our game. Here, we create two variables, one for the UMG, and one for the specific instance of the UMG in our game. The code that does this is seen below:

```
//The widget to use for our HUD Screen
UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category = "Time", Meta = (BlueprintProtected = "true"))
TSubclassOf<class UUserWidget> HUDWidgetClass;


//the instance of the HUD
UPROPERTY()
class UUserWidget* CurrentWidget;
```

Picture 4.18

Then, we go to the source file and create the instance of the HUD. This happens in BeginPlay():

```
void ACrazywheelsGameMode::BeginPlay() {
    SetCurrentState(ECrazyWheelsPlayState::EPlaying);


    if (HUDWidgetClass != nullptr) {
        CurrentWidget = CreateWidget<UUserWidget>(GetWorld(), HUDWidgetClass);
        if (CurrentWidget != nullptr) {
            CurrentWidget->AddToViewport();
        }
    }
}
```

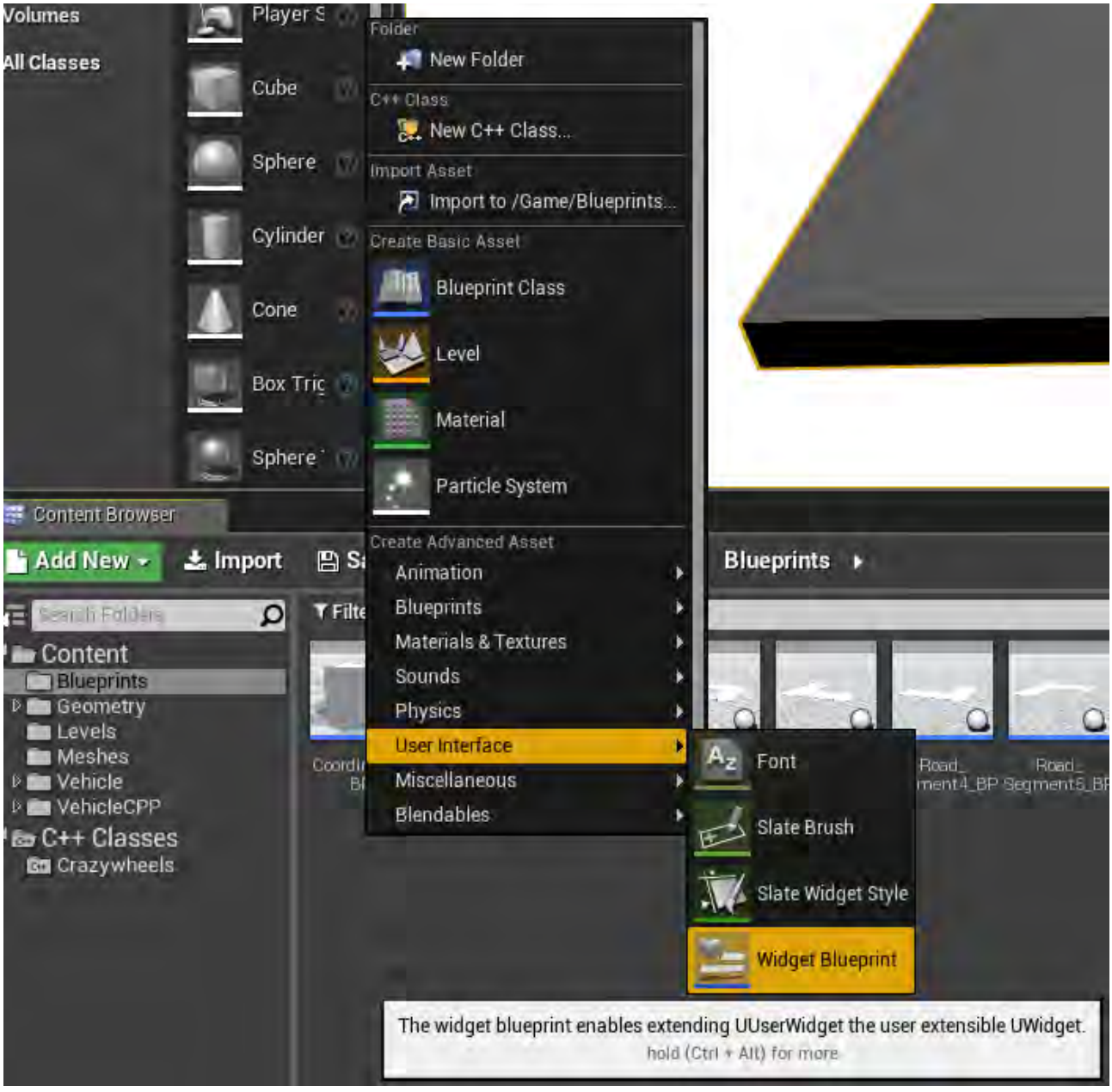
Pict

ure 4.19

Of course, to be able to “see” it from our Game Mode class, we need to include it (`#include"Blueprint/UserWidget.h"`) at the top of the file.

Next thing to do is create the HUD Blueprint to display the information we want.

Inside the blueprints folder, we create a new blueprint as seen in the picture below:



