PRIFYSGOL ABERYSTWYTH UNIVERSITY

MARS MISSION CONTROL

Supervisor: Dr Laurence Tyler – lgt@aber.ac.uk

Second Marker: Dr Helen Miles – hem23@aber.ac.uk

This Report is submitted as partial fulfillment of a BSc degree in Computer Science, Graphics, Vision & Games (G451)

4TH MAY 2018
LUKASZ WRZOLEK
Luw19@aber.ac.uk

Declaration of originality

I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the regulations on Unacceptable Academic Practice from the University's Academic Quality and Records Office (AQRO) and the relevant sections of the current Student Handbook of the Department of Computer Science.
- In submitting this work I understand and agree to abide by the University's regulations governing these issues.

Name: Lukasz Wrzolek

Date: 24/04/2018

Consent to share this work

By including my name below, I hereby agree to this dissertation being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Name: Lukasz Wrzolek

Date: 24/04/2018

Content

1. Abstract

Mars Mission Control is the project developed in Unreal Engine 4 by Lukasz Wrzolek. The main idea of the game is to give the player a chance to be on Mars and explore Victoria Crater. The Mission takes place in 2020 and it is a virtual representation of an existing NASA Mission on Mars. The Rover named *"Opportunity"* discovered the crater in 2006, then explored it from the inside for nearly a year to collect the data and measurements. Rover was collecting the image data using his Panoramic Camera (Pancam) and then send it through space to the base of the Earth.

The game provides a similar experience: The player can drive or fly around the crater, and take the scans for NASA. The modeled Crater is the 1 to 1 representation of Victoria (800-meter diameter) including all the shapes, sand waves and rocks around. Experience can be even more realistic as a player can use VR helmet or VR cardboard to play the game, switch the controlling object and fly around. People playing the game have an option to choose the controller type between Keyboard and Mouse, and Gamepad. While playing in VR, developer highly recommends using the gamepad and flying character to get the best experience and views.

2. Background, Analysis & Process

Background:

The investigation process took over game engines like UE4, Unity, Red Engine to improve the final experience of the application.

Unity is the engine with large community database of various projects and you can always find the help needed if you are struggling with something, but the engine itself uses C# and JavaSript languages to the program, and I did not feel comfortable with learning a new language to finish the project.

Red Engine is the popular engine from CD Project Red and it is an amazing tool to create a large game with advanced decision system. The engine is mostly used to create mods to the games like Witcher, and lack of tutorials/community help had a major impact on not to use Red Engine.

The chosen engine is Unreal Engine 4, created by Epic Games. The Engine itself gives a wide range of opportunities to develop the game, physics and import models from the other sources like Blender, what more – there is a possibility to develop the project in 3D and Virtual Reality, and develop the specified functionality of the engine by creating C++ classes. Except for this, the aim of the research was to identify various resources, which can help to visualize Victoria Crater, Mars environment, and Opportunity Rover. The main resource for the image data was the NASA News web page **[10]** to check how the environment looks at different parts of the day and how it changes the feeling of the planet. Furthermore, the recreation of Victoria Crater was able by using image data provided by Dr. Laurence Tyler **[5]**. The images show the crater from various perspectives, including an Orbital view of it.

Analysis:

Mars Mission Control is a Software Development project. It addresses various parts of the computing industry from research, through visual scripting, and finally to plain coding. All the things mentioned above were closely connected to the powerful engine used to develop the game. Unreal Engine is the software based on C++ and gives full access to the code inside the application. It has been used many times as a part of research to see how many things are connected together. The powerful software like UE4 comes with various bugs and unexpected crashes, as the engine itself is still in the development phase. Every month Epic Games tries to patch the most important problems by releasing the new version of the software.

The initial specification of the project and task list was fully achieved:

"Proposed task list:

- Modeling the Victoria Crater and Base - Achieved
- Environment modeling - Achieved
- Modeling the rover - Achieved
- Developing physics and gravity of the planet and implementing it to the rover - Achieved
- Developing user interface - Achieved

Optional tasks:

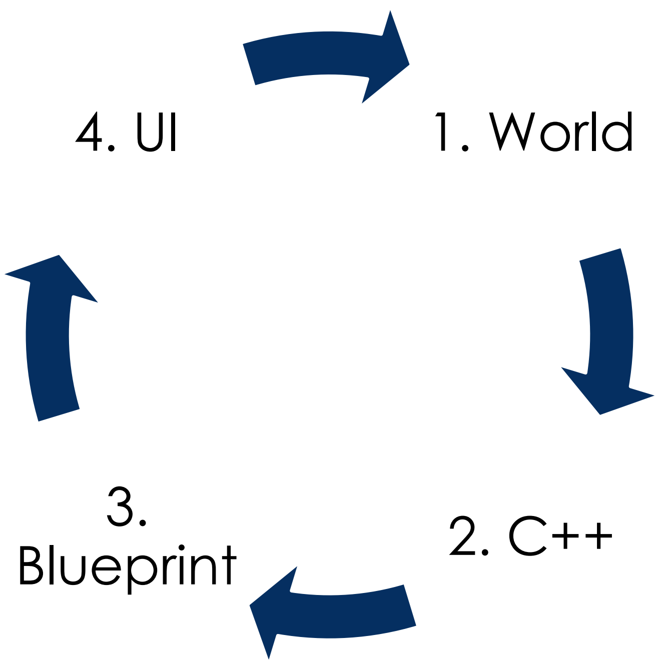
- VR experience - Achieved
- A time machine to travel back in time – not achieved." – Lukasz Wrzolek (Project outline Specification).

The environment modeling should be called as environment design and it includes atmospheric fog and Day – Night cycle. The environment changes independently and the player is not able to interact with it.

The main problem of the project is that models are imported from Blender. Blender is a free to use modeling software with a large database of users. It is quite simple to use from the start and provides complex tools as you gaining the experience. The main issue is that Blender world coordinates are not similar to Unreal World coordinates. It means that meshes imported from Blender has a different axis and the models are in different position. The model comes rotated 90 degrees on the X-axis, and the inside vectors are different as well. The problem will be further explained in the chapter: **Rotation and Movement.** This problem relates to the main problem of rotation of the particular components and solution for it (chapter: **Rotation Component).**

Process:

The proposed method to follow was Waterfall model, but it doesn't work really well for the big game project created in UE4, where designing part is as important as coding part, and each of this parts is dependent. The simple meaning of this is that every implementation in C++ was connected to the Visual Scripting solution. Component Initiation, Opportunity Rover components initialization, required a different type of model: Cycle model – The model was relatively simple to follow as it requires to check the other parts of the software, and it works perfectly with Unreal when changing one thing can cause a hard crash of the engine.



Cycle model requires checking every aspect of the software, while you are making changes in one of it. The most time-consuming part of the project was the world creation – which in that case is Victoria Crater. Doing this step by step, the initial skeleton of C++ code and classes was made, most of the Blueprints created and placed in the World are derived from the C++ classes, and their functionality is described by the code, and the end of the process is a simple UI creation to check the difference while playing the game from different level. As mentioned above Unreal is still in development process, so playing the game from Main Menu can result with a specified bug – the camera spawns in various locations. It can spawn in a specified location once, while you play the game next - it can spawn under, or on the side of the rover so you have to move it around to the original position.

The cycle process was crucial while the game was theoretically completed, but the code architecture was a mess. Every class had a reference to another, the pointers were going to the places where there was no need for that. Relaying on refactoring process: "Red, Green, Refactor" **[13],** the class, code and the structure has changed, and by the following Cycle process completed saving all the core functionality. The refactoring process will be described in **Classes in C++ and Blueprint Dependencies.**

3. What is a Game Engine?

The Game Engine or the Engine is a structure which helps in a priority element of the game like Graphics (Game visualization and look), Audio, Logic, Physics, Networking, and Artificial Intelligence. The engine is a program which provides the structure and tools to create the game. To describe better what the Engine is we have to go back in time and take a look at the first games.

Mario is a well-known game from 1983. It was created from scratch as in early 90's something like game engines didn't exist. The game scene built from pipes, simple terrain, character, enemies and bricks. Every single object had to have their World Space Coordinates, specified behavior, and at the end has to be deleted. The object rotation was needed by specified memory size of the hardware, otherwise, the game would simply crash. The concept of Memory Manager was created to dynamically assign, relocate and delete virtual objects, and today the game engines are simply modified Managers with more functionality mentioned above.

3.1 Unreal Engine 4:

Unreal Engine 4 is the open source engine created by *Epic Games* **[1]**, and it is the evolution of previously used Unreal Engine 3. The engine is free to use for everyone, and to use it you have to register on the Epic Games website, download the Launcher, and log in. *License agreement* **[2]** contains the *"Royalty"* section where are described terms of fees for any gross revenue.

Unreal Engine is one of the most powerful engines available on the market with the possibility to create the game with no coding background. It is created from C++ code assets and developer has a full access to every class in the engine source code, it gives unlimited possibilities of adapting the engine to the particular game and tasks.

<u>3.1.1 Blueprints in Unreal Engine:</u>

Blueprints in Unreal Engine is a Visual Scripting system to create a complete gameplay using node-based interface within Unreal Editor. *“As with many common scripting languages, it is used to define object-oriented (OO) classes or objects in the engine. As you use UE4, you'll often find that objects defined using Blueprint are colloquially referred to as just "Blueprints."*

This system is extremely flexible and powerful as it provides the ability for designers to use virtually the full range of concepts and tools generally only available to programmers. In addition, Blueprint-specific markup available in Unreal Engine's C++ implementation enables programmers to create baseline systems that can be extended by designers.” **[3]**

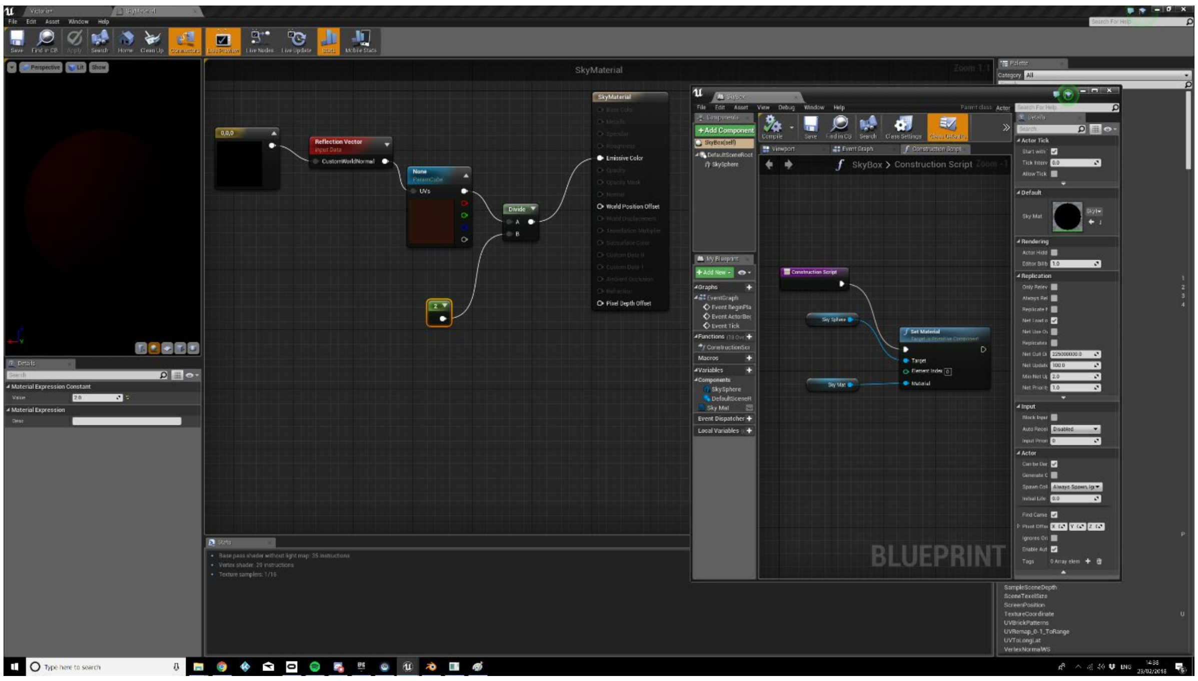
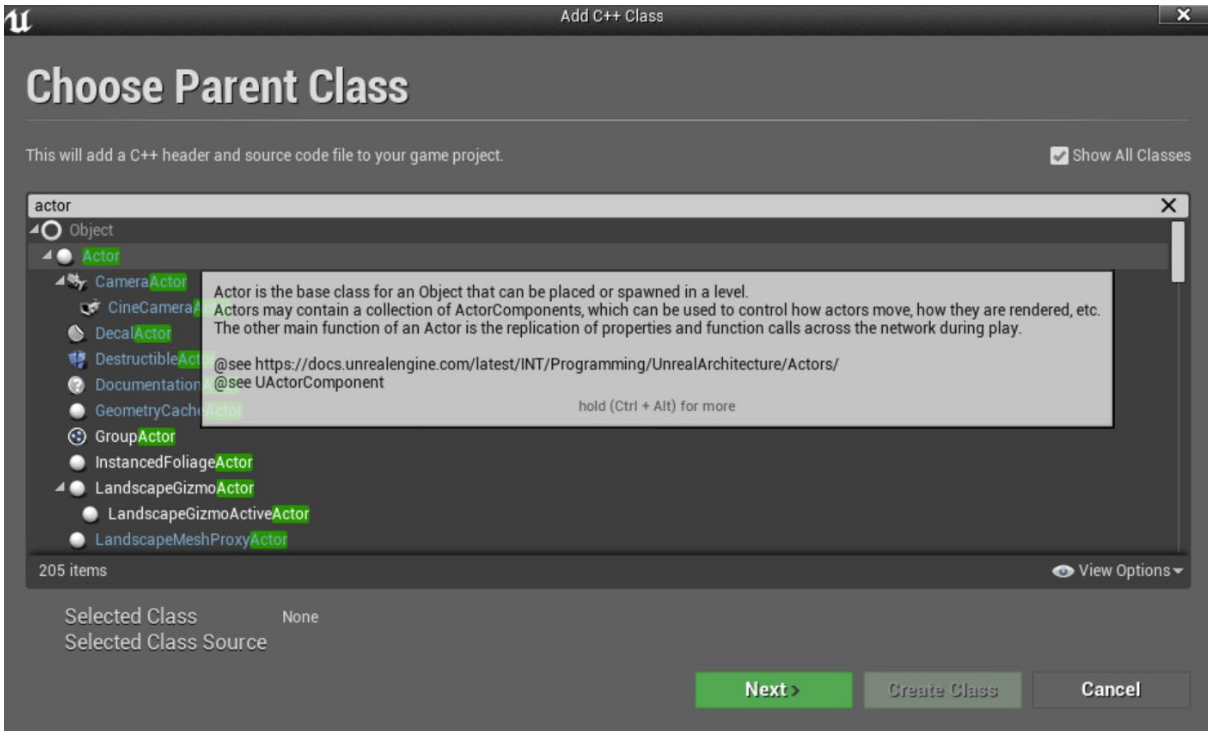


Figure 1: The figure above shows material creation for Sky in game and then applying it using Blueprints.

There are various different types of Blueprints in Unreal. Each one is different to another. Actor type blueprints are able to build an object from different meshes and be spawned into the world. Component types are responsible for being a component of the assigned actor and define their behavior, actions, and parameters. The best example is “Opportunity_BP”, which is the blueprint based on Pawn class responsible for spawning Opportunity Rover to the World during the gameplay. The player can take control of the Rover and drive around. It has two components “Rotation” and “Movement” to add to the rover possibility to rotate its elements, drive around etc. (Full description of the components in the chapter: 5.6). Blueprints have their own hierarchy: Level Blueprint, Game Mode Blueprint, and the other types. Level Blueprint is very sensitive as it is responsible for describing the level rules, and main operations. All the blueprints (excluding User Interface) has 3 main tabs: Viewport (how the blueprint looks like, when spawned in the world, and main settings), Construction Script (The script if the blueprint has to have any particular behaviour, look etc), and the Event Graph (Main behaviour, controls, abilities). User Interface is a bit different, as mainly it has only Viewport, but when we want to add the functionality to any object added to the viewport we have to bind it with the function defined in the “Graph” (Top right Corner, switch between Graph and Designer view).

3.1.2 The relation between Blueprints and C++

Class creation in Unreal Engine is not easy. You have to have the concept of what you want to create, what functionality it will have if it is a character/player or component if it is Actor or Pawn and by choosing or guessing wrong your final result can be different from the expected. Creating the class is based on the knowledge of what functionality you expect to have, and which class you will inherit from **(Figure 2)**. Each class can be a basement for your Blueprint in the editor, which means that Blueprint is based on the class, and that class describes the behavior, parameters, and functionality of that object. Every node existing in Blueprints is the representation of the method or parameter in C++ and that method has to be specially implemented using macros to be visible in Blueprints **(Figure 3)**. The code in C++ and objects created are depending on information passed from the blueprint to use and apply correct functionality to the object in the game **(Figure 4)**.


Add C++ Class

Choose Parent Class

This will add a C++ header and source code file to your game project.

Show All Classes

actor

Object
Actor
CameraActor
CineCameraActor
DecalActor
DestructibleActor
DocumentationActor
GeometryCacheActor
GroupActor
InstancedFoliageActor
LandscapeGizmoActor
LandscapeGizmoActiveActor
LandscapeMeshProxyActor

Actor is the base class for an Object that can be placed or spawned in a level.
Actors may contain a collection of ActorComponents, which can be used to control how actors move, how they are rendered, etc.
The other main function of an Actor is the replication of properties and function calls across the network during play.

@see https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/Actors/
@see UActorComponent

hold (Ctrl + Alt) for more

205 items

View Options

Selected Class None
Selected Class Source

Next › Create Class Cancel

Figure 2: Class creation and choosing the parent class

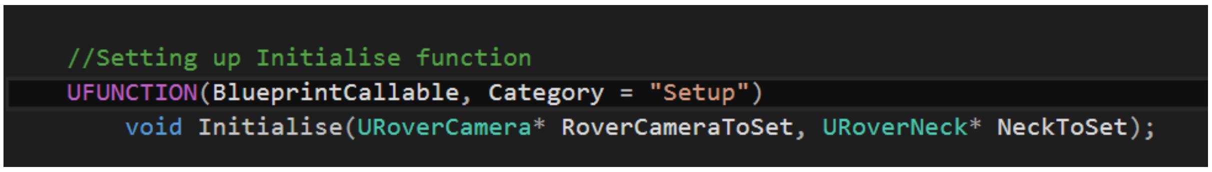

```
//Setting up Initialise function
UFUNCTION(BlueprintCallable, Category = "Setup")
    void Initialise(URoverCamera* RoverCameraToSet, URoverNeck* NeckToSet);
```

Figure 3: Macro Specification in C++

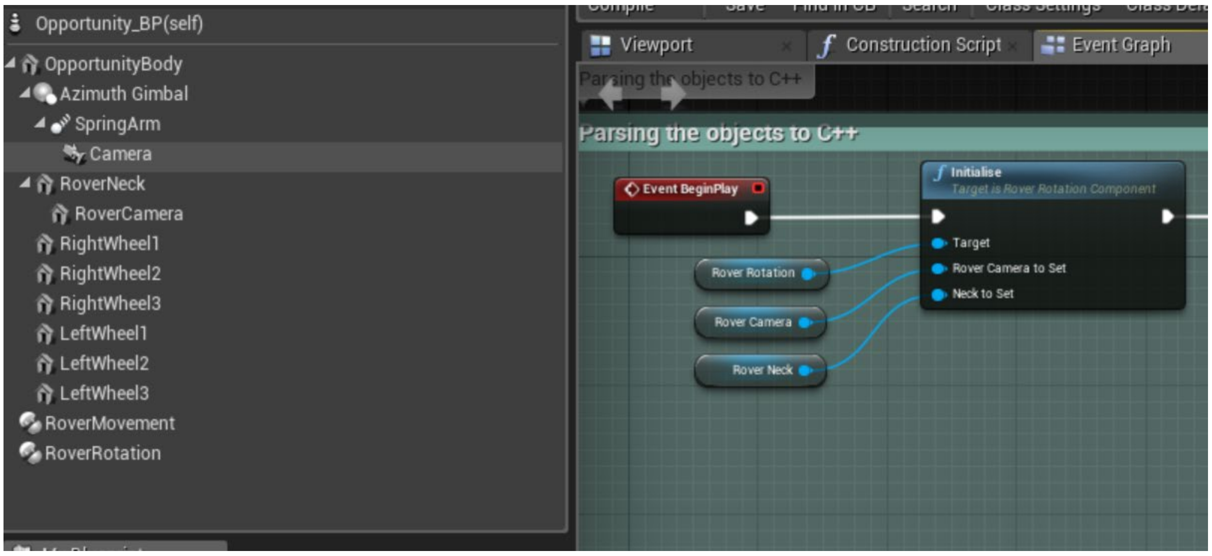


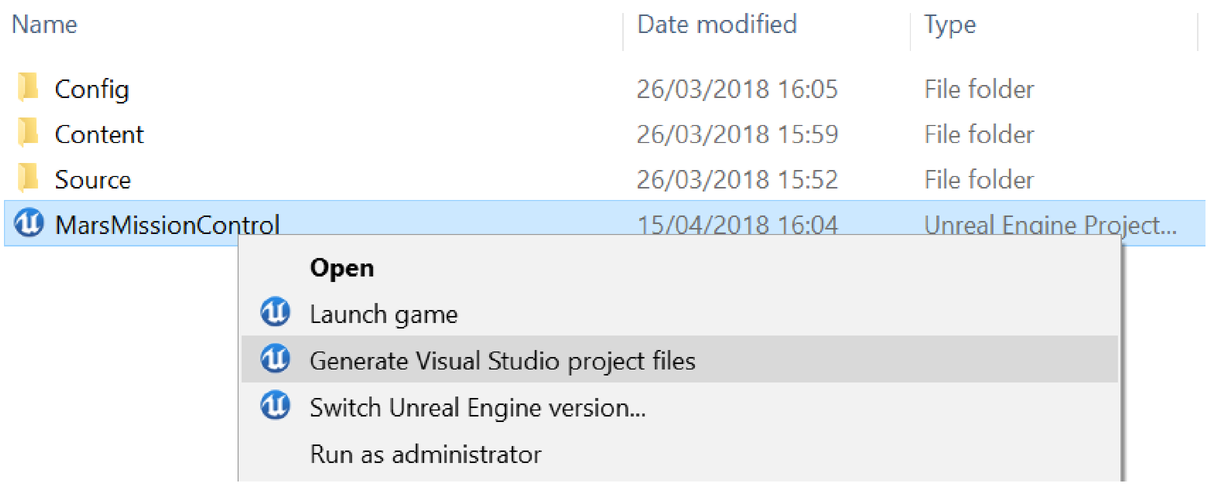
Figure 4: Dependency between Blueprint and Function in C++

4. Project Rebuild

**This section is created for marking purposes. It will describe how correctly rebuild the project to achieve the same effect.*

The project will be packed into a **.zip** file, which contains a **.exe** file – the final application, Config folder, Content folder, Source folder and **.uproject** file. To properly run the engine, you have to be logged in to Epic Games Application. It is necessary to download the engine and rebuild it. The project is based and created on the engine **version 4.16.3**, and to open the project you have to install and download the exact version of the engine. In Epic Games Launcher Application go to "Unreal Engine" -> Library -> Click on "+" – Install the new version of Unreal Engine -> From the expanded list choose 4.16.3, and then press "Install" **(Figure 5)**. When the installation is complete, extract the files to one folder. The second application that you will need is Visual Studio **[4]**. Epic Games are closely related to that software and Unreal Engine requires Microsoft product to compile, and run the code **(Figure 6)**. Both applications are required to rebuild the project, and the process should start from "Generate Visual Studio project files". The process creates folders required to run the Engine with all the settings specified by the developer.

Recreating missing folders is a quick process and it should generate (.vs, Binaries, Intermediate folders, and MarsMissionControl visual studio file). While all mentioned folders and file are generated, open Visual Studio file – "MarsMissionControl". Before Launching the Unreal 4 Engine, the project needs to be rebuilt, with the lighting, and solution. To do that on the right-hand side click on "MarsMissionControl" under Games Folder, and click "Build" **(Figure 7).** Until the process is finished, please do not open the engine.



Name	Date modified	Type
Config	26/03/2018 16:05	File folder
Content	26/03/2018 15:59	File folder
Source	26/03/2018 15:52	File folder
MarsMissionControl	15/04/2018 16:04	Unreal Engine Project...

Open
Launch game
Generate Visual Studio project files
Switch Unreal Engine version...
Run as administrator

Figure 5: Visual Studio project files

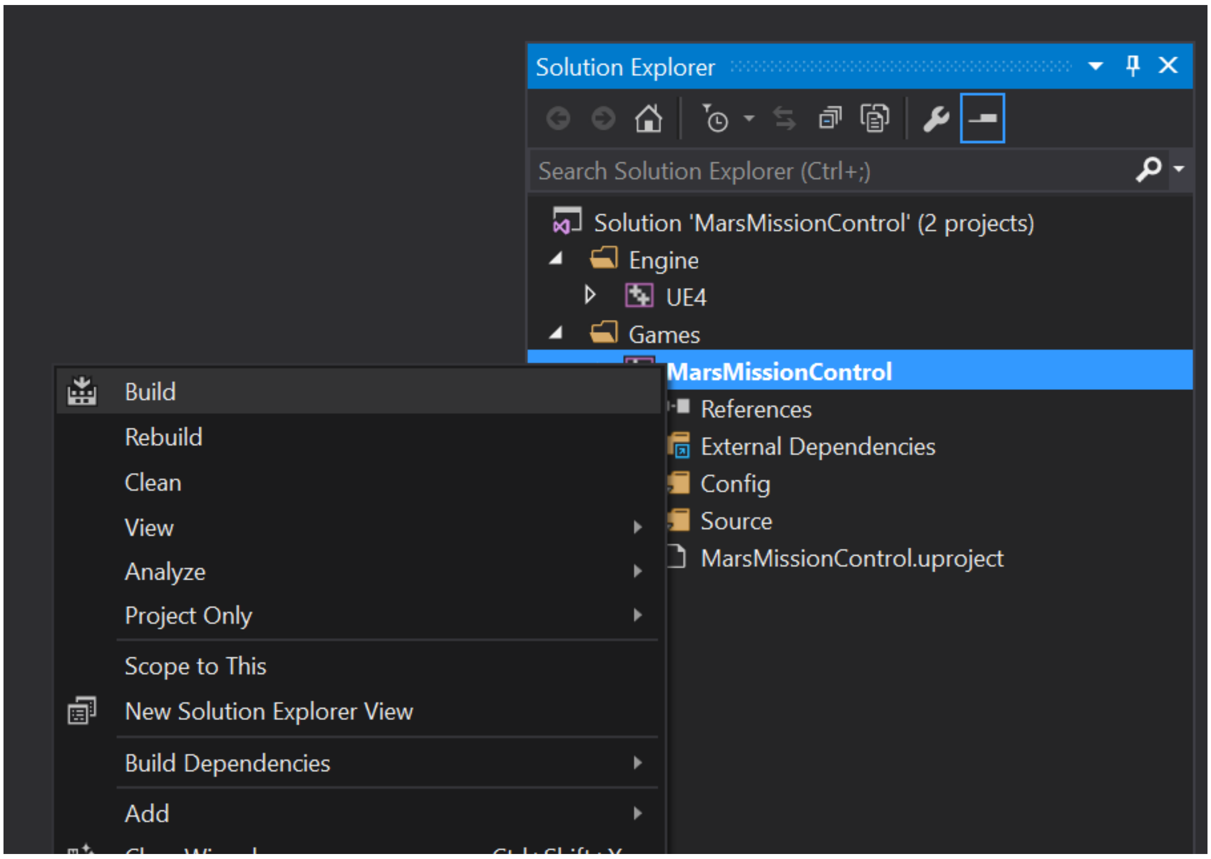


Figure 6: Rebuilding project Solution

Last part of the process is opening the **.uproject** file, which will create "Saved" folder and run the engine. The first run requires Lighting Rebuild – look at **(Figure 8).**