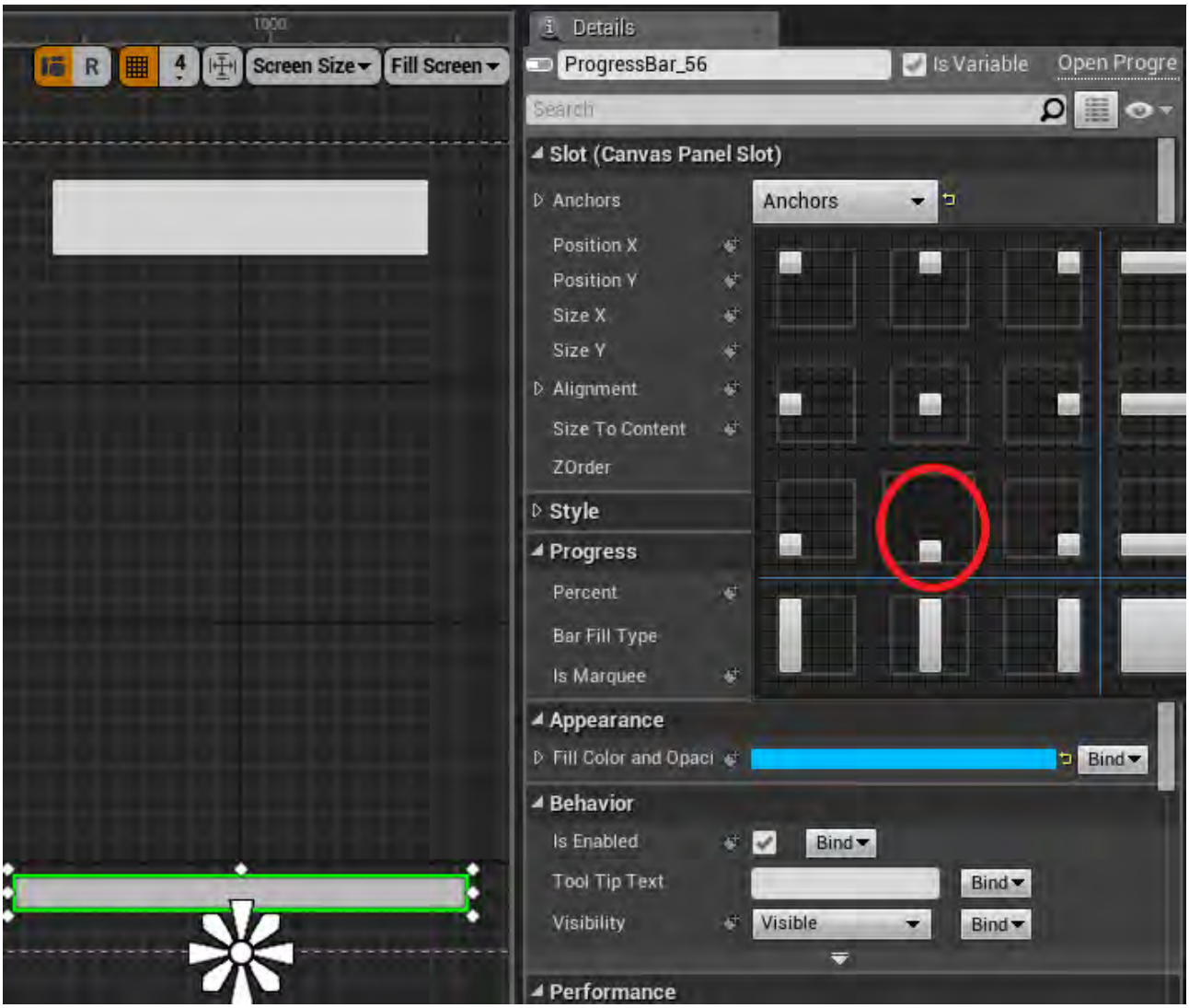
Picture 4.20

Right click → User Interface → Widget Blueprint. After naming (we named it

RoadHUD) and opening it, we see that there is a customized screen template, on which we will put the things we want to show. This will be done with “anchors” in order to pin the tools from the palette we choose, in order to remain in the same position regardless of resolution (for example if we want to create the game for different platforms. In this project, we will show two things on screen. First one will be the score progress bar, and the second one will be a message, showing if the player has won, lost, or is currently playing. So, from the left tab names palette, we drag a “progress bar”.

Picture 4.21

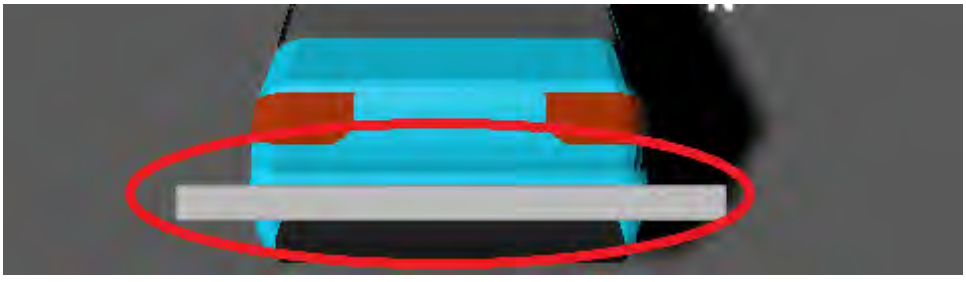
Then, after we have chosen the widget, we set the anchor to the lower middle part of the screen, as seen in the next screenshot:



Picture 4.22

This way, this progress bar will remain in that part of the screen, regardless of screen resolution or platform. After that, we adjust the position around that anchor from the same menu and set the alingment to 0.5, so that half of the bar is on the left of the anchor, and half of it is on the right.

Our progress bar is complete, but as seen in the picture below, it is empty.