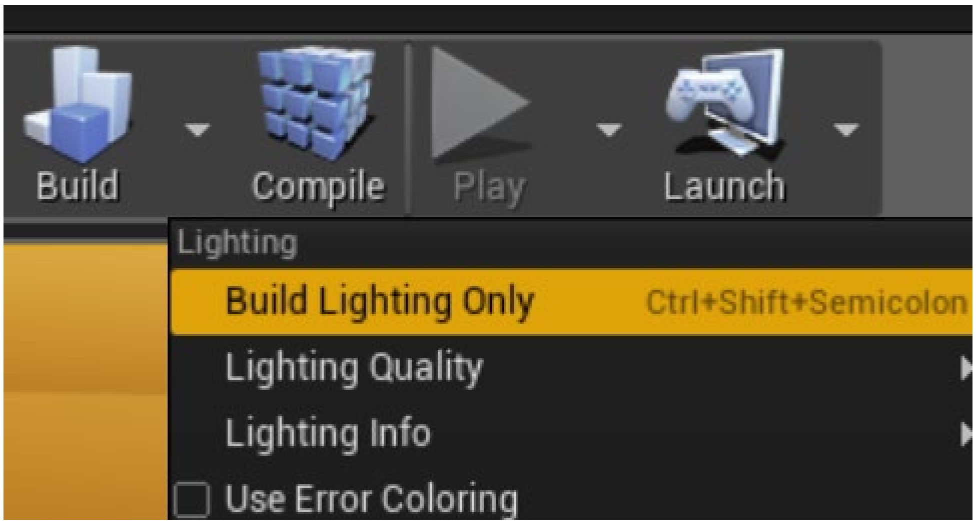


Figure 7: Build -> Press dropdown arrow -> Build Lighting Only

Rebuilding the Lighting is the long process and the duration time depends on the hardware.

4.1 Project Content and C++ Classes

The project is divided into two folders. First named "Content", which gathers all the Models, Meshes, Materials, Particles, and the second-one "C++ Classes", which includes the classes created. Blueprints folder contains the three main blueprints: Two of them "Action Interface" and "Swap Pawn" are the main blueprints responsible for passing the controller from one actor (Opportunity) to another(Flying Char). "Mars Mission Control Game Mode_BP" is the main game mode, which specifies the default Pawn, Objects to respawn, and fills up the character swap, by its own blueprint graph. "Geometry" folder contains basic shapes and meshes. "Levels" folder has two levels: "Main Menu", which is separate level only for the Menu, and "Victoria" – which is the main game level. "Opportunity" folder divides into three separate folders: Materials, Meshes, and OpportunityBP. "Materials" contains all the materials used to design the rover, specified physics material, and flash particle effect. "Meshes" is the folder with models imported from Blender, where I created them, and "OpportunityBP" is the folder with the main blueprint for the rover, player controller, camera flash and second flying character. "Starter Content" is the folder provided for starters by Epic Games **[1]**. "UI" is the folder responsible for User Interface parts, main blueprints for display, and showing content. "Victoria Crater" folder contains all the environmental effects and parameters. It is divided into seven different folders: Blueprints, DayNight, Level, Map, Materials, Sand, and Textures. Level and Map folders contain only the crater build data. Blueprints folder has all the actor blueprints visible on the scene: Sky, Sun, Moon, and hidden one "Pivit". The daylight folder is the representation of Day/Night cycle created in Level Sequencer. The sequencer is tool provided by Unreal, and using it developers are able to create non-linear animations and sequences, modify the object main coordinates like location and transform, create cutscenes and iterations. Materials folder has the materials used to create the sky, sun, and moon. The sand folder has sand material, which is applied to the crater. Textures folder is the texture of the sky.

C++ classes folder contains all the classes created by the developer. It should have 8 classes, and clicking on one of them takes us to the code in Visual Studio.

5. Modeling

The biggest part of the project and the most time consuming was the modeling. The project aims to reproduce Victoria Crater and try to give the representation of Mars environment. The highly detailed photos of the crater were provided by Dr. Tyler **[5]**, and covered every angle of the crater, alongside with the orbit view.

5.1 Victoria Crater in UE4

Unreal Engine 4 provides Landscape mode to create various types of the terrain. The very first landscape produced in UE4 was far from the "best visualization" of the crater, and Dr. Tyler suggested to use heightmap of the crater to achieve a better result. The best solution was using "heighfield_outde m_050.png" file provided by Dr. Tyler and polished in the engine of the reproduced map. Importing crater from the file gave a result of 15 KM crater with various weird effects. The middle of the crater was flat covered with few waves so the most of the waves had to be reproduced. The corrosion effect on the side walls was added by using "noise brush" in the Landscape tab, to achieve the more natural look. The "Sculp" Tool helped with achieving natural terrain outside the Crater, and after that, to reduce noise effect, the terrain was polished by "Smooth" Brush in the landscape tab. At the end of the process, the crater had "Sand_MAT"**[6]** material applied.

5.2 Opportunity Rover in Blender

Opportunity Rover has been divided into 4 main parts to focus on the details and further give the different abilities to the particular part of the Unreal. The rover has 4 main parts: Camera, Neck, Body, and Wheels. To create all of these parts students was following the pattern of existing model created by Christian A. Lopez **[7].** However Christian created full animated Rover as a single unit, and students approach was slightly different, he used the textures provided by Christian to create materials and map his parts similarly to the original. The rover has been created by using simple shapes and further changing the visual look by correcting the "faces" and "edges" in "Edit Mode" in Blender. The task was even more challenging as I didn't have previous knowledge and experience of modeling. The Body of Opportunity is the simple Cube, the suspension has been created from cubes as well, but to achieve similar shapes, the developer was using "extrude" option, and extruded faces to the particular directions. Neck and Camera models are simple and following the same creation pattern. The most challenging part was remapping the textures to the particular part of body, panels, and suspension, so names and ids had to be the same as in inspired Christians **[7]** model.

5.3 Base in Blender

The player starting point is in base placed on the West side of the crater. The base is a simple dome created from the single circle in Edit mode provided by Blender. The hardest part was to import the mesh to Unreal Engine and reproduce correct Collision boundaries to give the player possibility to spawn inside the base. The detailed collision mesh by the following tutorial promoted on Unreal Wiki **[8] –** by "Metal Game Studios".

5.4 Materials in UE4

There are various ways of creating the material types. One way is to creating them in external software and importing, but it doesn't work most of the time. According to Unreal Documentation and Materials section **[9],** the development of "Sand_MAT" material with samples exported from Blender. Materials are created in special Material Blueprint **(Figure 9)** called Material Expressions. The blueprint has a base for every material to specify basic parameters and look. Materials can represent any existing object in the real world, and only the experience level and knowledge limit the developer imagination. Materials are the crucial aspect of every game as it is the visual representation of the details and objects in the game. The sand material has been modified by applying wind node to "World Displacement" node. It creates the sand material to appear as wavy, so it illuminates the wind and creates living environment instead of static one. The wind itself is impossible to create as a particle effect now because there is a bug in Unreal which makes the particle effects disappear at certain angles. This means that the dust was taken from the Mars surface as a flying "sand" can appear in front of the player and suddenly disappear. It completely destroys the hoy of the game.

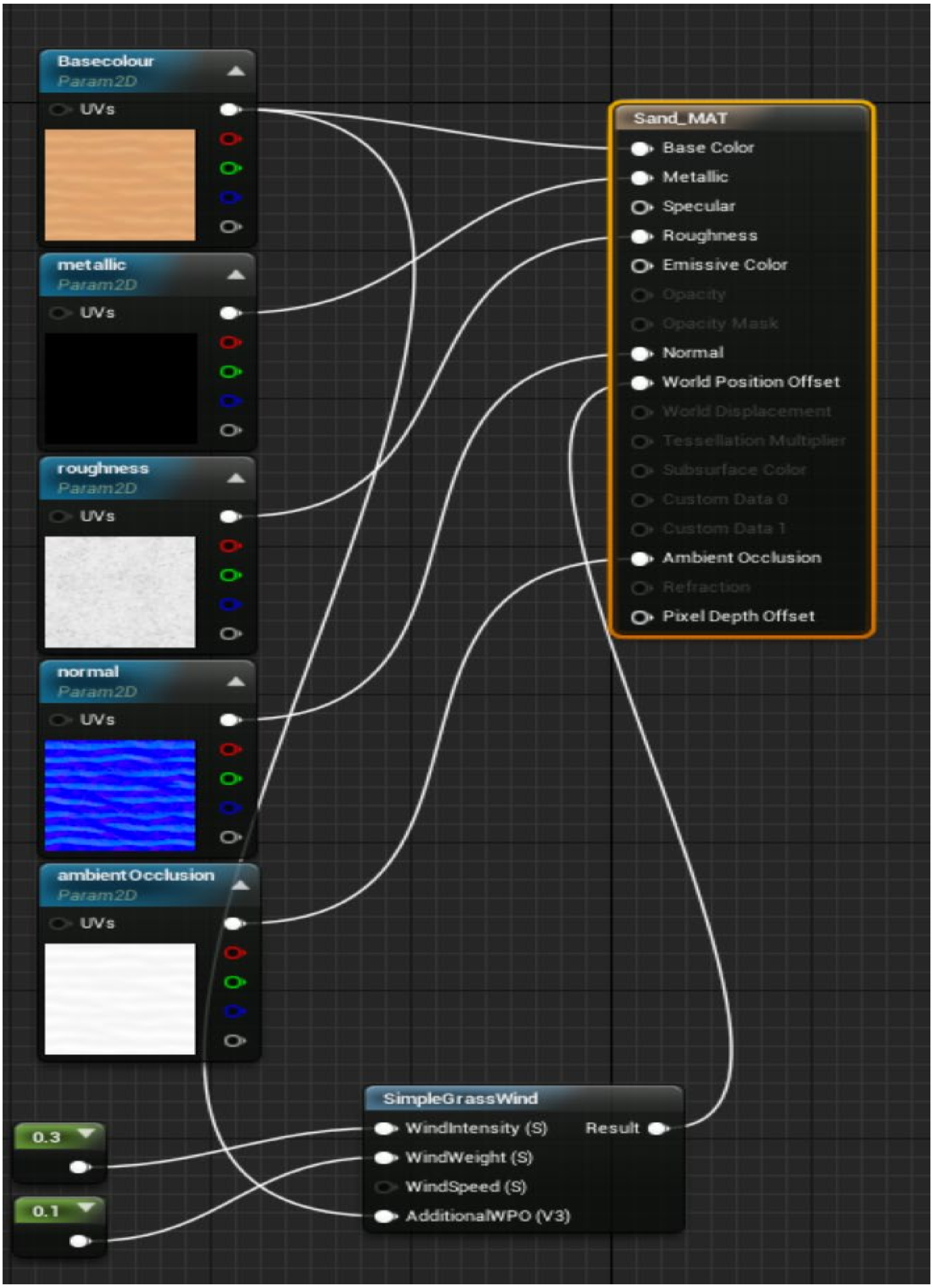


Figure 8: Material Blueprint

6. Game Design

Game Design is one of the most crucial things in game development. It is not only the visual aspect of the game and User Interface, but also structural, and interactional. The core of Mars Mission Control game is built in C++ code, but the big role of development is how the code works with blueprints. This section will focus on every aspect of the core gameplay, User interface, and how everything was implemented in the code, and finally how it works in the Unreal Editor. The code is implemented using a coding style with Unreal Coding Standards **[9]** to achieve the most efficient way of using the engine through C++ classes.

6.1 Main Menu

The "Main Menu" is created on its own level to maximize the performance and avoid long loading times. It is built from User Interface Widget, which is the type of Blueprint with special viewport modification to add buttons, text and bind events to the particular parts of the interface **(Figure 10)**.

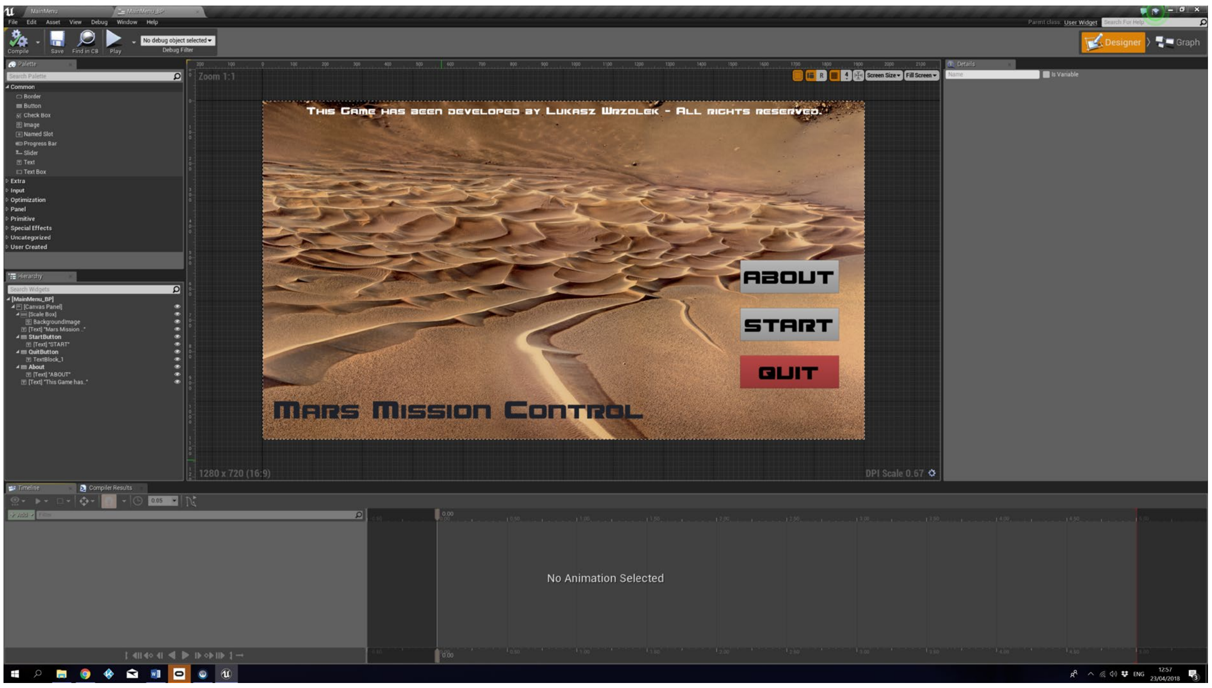

This game has been developed by Lukasz Wrzolek - All rights reserved.
ABOUT
START
QUIT
MARS MISSION CONTROL
No Animation Selected

Figure 10: Main Menu Blueprint

The difference between this blueprints is significant. On the left-hand side, we have different tools assigned to create the best-looking interface. Various types of buttons and sliders, progress bars, and Scale Boxes to align the layout to various types of the screens. Every created button comes with the same abilities to create events **(Figure 11,12)**.