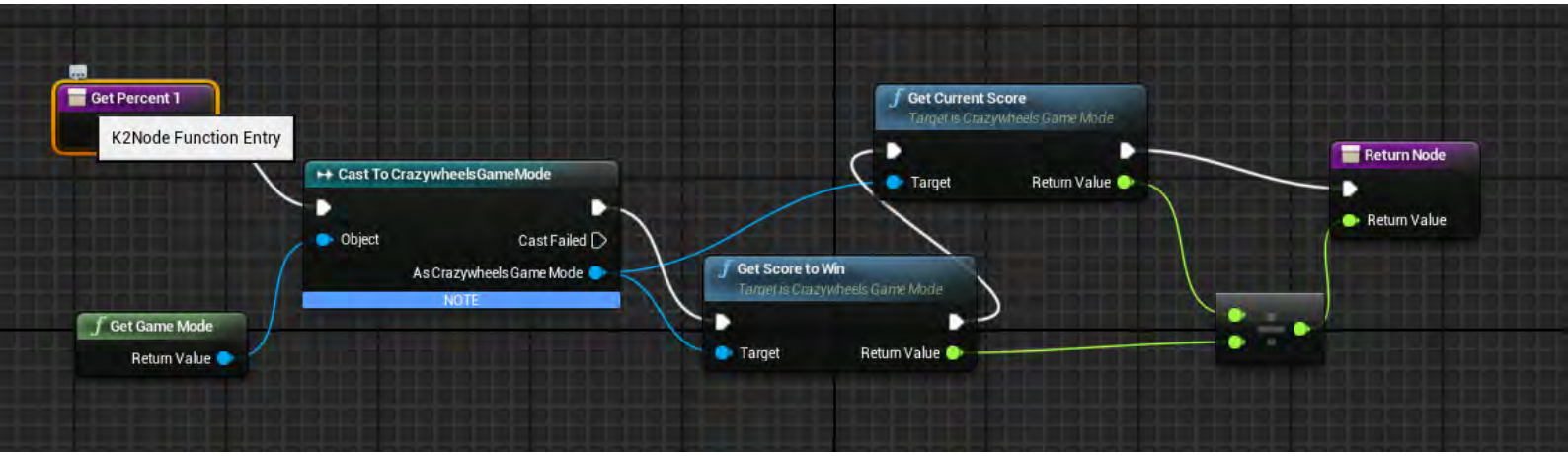
Picture 4.23

This happens because it doesn't have any functionality yet. What we are going to do, is make this bar change color up to a percentage. Theis percentage will be the current score/score needed to win. We talked about the score before, so we already know how it is counted. To do this, we are going to do a binding. In the details panel, in the progress category, we create a binding as seen in the following screenshot:

Pictur

e 4.24

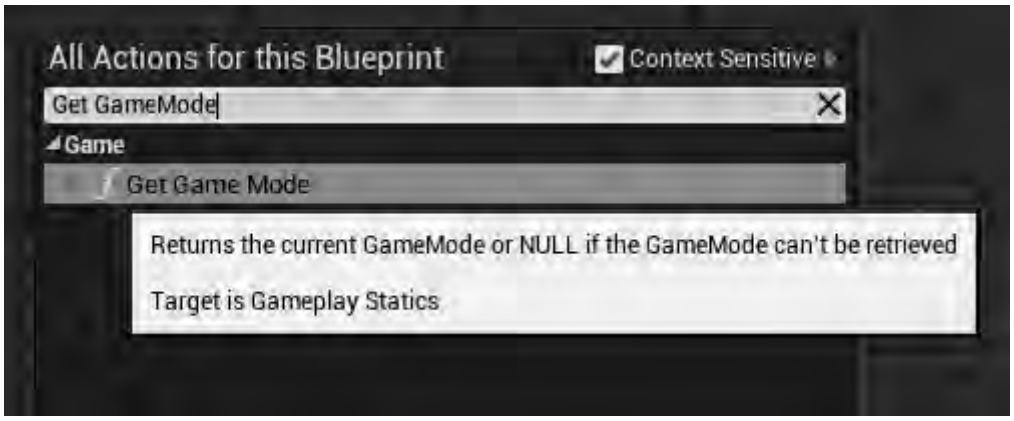
Here we have opened the full Unreal Engine Blueprint Editor. This editor is a node-based visual scripting tool, that allows us to execute blueprint commands by inserting them as nodes. I will firstly present what the procedure will look like in the end, and then explain step by step how we got there. The full functionality is shown in the screenshot that follows:



Picture 4.24

Each of these nodes is a command of the blueprints scripting language. Let's start explaining.
The purple top node is the Function entry node and exists by default. Everything else should come after this. The white line that comes out of it is an execute line. As we can see, this line exists in most of the nodes here. What this line does is say which nodes is executed after this one is complete.

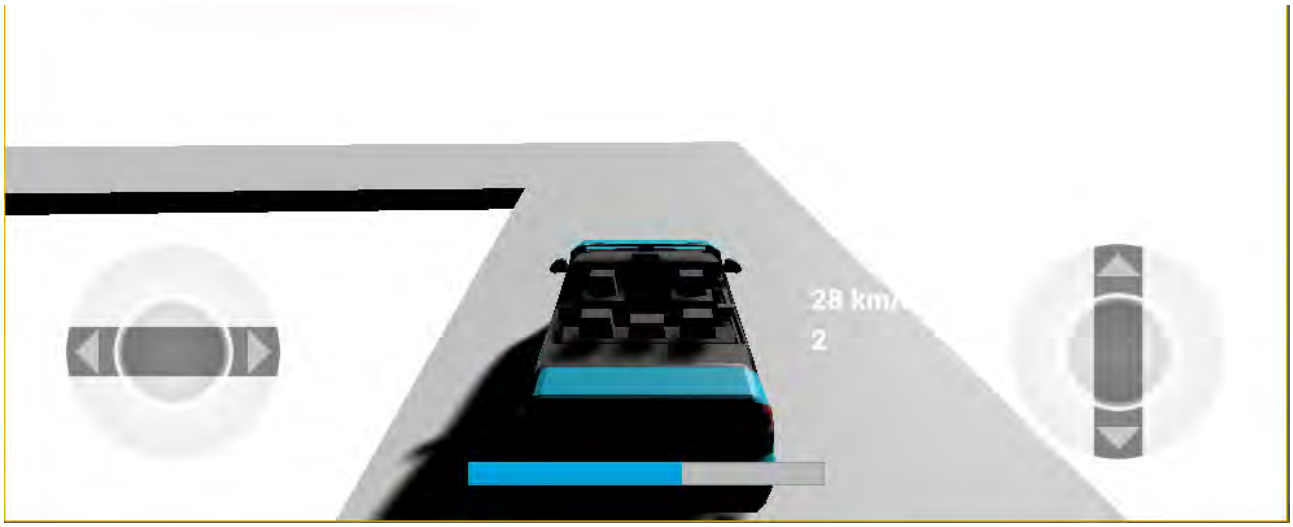
First of all, we need the variables ScoretoWin and CurrentScore. These variables are stored in the Game Mode class. So in order to access them, we need to retrieve the game mode. So, we right click and place the node Get Game Mode, as seen below:



Picture 4.25

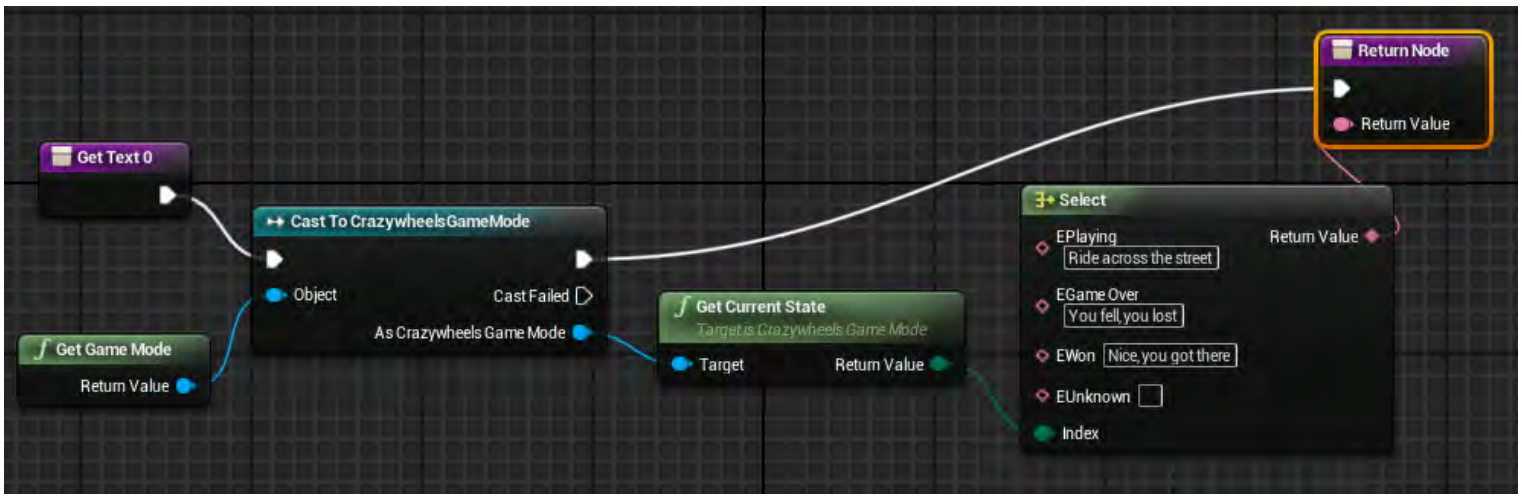
For all other nodes we just type the function name and choose it from the menu. After we select it, we drag a pin from its return value. GetGameMode returns the game mode we currently have. To be certain that this is the game mode of our project, we cast it to CrazyWheelsGameMode, and then (if the cast is successful) we execute the next two commands, which we wrote in C++ before, GetCurrentScore() and GetScoreToWin(). The return values of these functions are put into a divider, the result of which is returned.

Back in the details panel, in the category appearance we can choose the color of the colored part. Below is a snapshot of the progress bar half-full.