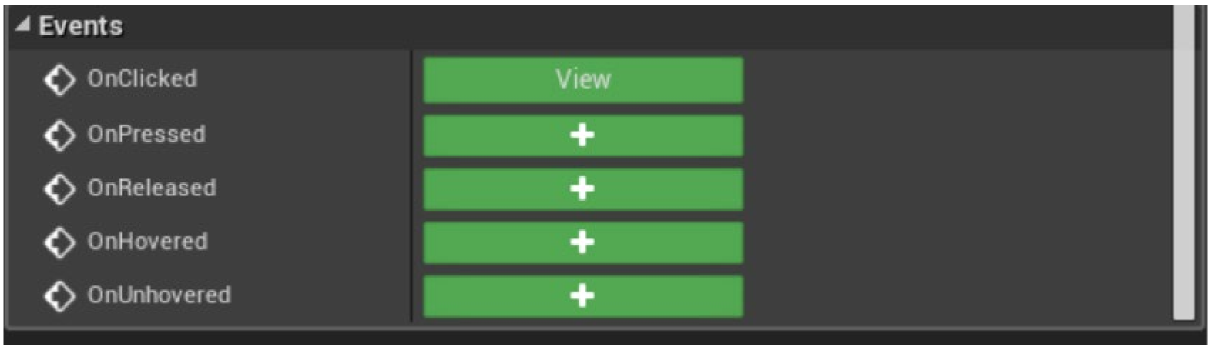


Figure 11: Button Events:

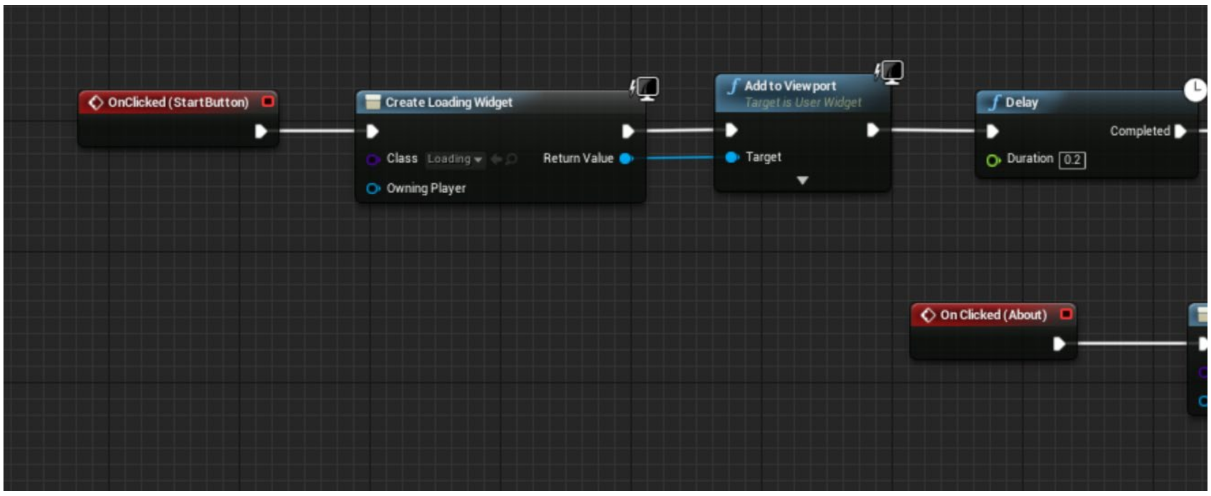


Figure 12: Event Graph for various buttons

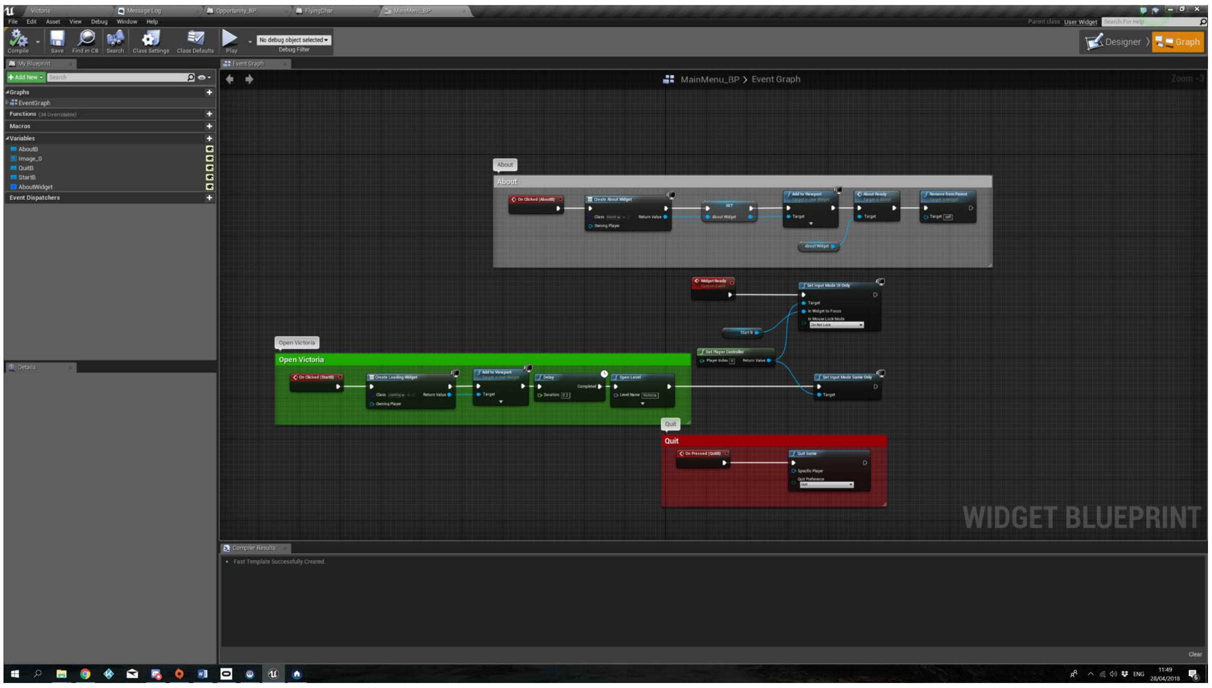
Every Widget type Blueprint has the switch on the Top Right corner. It is used to see the event graph, event creation, and the action flow. Interface Blueprints provides two types of binding – Event Binding **(Figure 12),** and Function Binding **(more in chapter 6.4).** Main Menu is built from three different widgets:

1. Main Menu – The main blueprint that connects the game level with the user interface. At this stage the constructors in C++ are preparing Victoria crater, actors and scene components to reduce Loading time. Unreal creates the references from C++ to Blueprints at the beginning of the game, even if the game starts from different levels. This means that every unprotected pointer caused a <u>hard crash</u> of the Game and Engine itself. Hard Crash means that

Unreal closes immediately without any warning and without saving any unsaved progress.

2. About – Simple widget to get the player idea of the controls in the game and what is the main approach to the software.
3. Loading – Simple loading screen to avoid the game freeze. The freeze was seen while the "Start Button" was pressed, which fired the action event to load the main level of the game – Victoria Crater. The game was freezing for a particular time to load the stage, and any player unfamiliar with this could close the program instead of wait.

6.1.1 Main Menu workflow description



Fast Template Successfully Created

Figure 13: Representation of Main Menu Blueprint

On the screen capture, we can see various events. This blueprint is constructed when the "Widget Ready" Event is fired from the Level Blueprint **(Figure 14). "**Begin Play" firstly creates the widget (the object) and setting up Mouse Cursor visible for the player, by getting their controller index: (Get Player Controller function gets the array of the controller type objects and by taking index 0, it returns the current player input in a single game mode). The end of the wire is "Widget Ready" event which is the beginning of the "Event Graph" in Main Menu Blueprint **(Figure 13).** Computing this Blueprint is slightly different. As the event fires the nodes are backtracking to get the variables from the previous components: Widget Ready -> Set Input Mode <- get the controller and focus on the correct button Gamepad users). Clicking the "Start" button first creates the widget blueprint and adds it to the viewport, then it waits 0.2s and loads the level. The delay is necessary to split two events: Loading the level and adding a widget to the viewport. Otherwise, the software would freeze too soon without a correct loading screen. About Button triggers, another event created in About_BP to add "About" menu to the player screen and remove the Main Menu blueprint. The game has to be Gamepad friendly so the events like this are necessary to keep the focus on the buttons. Removing previous Widgets from the tree is also important as keeping them under and switching overlays messes with "Widget to Focus" dependencies and provides unexpected functionality. The best example is that using gamepad player was able to go to the About tab, but as soon as he wanted to go back to the previous screen, the game started. The problem was that "About_BP" was added to the viewport on top of "Main Menu_BP" and "Widget to Focus" had two buttons assigned: Start and Back. Removing Main Menu and later recreating it doesn't take much computing power, but avoid situations like that.

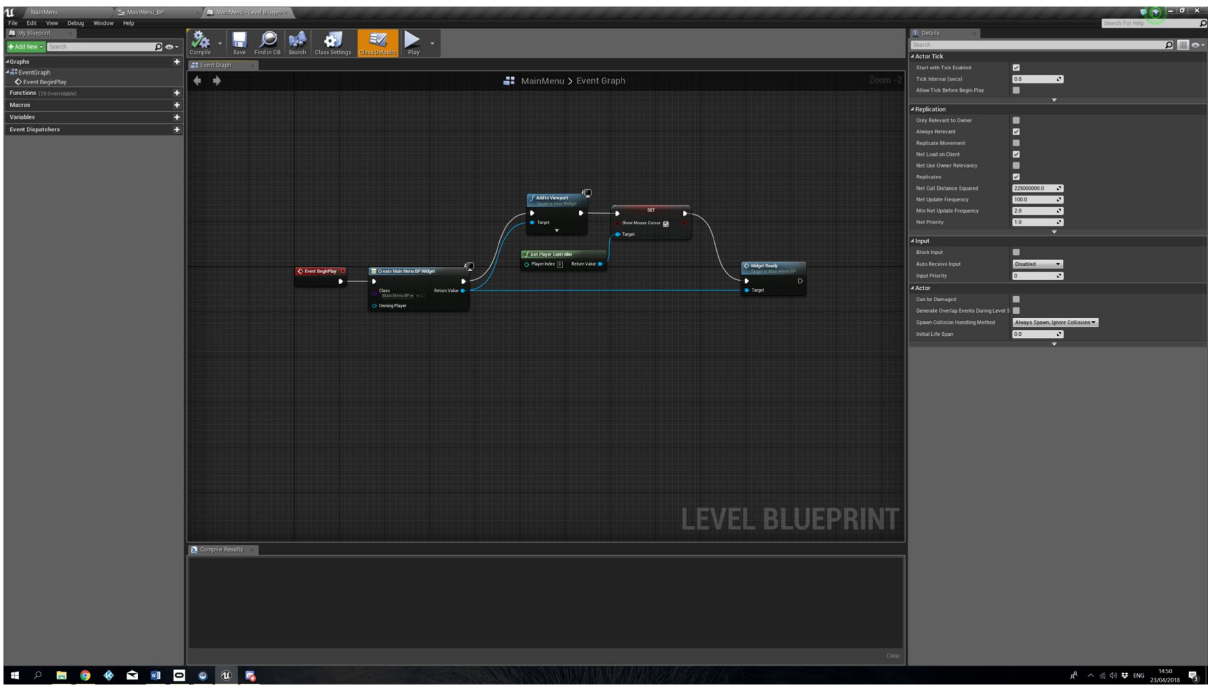


Figure 14: Main Menu Level Blueprint

About button adds the simple widget to the screen as described above and redirects the user to About page where he can find useful information.

Quit button closes the software.

<u>6.2 – Audio</u>

Audio for Main Menu and core game has been downloaded from YouTube **[11][12].** Audio is free to use as long as the source is specified. In the Unreal Engine 4, there is a special way of music implementation. It has to be in the WAV 16-bit format and only this format is supported. To play the music, the audio file has to be converted into the engine to the CUE format. Cue is a blueprint representation for Audio files **(Figure 15)** and all I had to do is create a loop for each audio, so it plays through the whole gameplay instead of only one time. The audio cues are bind to the level blueprints and Begin to play events.