Picture 4.26

The last thing we will implement in this project is another widget. This one will display a message on the middle top of the screen, regarding which play state we are in. Once again, we open the RoadHUD blueprint, and drag a box entry inside the canvas panel. Then, like before we set the anchor to the point we want, and set the anchor to the point where we want. Then, in the details panel, in the content category, we create a binding in the text option. Following a similar procedure like the process bar, we end up with the following nodes:



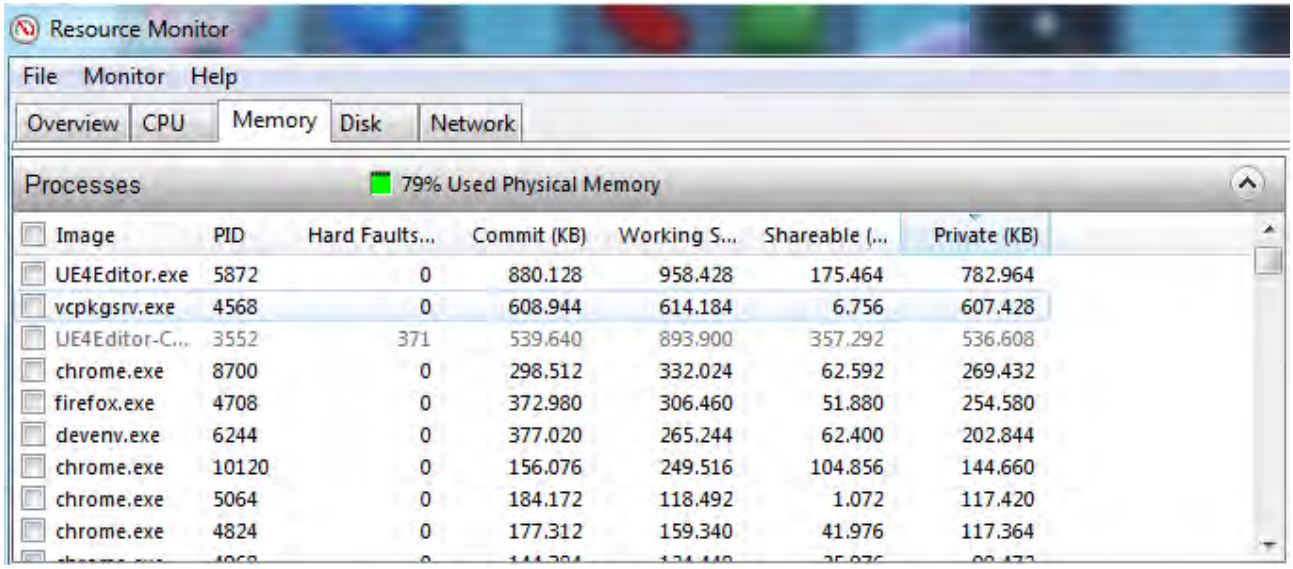
Picture 4.27

Since most things are similar as the progress bar widget, we will only comment the selector. The selector takes the play state as input, and for each value returns a text. If the player has won, the screen will display “Ride across the street”. If the player has won, the message will be “Nice, you got there”, and lastly, if the player loses, the message displayed will be “You fell, you lost”.

4.5. Packaging the project

In order to distribute our project, we must package it to ensure a number of things, such as the format it is built on for the device it is aimed for. To package the project, we go to File → Package Project and choose the platform we want to build for. However, the game will then be able to run on a computer that meets certain requirements, such as a minimum hardware, as well as some prerequisites, like C++ redistributable and Direct X 11 or 12. Even after the game is released, the developer can package patches to update the game in each user's device. Packaging is a costly process, but luckily it only needs to be done rarely, maybe even once. In a computer running a Core 2 Quad 2.5GHz and memory 6GB DDR2, the processor ran at full speed during packaging, and memory usage was at 80%. The whole process took about 5 minutes to complete. However, it goes without saying that creating AAA games takes a lot more time and requires additional resources and better computers.

A screenshot of the device manager at the time of packaging can be seen below, showing the additional processes created for this purpose (which are killed after packaging is done) and the ram usage.


Resource Monitor
File Monitor Help
Overview CPU Memory Disk Network
Processes
79% Used Physical Memory
Image PID Hard Faults... Commit (KB) Working S... Shareable (... Private (KB)
UE4Editor.exe 5872 0 880.128 958.428 175.464 782.964
vcpkgsrv.exe 4568 0 608.944 614.184 6.756 607.428
UE4Editor-C... 3552 371 539.640 893.900 357.292 536.608
chrome.exe 8700 0 298.512 332.024 62.592 269.432
firefox.exe 4708 0 372.980 306.460 51.880 254.580
devenv.exe 6244 0 377.020 265.244 62.400 202.844
chrome.exe 10120 0 156.076 249.516 104.856 144.660
chrome.exe 5064 0 184.172 118.492 1.072 117.420
chrome.exe 4824 0 177.312 159.340 41.976 117.364
Picture 4.28

Chapter 5
Conclusion

To conclude this assignment, there should be made some general notes on what we have accomplished, note some problems encountered during development, as well as draw some inductions for future reference. Last but not least, an evaluation should take place and point out options to make the game better.

5.1. The game created

The game created is a simple 3D vehicle game, created with Unreal Engine, and based on Unreal Engine's template for 3D vehicle games. It includes basic, yet powerful functionality, and simple methods that can create powerful and complex functionality in the game. More importantly though, the concept and mechanics of how a video game works was showcased, allowing the reader of this assignment to get a basic idea around game development. The basic mechanic around game development is that everything is a class. More specifically, in Unreal a C++ or a blueprint class. Unreal also allows for single inheritance, and most classes we use inherit from preexisting classes with lots of functionality. Regardless of the game engine used, this functionality is the same. For example, the actor class contains functionality for physics, can get a mesh to represent it in the physical level, collision, movement, textures etc. The GameMode class contains all the rules associated with the game.

Regarding the game, we introduced lots of functionality mentioned in the list below, but not entirely covered by it.

- We introduced basic actor classes, as parts of the road created, the road segments. The inheritance part was covered, since there was a parent Road_Segment class with some common functionality, from which the different kinds of segments inherit from. Furthermore, Blueprint classes were created for each of the child segments, which inherited and implemented functionality of their own. Blueprints and C++ working complementary to each other is the best way to develop games, as the powerful tools blueprints provide is combined with the execution speed of

C++ for the best performance. Creating and destroying actors was also implemented. Randomizing played a big part too, as each time the next segment to be created was chosen randomly. Another basic yet powerful functionality, overlapping, was implemented. This is used in video games as a trigger for events to happen. In this case, when the vehicle overlapped an invisible box created, this was the trigger for lots of actions to happen, like spawning the next segment, increasing the score and setting some variable values that allowed the algorithm to work.

- Casting was also implemented a couple of times. Casting essentially is when we check to see if a class of type 1 a child of another class, let's say type 2. In this project, Casting was used twice. Once when we cast all the actors overlapping the creation box to the vehicle class. This happened because there could be another actor overlapping, and we wanted to execute the code only when the specific actor reached that point. The second case is when we cast the game's GameMode class to CrazyWheelsGameMode. While in our game this couldn't be false, there are cases where the rules of the game need to change, so we should check for which game mode is active.
- Scoring and play states were also introduced, a game mechanics that implements the concept of winning and losing a game. In our case, the player wins when they have successfully passed over 5 road segments (actually when the vehicle has overlapped 5 creation boxes). The value 5 is randomly selected, and was selected to be low in order to test the game. This can be easily changed from the GameMode Blueprint, by changing the ScoreToWin value in the “Score” category. On the other hand, the player loses if a Road_Segment is about to be deleted and the vehicle hasn't overlapped yet. This functionality, albeit much different, is implemented in almost every game. For example, in shooter games, the game state changes to GameOver when the player's health reaches zero.
- An implementation of the HUD is also implemented. We used UMG (Unreal Motion Graphics) and implemented two widgets, a progress bar and a text box. The progress bar displayed the ratio between the ScoreToWin and the CurrentScore, by painting the percentage of the progress bar. The second widget, the text box, displays a text depending

on the game state we are. Almost all games include functionality like this, like when a health bar is displayed, a score, messages and many others.

- Last but not least, we packaged the game to play to be played in standalone version. Additional features were also explained, as the target platforms we can choose, the settings we can edit in order to make the game run more smoothly on the target hardware. A user reading this shuld be able to package a game for windows with no problem. Packing for additional platforms is the same regarding Unreal Engine, but may need additional software, like Android SDK to package for android devices.

5.2. Problems and difficulties encountered

During the stage of development, there were quite a few problems and difficulties.

The first one, which was quite quickly surpassed, was getting to know the environment of the engine, as well as the concept behind it. As soon as I understood that everything is a class, things became quite easy. Another problem was the compile errors I kept getting, especially at the beginning. These were easily surpassed too, as I became familiar with programming in blueprints and C++, as well as when the documentation became more familiar.