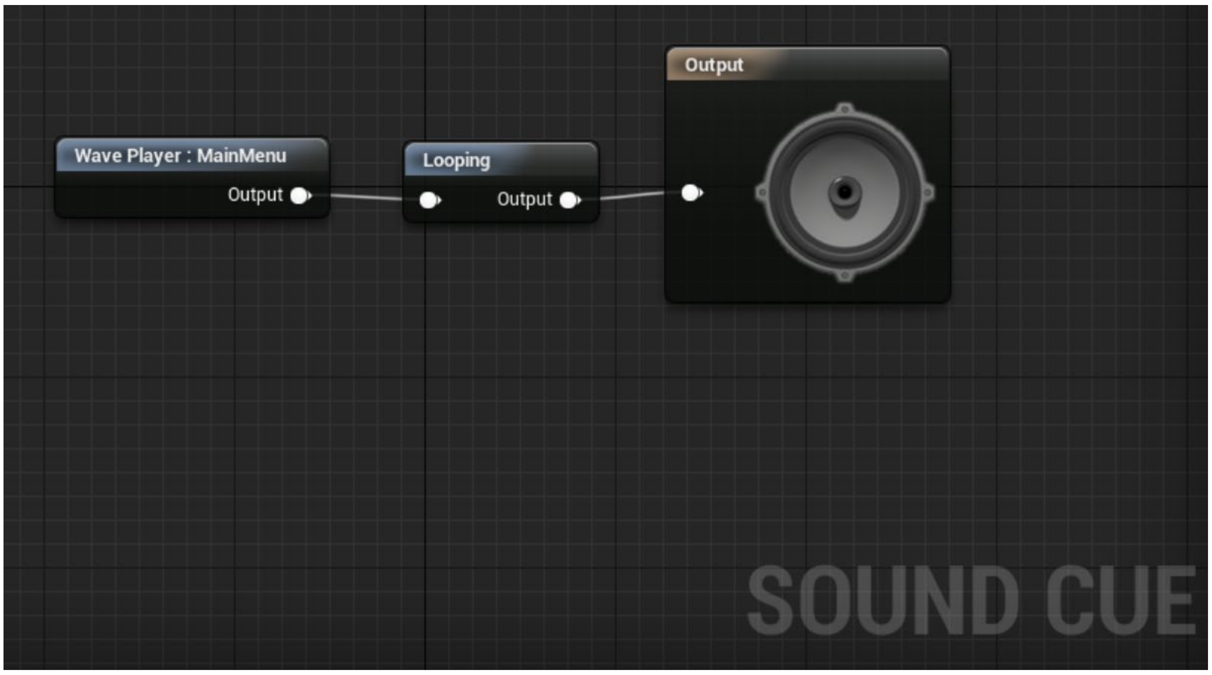


Figure 15: Sound Cue – Audio Blueprint

<u>6.3 – Controls</u>

Controls in the game have been designed to work on two types of controllers. One of them is "keyboard and mouse", and the second one is "Gamepad". The gamepad can be on Xbox or Ps4 as both of them are working. It is highly recommended to use Gamepad while the game runs in VR. The Keyboard controls **(Figure 16)** are designed to meet the gaming standards of input binding. Popular W, S, A, D is the movement input and E, Q is the flying input. Flying controls only works when the player possessing the flying character so he is able to change the altitude. **R key** is resetting the Rover position in case of collision bug. Mouse simply rotates the camera around the Rover or flying character and also rotates flying sphere mesh, so forward vector is always the same, otherwise pressing the forward key, could result with various directions of movement. Gamepad controls **(Figure 17)** are also simple. The left thumbstick is responsible for any type of movement, while the right thumbstick rotates the camera, triggers are moving up and down, and face buttons are properly described in the Figure below. The **R1** key is able to reset the rover position in case of collision bug.

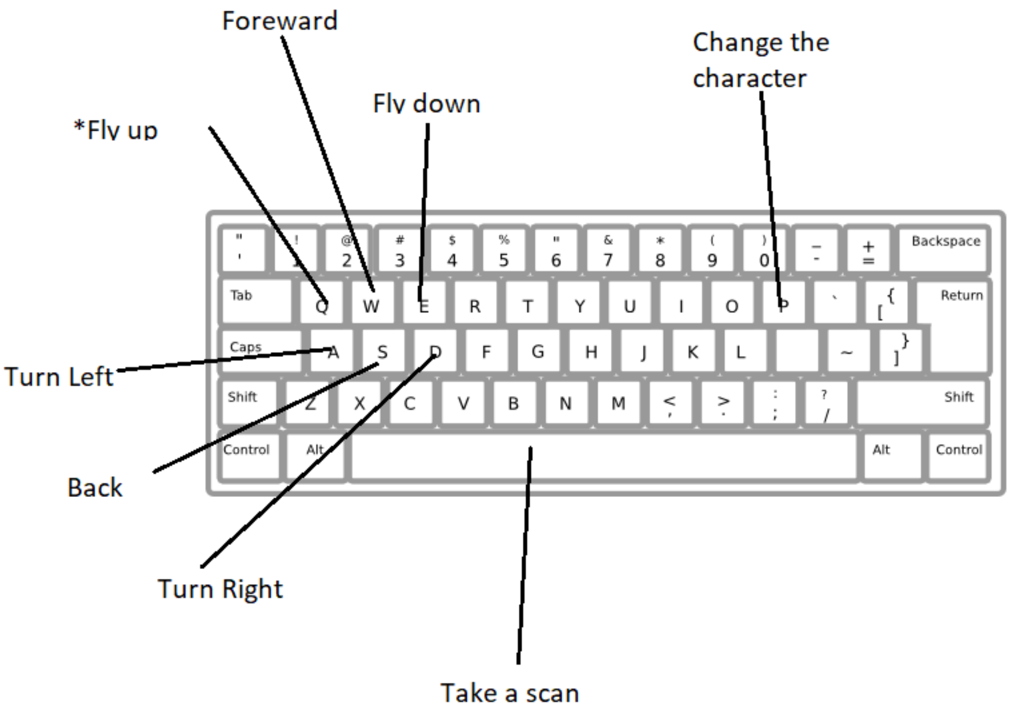


Figure 16: Keyboard controls

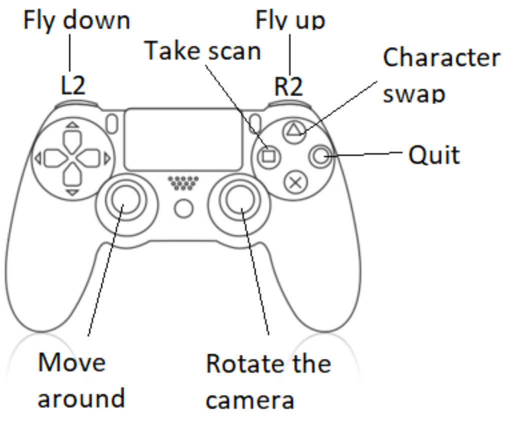


Figure 17: Gamepad controls

6.4 User Interface

User Interface is simple with many different functionalities behind the scenes. The main idea is to visualize the view from the camera, so there are two lines and a middle dot to represent it. It is placed 33% from the top border of the screen and from this dot, as a virtual reference the engine produces Line trace and check the collisions **(more in the chapter: Methods and Unreal Coding Standards).** The Distance value is also taken from the C++ and it measures the distance from the rover to the point of the interest. There is no possibility to measure the distance to the Sky and objects too far away. The interface changes the color from black to blue, and back when the rover is ready to take a Scan for Nasa. In the editor, it executes the function through the command line to take the photo, but in the final version of the game, the command line doesn't work. The indicator goes black for 10 second to store the data and image and then goes blue again, as it is ready to take another scan. Colour change and distance are two separate functions represented in the graph: **Figure 18.**

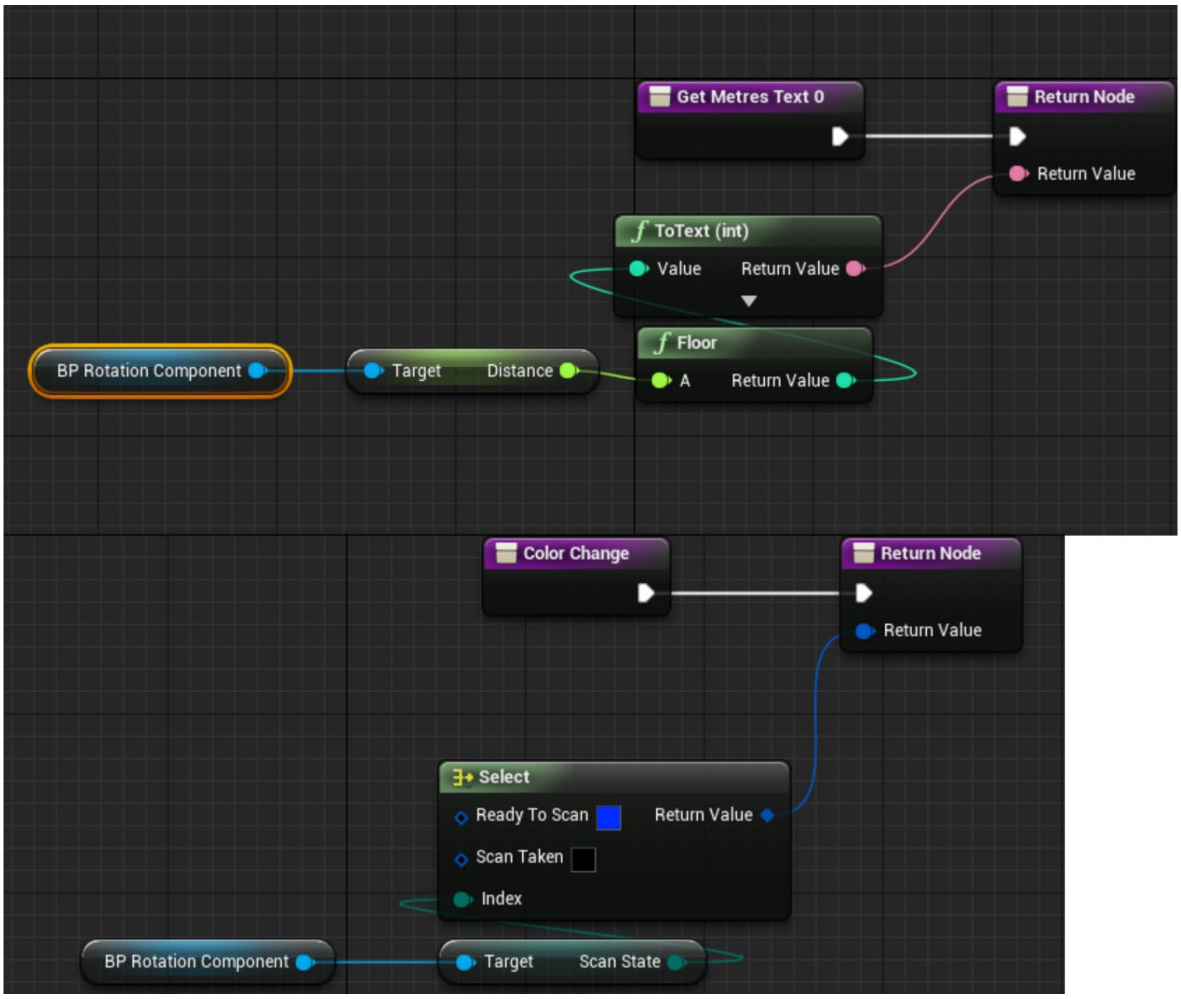


Figure 18: Color Change and Get Metres functions in Player UI

Two functions base on the information given from the Rotation Component – which is a C++ class, and Meters are a simple float value which is Floored to get a relational distance, and "Scan State" is an Enumeration defined in the Rotation Component with two states: Ready to Scan and Scan Taken. Each of this state has a different color assigned and depending on the state defined in C++ the color is pushed as a value.

6.5 Classes in C++ and Blueprint dependencies

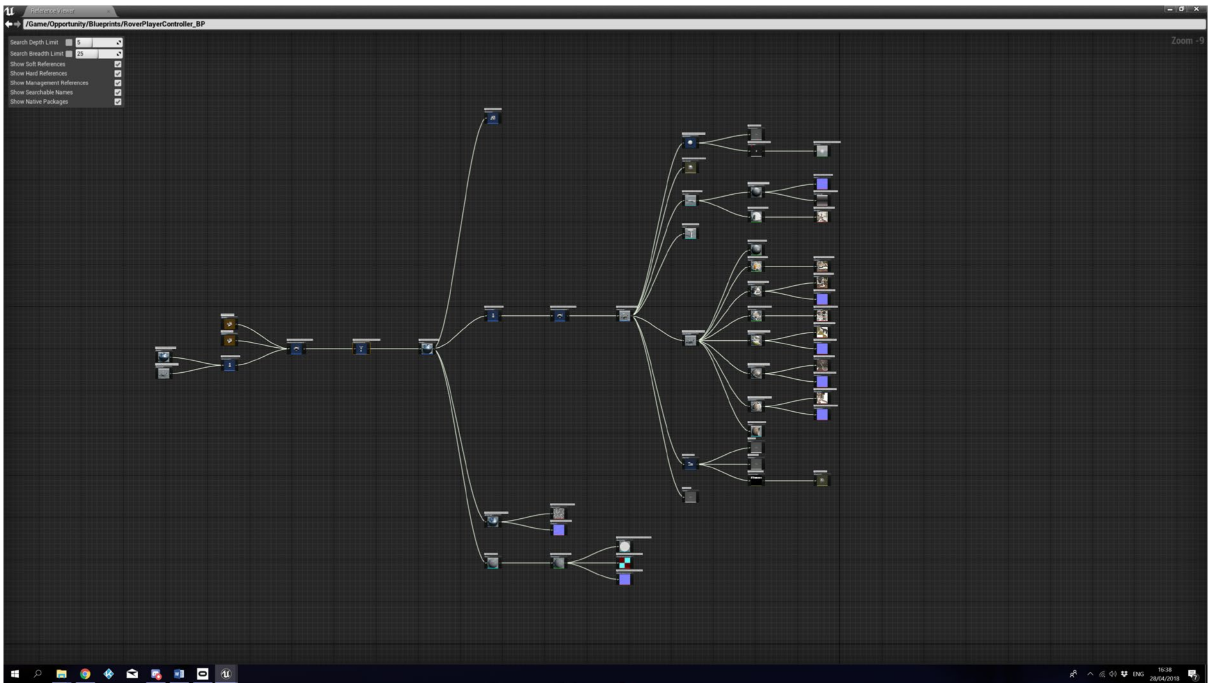
There are eight classes in Unreal (total of 9, but one is a simple spelling mistake and deleting the class in the engine is almost impossible. The mistake was done, when I was rushing with recreation after I lost all the project). Eight classes can be divided into five types, and each type is defined by the parent class:

- Player Controller type: Rover Player Controller
- Pawn type: Opportunity
- Actor type: Camera Flash
- Static Mesh Component type: Rover Camera, Rover Neck, and Rover Wheel
- Component types: Rover Movement Component, Rotation Component

The types have various functionality:

- Player Controller: The type of the class which defines the control system of the game, it is an interface between the Human controlling the Pawn, it represents the human will.
- Pawn: This is the physical representation of the human (usually an Actor) and can be controlled by Player or an AI entity within the world. The pawn behavior is a one to one representation of Human will or an Ai will.
- Actor: This is an object that can be placed into a game world. Actors can be destroyed, damaged, rotated scaled etc. There are various different actor types like Static mesh, camera actor, player start etc.
- Static Mesh Component: It is a component of a Static mesh that can be used to build the geometry. Four pieces of static mesh components are creating the Opportunity Rover.
- Components: Components are the functionality and can be added to the actors to give the particular actor access to the functionality provided by the component.