The major difficulties, however, had to do with Unreal Engine and Visual Studio. At some point, both these programs updated automatically, and errors introduced in the new version made it impossible for the code to work with the new version. With Unreal, the problem was that in the new version, some header files were removed from certain files to increase performance. It took a few days until developers posted a fix online about how to make the project again compatible with the old version. Ever since, the version used in this is 4.10, while the current version at the time this project is made is 4.13. The problem with Visual Studio was worse though, as reverting to a previous version or even uninstalling Visual Studio is really hard, and may even require a complete format of the drive. After trying a lot of things and combining different methods to erase all files and registry entries from the memory, I was finally able to reinstall the old version, and change settings so that Visual Studio wouldn't update to Update 3. Lots of other users reported the same

problems too in Unreal and Visual studio forums.

Besides these problems, everything else went smoothly, to which a huge part played the large amount of tutorials, guides and community members willing to help fellow developers.

5.3. Future updates

When it comes to gaming, one of the most difficult, and more important, questions, is “What else does it need?”. Creating future updates and making a game better, is one of the keys to success.

The game created in this project is hardly a complete game. It is a showcase of the most important game mechanics implemented in video games. From the programming side, any idea someone could mention could be implemented in the game. Comparing the game to a professional racing game, there are lots of things that could come in mind. Multiplayer implementation is one of them. This can be done through Unreal. The number of players is defined in the GameMode class. Of course it would be more difficult than what we did here, but it would follow the same concept. Another thing we could implement is the ability to pick up and use power-ups. Again, power-ups will spawn and destroyed, things we already did with Road_Segment. Also, the trigger to pick them up would be to get close, something we also implemented with the road_Segment and the creation box.

On the other hand, what this game lacks is graphics. As we said in the beginning, graphics was never intended to be a focus in this project, as graphics development is a completely different area than coding. However, to create a game at the level of today's published games, graphic designers are also needed to work on it, as well as use graphic assets to work on. Such assets, like models, textures etc are available through the Unreal Marketplace, but come at a cost.

Chapter 6
References

1. https://en.wikipedia.org/wiki/Game_engine#History

2. https://en.wikipedia.org/wiki/Golden_age_of_arcade_video_games

3.http://www.gamasutra.com/view/feature/173068/congratulations_your_first_indie_.php?page=2

4.http://www.riotgames.com/riot-manifesto

5.http://cs.gmu.edu/~gaia/SeriousGames/index.html

6.https://forums.unrealengine.com/showthread.php?99776-Unreal-Editor-in-VR-Official-megathread

7.http://www.gartner.com/newsroom/id/2614915

8.https://blog.mozilla.org/blog/2014/03/12/mozilla-and-epic-preview-unreal-engine-4-running-in-firefox/

9.http://www.html5report.com/topics/html5/articles/373408-html5-gaming-just-got-faster-with-unreal-engine.htm

10.http://www.polygon.com/2014/4/1/5568378/epic-games-ceo-tim-sweeney-unreal-engine-vr-oculus-rift

11.https://www.unrealengine.com/blog/ue4-is-free

12.https://answers.unrealengine.com/index.html

13.https://forums.unrealengine.com

14. https://wiki.unrealengine.com/Main_Page

15.https://www.unrealengine.com/blog

16.http://venturebeat.com/2012/11/02/game-developers-start-your-unity-3d-engines-interview/

17.https://docs.unity3d.com/Manual/index.html

18.https://unity3d.com/learn/premium-support

19.https://www.unrealengine.com/faq

20.https://unity3d.com/unity/multiplatform

21.https://docs.unrealengine.com/latest/INT/Platforms/Mobile/Performance/

22.http://not-lonely.com/blog/making-of/unity-ue-comparison/

23.http://not-lonely.com/blog/making-of/unity-5-vs-ue4-pt-2/

24.http://www.tomlooman.com/ue4-ufunction-keywords-explained

25.https://docs.unrealengine.com/latest/INT/Gameplay/Framework/Pawn/

26.https://docs.unrealengine.com/latest/INT/API/Runtime/Engine/Engine/UWorld/SpawnActor/4/index.html

27.https://docs.unrealengine.com/latest/INT/Engine/UMG/