Unreal Engine provides the Dependency Viewer. It is a tool that maps the dependency flow across various elements **(Figure 19)**.



/Game/Opportunity/Blueprints/RoverPlayerController_BP

Search Depth Limit 5
Search Breadth Limit 25
Show Soft References ✓
Show Hard References ✓
Show Management References ✓
Show Searchable Names ✓
Show Native Packages ✓

Figure 19: Dependency Viewer for Rover Player Controller

The figure above shows the complex dependency system built from the Code, different blueprints, actors, and materials. The complex dependency system is not the best to visualize the core system so below you can see the simplified architecture of C++ Classes **(Figure 20 and Figure 21)**

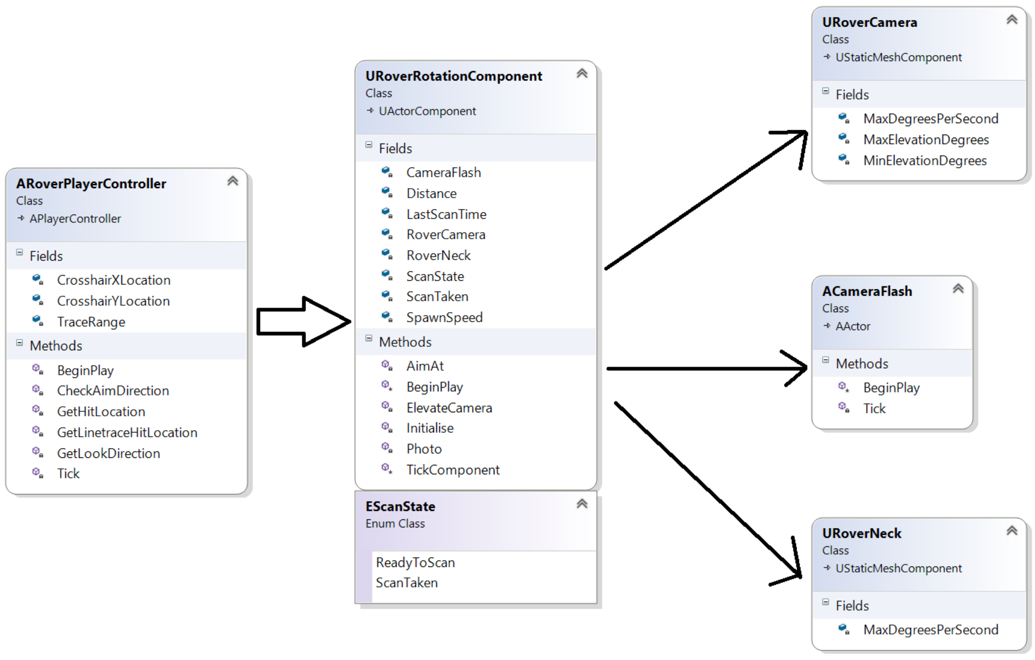


Figure 20: Rover Rotation Component – after refactoring process

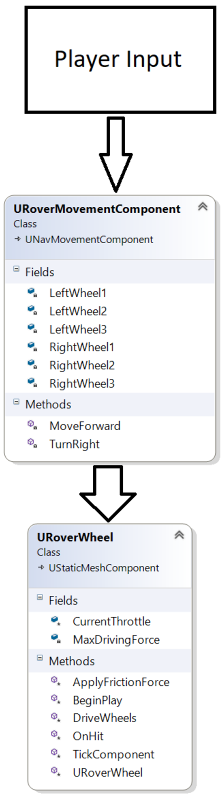


Figure 21: Rover Movement Component – after refactoring process

Blueprint Dependencies:

The subsection above described types of the classes used in the project showed their hierarchy and explained each of them. The Blueprints are using this classes to inherit their functionality over the game. "Opportunity_BP" is a Pawn type of blueprint, and the player can "posses" it to drive around the Victoria Crater. The parent class of this Blueprint is an Opportunity class, and the abilities to move, and function are taken from Rotation and Movement component. To create a dependency between C++ and Blueprint itself, the properties and methods have to be visible for the Engine. The variables and functions have to be indexed by the following macros:

```
UFUNCTION(BlueprintCallable, Category = "Setup")
        void Initialise(URoverWheel* LeftWheel1ToSet, URoverWheel*
LeftWheel2ToSet, URoverWheel* LeftWheel3ToSet, URoverWheel* RightWheel1ToSet,
URoverWheel* RightWheel2ToSet, URoverWheel* RightWheel3ToSet);
```

It means that method "Initialise" is visible for the Blueprint to assign the Rover Wheel type specified by the class URoverWheel to pass the reference from the editor to the code **(Figure 22).** Wheel initialization goes through the Movement Component, so in the blueprint, there is a "Target" value, which in that case are the components, and each Initialise function requires defined parts to assign defined by their targets. Wheels, rover camera, and neck are based on the same named C++ classes and declaring class pointers in the initialization method require the same type components in the blueprint. We are not able to assign the neck in a place of the wheel and apply force to it as the types don't match.

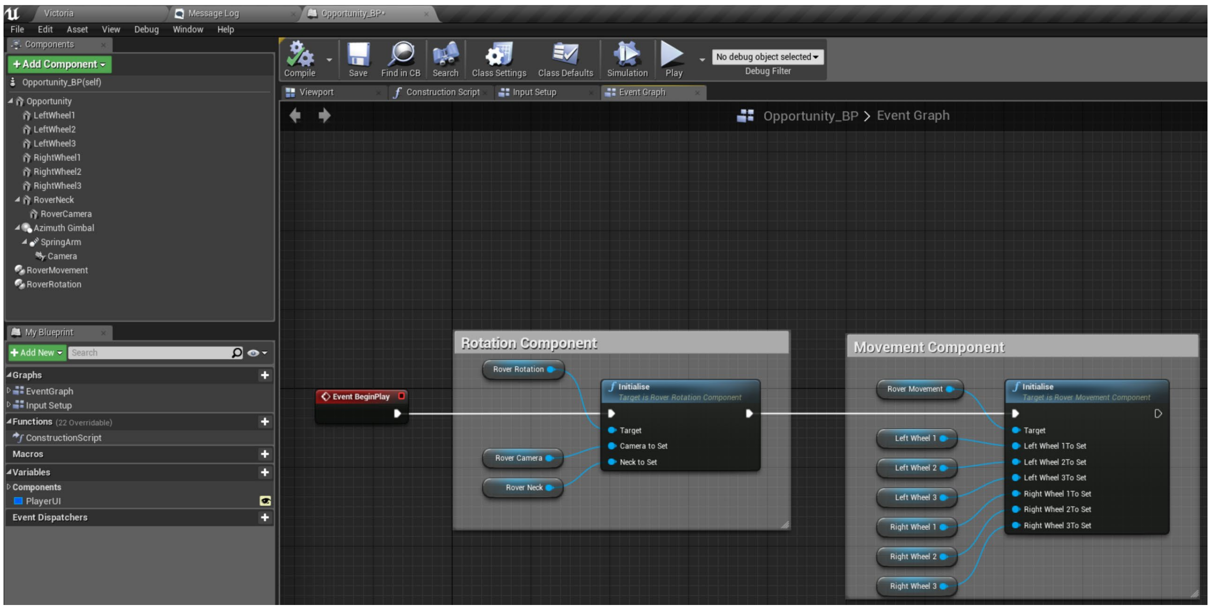


Figure 22: Opportunity parts initialization.

Adding the component based on the specific class is possible by making the whole class "Spawnable" and add it to the component list **(Figure 23)**:

UCLASS(ClassGroup = (Custom), meta = (BlueprintSpawnableComponent))

class MARSMISSIONDISS_API URoverWheel: public UStaticMeshComponent

ClassGroup: Defines the name of the group where the component appears

Meta: defines if it's spawnable, readable

Static Mesh Component with the mesh specified inherits all the collisions and structure from this mesh. It gives the ability to define the functionality of the mesh in C++ code and "Add" it in the Viewport tab **(Figure 24)**.

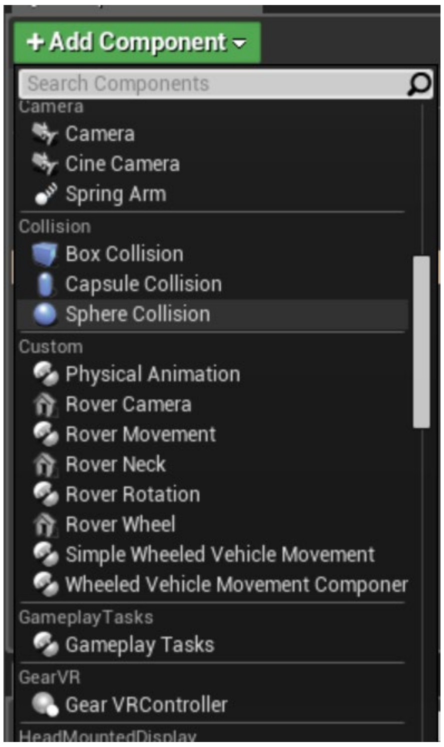


Figure 23: Adding the components

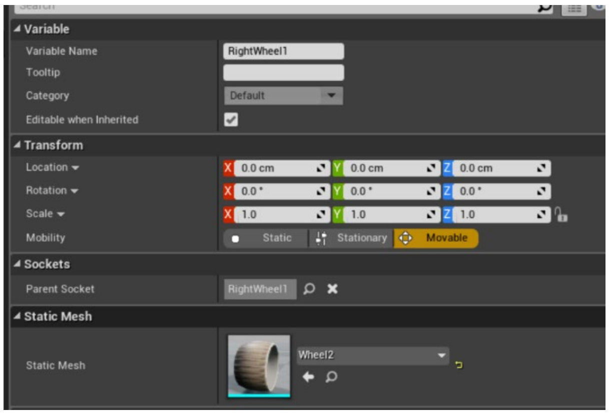

Variable
Variable Name: RightWheel1
Tooltip:
Category: Default
Editable when Inherited: ✓
Transform
Location: X 0.0 cm, Y 0.0 cm, Z 0.0 cm
Rotation: X 0.0 °, Y 0.0 °, Z 0.0 °
Scale: X 1.0, Y 1.0, Z 1.0
Mobility: Static, Stationary, [✓ Movable]
Sockets
Parent Socket: RightWheel1
Static Mesh
Static Mesh: Wheel2

Figure 24: Mesh representation in the game

Description above clearly shows the importance of blueprint implementation in Unreal Engine 4. The engine is highly dependent on the scripts and it is required from the developer to use it. Referencing the objects from the blueprint to the code gives the designer flexibility to change things and customize them for the gameplay. Developers can specify which properties can be modified by using following macros:

UPROPERTY(EditAnywhere) – Allows to change the value of the following variable in the blueprint and adjust it to the gameplay purposes. Different values can be assigned to different actors spawned in the World based on that class.

UPROPERTY(EditDefaultsOnly) **[16]**– Allows to change the value as well, but changing this values is possible only in Blueprint and every Actor spawned in the World has the same properties.

Refactoring Process:

Refactoring is needed when the class structure is not clear and various improvements can be made to speed up the process or make it more efficient. When the project was theoretically finished the class hierarchy was a mess and the components (Rover Rotation and Rover Movement) were added during construction.

It is not the best way of adding the component using C++ because if there is any other Actor in the game that can use the same type of movement (i.e. Rover and Sphere in the project), you have to create the class in C++ and initiate the pointer to that particular movement component, and add it to the constructor. It leads to the stack of useless classes created only to inherit various (movement, or actor, or any kind) components. The old classes architecture was also a mess **(Figure 25).**

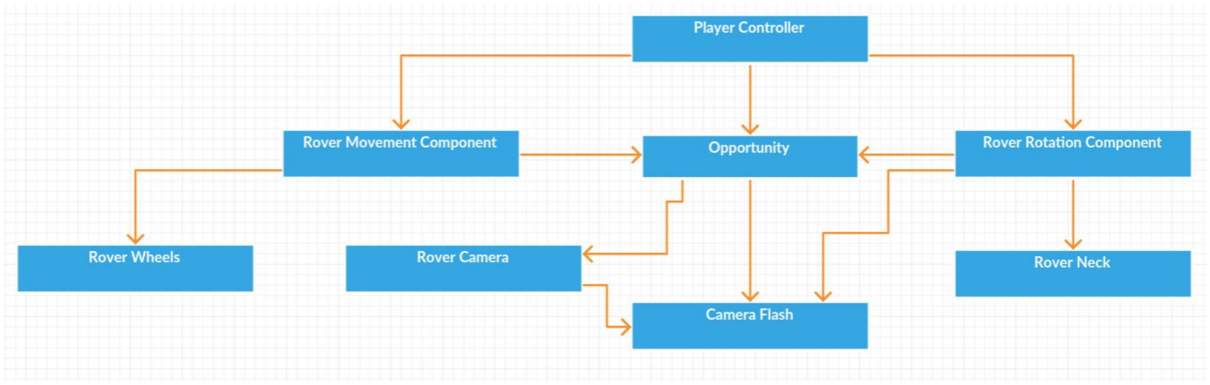


Figure 25: Class Architecture before Refactoring Process

The first step of the refactoring was to find a good method to follow. The described Red->Green->Refactor**[13]** method was perfect for this task. The second step was to identify the problematic parts – in that case, most of the references were going through Opportunity class and most of them were going through the components. It was obvious to choose the components as connectors and try to remove Opportunity class from the dependencies as the components could be added like shows **Figure 23** above. Removing Opportunity class and passing Rover camera dependency from the opportunity to rover rotation component result with the new structure shown in **Figures 20 and 21.** The new solution is clear and easy to understand, it removes useless pointers from flying between the classes and reduces the field of unexpected crashes. The current structure is even better because the Component functionality can be implemented into the next actors without touching the code, creating new classes and new dependencies. The best example of this process is already shown in the game: Flying character is the same type of blueprint like Opportunity. The flying sphere inherits the same movement, which means that the keys and input binding is identical, so from the player perspective it avoids confusion and helps to get more fun while playing the game.

6.5.1 Unreal construction and initiation – Constructors and Begin Play.

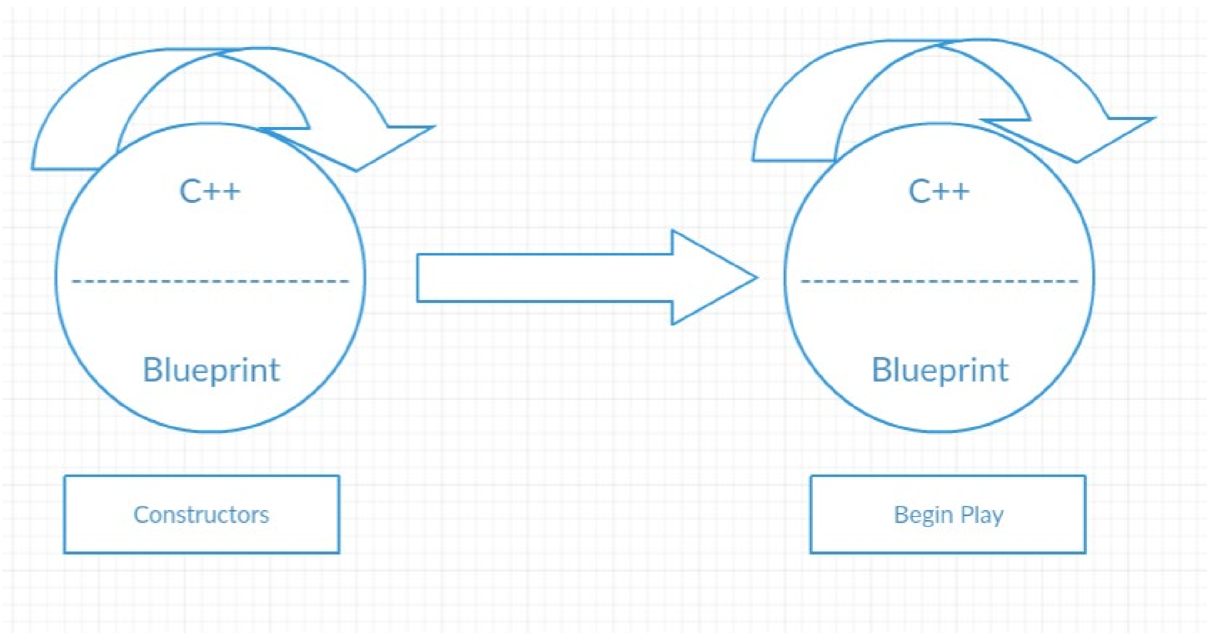


Figure 26: When things are happening – Constructor, and Begin Play

Refactoring process helped with structure and hierarchy of the classes, but creating standalone components led to various bugs in Unreal Engine editor. For unknown reasons editor wasn't able to find the references and initiate the corresponding variables in blueprints to the components. PlayerUI_BP takes the information directly from the Rotation component like Distance and scan state, but after refactoring the variables changed to NULL. The solution was provided by investigating when the things are happening **(Figure 26).** The diagram above shows the order of creating and calling the thing in Unreal Engine. **(The investigation process will be presented at the Final Demonstration).** The investigation was relatively simple. To know when things are actually created and then called during the gameplay, we are using the keyword like: "BULL" and then printing the string, so it's easy to filter the output log using the keyword. We have two types of constructors and two types of Begin Play. One is in Blueprint and the second one is in C++ code. To know which order UE4 has, constructors have been printed separately to the begin play. The result was impressive as the constructors have been called every time when we play the game or when we place something to the word. C++ code is first, then the Blueprint. When the game starts first it calls the Construction (C++ and BP), then Begin Play (C++ and BP). Removing Components from the constructor and adding them to the component list created the bug of Null pointers between the Blueprints as the variable hasn't been initialized in C++ or blueprints. The problem appeared only on Rotation component as various blueprints in the game are using the reference to that component, while Movement component only gives functionality to the rover and the references are initialized in time. To solve the Rotation component initialization Unreal provides special macro: Blueprint Implementable Event. Creating function in C++ header with this macro doesn't require the code implementation in the **.cpp** file. The implementation goes through Blueprint itself and initializes it **[17]**. The function appears in the blueprint as an Event, and fires the implementation for the variable, and further passes it to the PlayerUI_BP:

```
UFUNCTION(BlueprintImplementableEvent, Category = "Setup")

void FoundRotationComponent(URoverRotationComponent* RotationReference);
```

It creates the reference in C++ to the component, and then the reference is set in the UI blueprint by passing it through from the code **(Figure 27).** To make sure that this event will happen before the blueprint initialization, C++ code fires it up from the Begin Play method. According to the diagram represented by **Figure 26**, Begin Play in C++ is called before the Begin Play in Blueprint, so the variable in UI **(Figure 18),** has already the value.

```
Super::BeginPlay();

auto RotationComponent = GetPawn() -> FindComponentByClass <URoverRotationComponent> ();

if (!ensure(RotationComponent)) { return; }

FoundRotationComponent(RotationComponent);
```

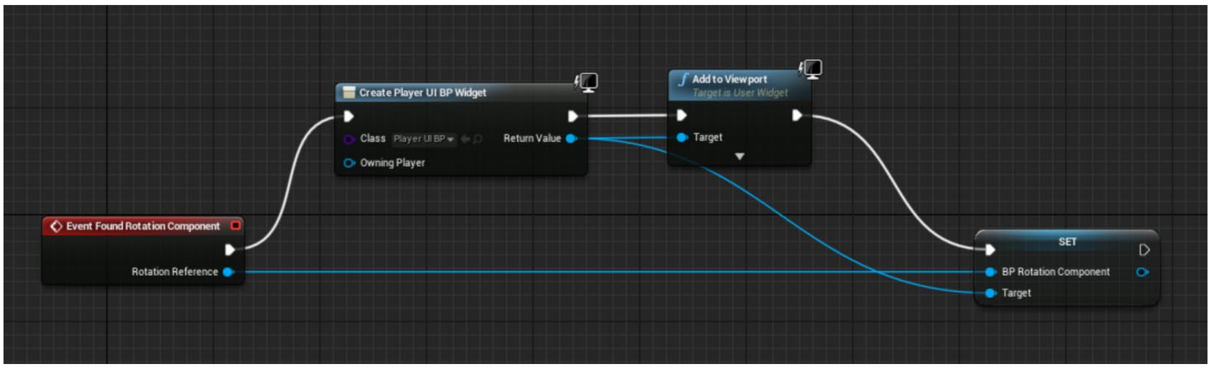


Figure 27: Found Rotation Component event in blueprint

6.7 Methods and Unreal Coding Standards

Unreal Coding Standards **[9]** had a high impact on the overall code structure. The standards are defined by the Epic Games and explain all the prefixes used in the code. Mars Mission Control has 8 classes divided into different types and only this types will be described. Prefixes **[9]**:

- **U**: Classes that inherit from UObjects
- **A**: Classes that inherit from AActor
- **E**: Enum Classes – there is only one Enum in Rover Rotation Component and it is a strongly typed Enum, with "New coding style" indication.
- **F**: it is a typedef of the structure
- **Int32**: for signed integers
- **Auto**: this prefix should be used, but in the cases where it appears in the code it has a function of detecting the variable type, and it is easier to understand instead of TPair<Generic, int>& Kvp: MyMap
- **UPORPERTY** – It is a Macro defining the variable to be shown in Blueprint. The new coding standards are also applied
- **UFUNCTION** – It is a Macro allowing the function to be visible in the blueprint. The function can be called and take parameters directly from the blueprints.
- **Forward Declaration [22]:** All the classes using it as it is recommended by the Unreal Wiki to avoid circular dependencies.

Methods in C++ and functionality description:

The description of the methods will be divided into two parts if necessary. First one is the functionality description, the second one is the description of blueprint connection (if there is any) and how the blueprint cooperates with the C++ method.

Class: Camera Flash **Children:** CmeraFlash_BP

This class is an empty parent class of Camera Blueprint. The class was created to give the functionality to the flash in the game. The basic concept was to get the Flash after the spawn and assign the velocity from Rotation Component to create a visual effect of spreading the light or light travel. In the end, the effect wasn't working with the particle effect.

Class: Camera Flasj **Children: -**

Typo mistake

Class: Opportunity **Children:** Opportunity_BP

The main concept of this class was to apply the core functionality to the rover, and before refactoring process according to the **Figure 25,** Opportunity was the biggest class in the project and carried over many functionalities. Although, there was a better way to create the functionality and implement it independently of the rover as described above in the refactoring chapter.

The class does not carry over any functionality of implementation to the blueprints.

Class: Rover Camera **Children:** Rover Camera Component

The class description in meta tag (BlueprintSpawnableComponent) means that every class indicated with this tag can be added in the blueprint through "Add Component" tab **Figure 23**. The class has 3 properties describing max elevation degrees, max degrees per second and minimal elevation degrees. There is one method Elevate: **void Elevate(float RelativeSpeed);** - The method is called, but is not doing anything as there is a problem with the models and insight Axis of the models. The method idea is to get the delta degrees as Relative Rotation and apply the difference to the camera, which will rotate it. The actual rotation is commented as it doesn't work properly, to see the result of the function simply uncomment the last line in the .cpp file and compile.

Class: Rover Movement Component **Children:** -

Movement Component class has 3 Methods and all of them are exposed to the Blueprint.

UFUNCTION(BlueprintCallable, Category = "Setup")

void Initialise(…) – The function takes 6 parameters for each wheel and assigning the pointers to the wheels.

UFUNCTION(BlueprintCallable, Category = "Setup")

void MoveForward(float Throttle) – Points to the Wheel Class and passes the throttle value, -value for the left wheels to actually move forward

UFUNCTION(BlueprintCallable, Category = "Setup")

void TurnRight(float Throttle) – Points to the Wheel Class and passes -Throttle value to each wheel as it is necessary for the rover to turn in the particular direction, otherwise Turning Right was actually turning left in the game.

The methods are called from the Blueprint and are bind to the Keyboard/Gamepad Inputs. The Initialise function takes the references to the wheel object from the blueprint (Opportunity_BP - Event Graph), and it starts at Begin Play. The pointers are protected by the ensure assertion **[18]**, as it is a better solution that simple if/print statement. It shows where is the problem and also returns the class name and exact line when something is not working. It also prevents the engine from the crash. Move Forward and Turn Right functions are initiated by the same name events (Opportunity_BP, Input Setup Graph). The Input events are only fires when the particular keys are pressed, otherwise, the event doesn't happen. The Inputs are assigned to the Input Axis events. It means that the values taken from the input (Keyboard 0 or 1, Gamepad 0 to 1) are the input values for the methods and further transferred and calculated as the force.

Class: Rover Neck **Children:** Rover Neck Static Mesh Component

Rover Neck is a similar class to rover camera. It has one property – Max Degrees Per Second to set the rotation speed of the neck. The function Rotate works exactly the same like function Elevate in Rover Camera class, but it takes Yaw parameter and calculates the difference. The same modeling problem appears here as well and the function is commented out as it is not working properly. To solve the rotation and elevation problem of two components I have implemented similar functionality in Opportunity Blueprint in the Input Setup graph. When the camera moves on the gimbals (Azimuth, and Elevation) **[19],** the blueprint checks the delta azimuth on the particular frame and rotates the neck to face the same direction like camera component.

Class: Rover Player Controller **Children:** Rover Player Controller Blueprint

Rover Player Controller class is a second largest class in the project it is responsible for the input and distance, it re-projects the screen to the world and finds the crosshair coordinates, and creates the line trace**[14]** to check the collisions and return Hit Location.

The Crosshair X and Y location is simply the middle dot visible on the User Interface while you are playing the game. X and Y are the coordinates on the screen to create a projection of that dot in the C++ at the same position.

bool GetLookDirection(FVector2D ScreenLocation, FVector& LookDirection) const;

The GetLookDirection takes a ScreenLocation as Vector parameter which is the Screen Size X * CrosshairXLocation and Screen Size Y * CrosshairY location. This calculation determines the position of the dot at the same position regardless of the screen size. Look Direction is an OUT parameter **[20]**, which means that the variable doesn't have to be initialized as the function returns it (not directly). The method is returning true or false dependable of DeprojectScreenPositionToWorld **[21]**, which creates the in-game version of our screen as well, or if it's unable to do it – returns false.

bool GetLinetraceHitLocation(FVector LookDirection, FVector& HitLocation) const;

the method above uses the Look direction to determine where the player is looking at this particular frame, it creates a start location on the player camera, and End Location, which is the Vector of start location + (LookDirection * TaceRange). The Trace Range is a property defined in a header file and it sets the range of the projected linetrace. The method base on the function LineTraceSingleByChannel **[14]**, it is the function provided by Unreal and it creates the line from the start point, to the end point and checks if there is a collision on the way. If there is a collision it returns true and initializes HitLocation variable with the value of the point where the line collides, if there is no collision it sets the HitLocation to 0 (to initialize the variable) and returns false - **Figure 28.** Hit Collision always has to be initialized here, otherwise, it will result in the bug of weird values shown as the distance.

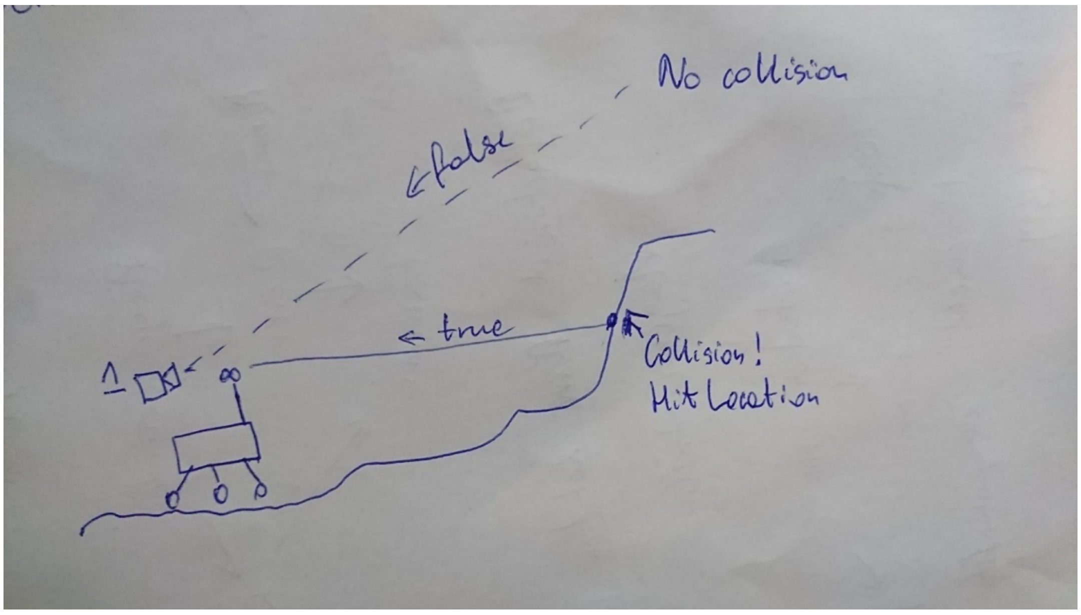


Figure 28: Simplified Linetrace Graph functionality.

bool GetHitLocation(FVector& HitLocation) const;

A function which uses previous two to recreates the screen and get the Hit Location. Returns true and the location of the git, otherwise false. This method is not complicated like the others as it simply uses previous methods to get the hit location variable.

void CheckAimDirection();

The most important method to actually get the HitLocation, calculate the distance and then push it to the Rotation Component. The Distance variable is a simple calculation of the distance between two points in the world – each point has to be a 3-dimensional vector. If there is the solution, the result is passed to the rotation component (HitLocation and Distance).

Class: Rover Rotation Component **Children:** -

Rover Rotation Component class is two classes in one, as it includes Enumeration class as well. The enum class is a simple two status class with ReadyToScan, and ScanTaken states to determine if the scan was taken or rover is ready to take one.

void AimAt(FVector HitLocation, float DistanceToSet);

The method called from the Rover Player Controller class. It takes the HitLocation and DistanceToSet, which is the distance value just to assign to the Distance variable in the class header file. The Distance is visible for the blueprint as it is the value displayed on the screen. The function uses SuggestProjectileVelocity **[22]** function from the engine, and it should move the particle effect alongside the line trace to give the effect of spreading light. Unfortunately, it wasn't possible, but the function was further used to elevate the camera.

UFUNCTION(BlueprintCallable, Category = "Setup")

void Initialise(URoverCamera* CameraToSet, URoverNeck* NeckToSet);

The Initialise function is the same as in Rover Movement Component. The difference is the type of the pointer assigned to the Camera and Neck **(Figure 22).**

void ElevateCamera(FVector AimDirection);

This method is responsible for calculating the delta rotation between camera and the Aim direction rotation – which simply is the line trace rotation. The calculated rotation is a 3D vector (Pitch, Yaw, and Roll), and the pitch value is pushed to the Camera Elevate method, while Yaw value is pushed to the Neck Rotate() method. The methods are not working properly as described above.

UFUNCTION(BlueprintCallable, Category = "Scan")

void Photo();

The function is called from the blueprint to spawn the Camera Flash Blueprint. It takes the location and rotation of the socket assigned to the Camera Mesh named "Flash" and spawns it there. It looks like the rover actually takes a scan of the environment around. When the scan is taken, the Last Scan Time variable is initiated to that time and wait 10 seconds for the next scan.

Class: Rover Wheel **Children:** Rover Wheel Static Mesh Component

Rover Wheel class is a C++ representation of wheels in game. It is responsible for adding the force at the particular location and simulate the movement. To determine when the force is applied the wheels generate Hit events. Hit events is an event when the wheel touches the ground so it prevents from adding the force when the rover is not on the ground **[23].**

UFUNCTION()

void OnTheGround(UPrimitiveComponent* HitComponent, AActor* OtherActor, UPrimitiveComponent* OtherComponent, FVector NormalImpulse, const FHitResult& Hit);

On the ground is the function which adds dynamics to the wheels. Basically every frame, the function checks if the wheel is on the ground and if is, allows to apply the force to it. The same thing could be done via Blueprint by using OnHit Event on the wheel mesh tab, but it is not that efficient and there is no way to control it, as the event always fires up when the particular circumstances are met.

UFUNCTION(BlueprintCallable, Category = "Input")

void SetThrottle(float Throttle);

Set Throttle is the function called from the Rover Movement Component. It receives the axis value as a throttle and clamp that value if it's higher than 1 or lower than -1, and assigns it to the Current Throttle value. Storing the throttle value is important as the functions are called every frame, so without resetting the throttle value, Rover would increase the speed every time when the button is pressed, and would never stop.

void DriveWheels();

Drive Wheels function calculates the force that needs to be applied, takes the component root location, which is the point on the ground **Figure 29,** and cast the static component to the primitive component. It is necessary as the Static mesh component doesn't have the ability to possess the variable like force. Casting it to the primitive component gives us the possibility to call function AddForceAtLocation() and actually apply the force to the wheels. The force is added to the Mesh Pivot point, which had to be modeled at the bottom of it.

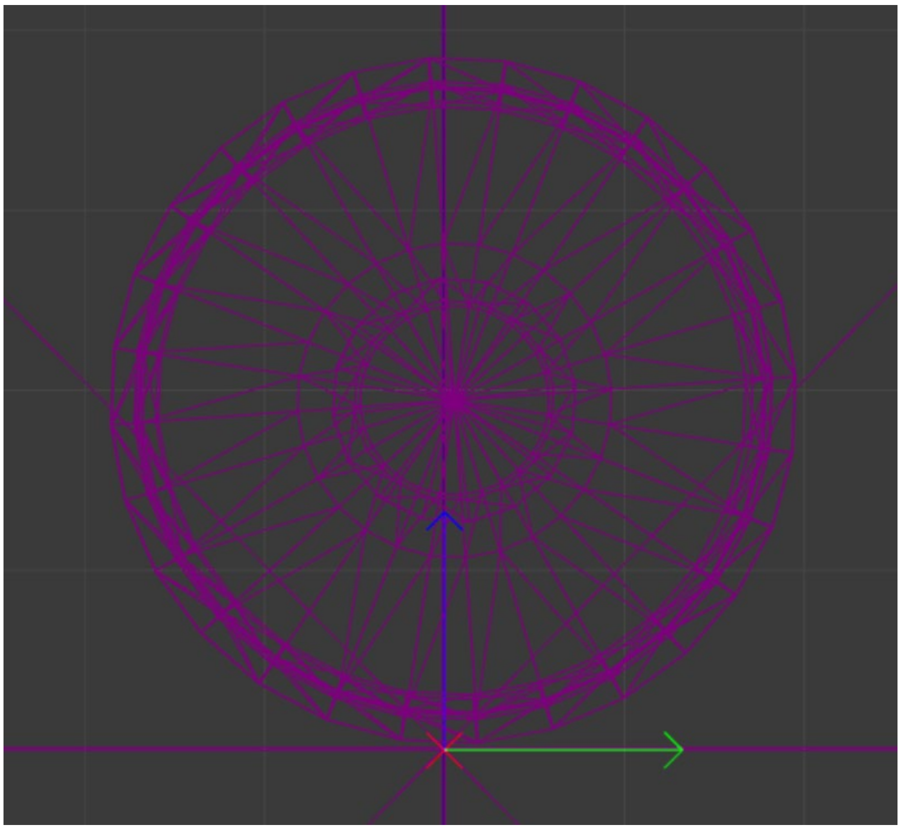


Figure 29: Wheel Pivot point location

void ApplyFrictionForce();

The Apply Friction Force function is responsible for adding the force vector simulating the friction and calculating it every frame. When the rover drives and turns Apply Friction prevents wheels from sliding on the sides. It gets the Forward vector (which faces the side of the wheel) velocity and calculating it using the Dot product function. Correction is a variable to store the value of the friction every frame with (-) to point the vector to the opposite direction. In the end, it gets the Correction velocity and multiplies it by rover mass to get the force, and applies it to the root, which is shown in **Figure 29.**

7. Rotation and Movement components

Rotation and Movement components are the components, which adds the core functionality for the Opportunity. This chapter will focus on this components and describe core solutions for the friction, components rotation and movement. Although camera rotation doesn't use any of this components, and the rotation approach is slightly different it is still a big part of rotation itself.

7.1.1 Camera Rotation [19][24]:

Camera Rotation has been developed from a single spring arm rotation because rotating the "Spring Arm" – the arm to which Camera is attached – caused a camera roll. This means that the camera was rolling right or left during the turn, and sometimes it finished with the upside-down view. The solution was to use the Gimbal system which rotates independently of the Opportunity. It means that even if the rover is upside-down, rolling down etc. the camera doesn't inherit the transform of the root. It is important from the VR perspective players, as previously rolling rover created a motion sickness effect. The gimbal is a simple "Scene Root" component which doesn't inherit the transform from the Rover and is responsible for the Azimuth aiming. The spring arm is a second gimbal itself and is responsible for the elevation. Both elements create the gimbal system of independent elevation, rotation, and azimuth.

7.1.2 Rotation Component:

The main concept of the rotation component was just to rotate the specified meshes in the rover. During refactor process the concept has changed and the component evolved to create a full "Aiming" system for the Opportunity. It changes the states of the Scan Status and communicates with the User Interface Blueprint to change the color according to the state and pass the distance value. In the blueprints, the component value is initiated at the beginning of the game as described in the previous section.

7.1.3 Applying Forces and Movement Component:

Movement Component is responsible for getting the input value as a Throttle and pass it through to the wheels. It also initiates the particular components as correct wheels. The force is calculated by the simple equation – Force = mass * acceleration **[25]** and applied. In the Unreal Engine the default units are cm and kg, so first, the force has to be transformed from cm to m and then calculated. Force is applied in the "Pivot" point on each wheel. The pivot has to be located on the ground to apply the force correctly **(Figure 29).** The problem of the axis is shown in **Figure 30,** where we can see the Forward Vector (X-Axis) and Right Vector (Y-Axis).

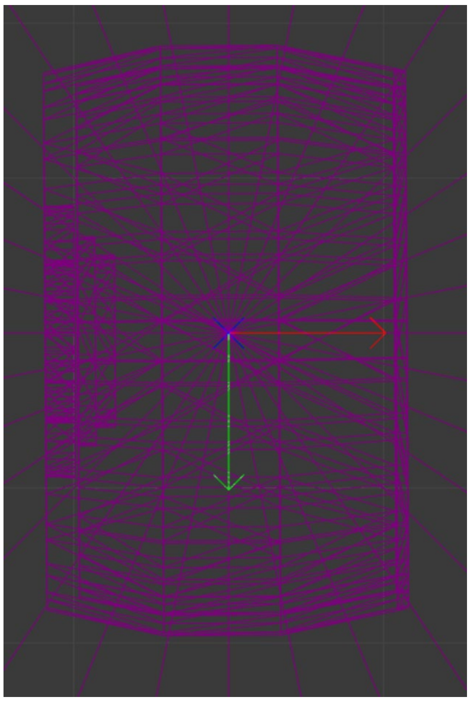


Figure 30: Top view of the wheel and Axis (Green – Y, Red – X)

The force vector is aligned with Y axis, and it is always pointing the same direction.

7.1.4 Friction Control:

The friction force is calculated every frame to apply it correctly. The problem was visible every time when rover turned while driving forward – it was slipping on the sides, and that problem was destroying all the immersive feeling of the game. The friction is calculated by Dot Product **[26].** The dot product is the angle and distance between two vectors in the coordinate system. In Wheel case to get the Slippage speed, the calculation was done between the Component speed and Forward Vector (Alongside X-Axis). Correction force to actually get the force needed to avoid slipping is the calculation of Correction – The velocity vector in the particular frame, but the opposite angle, and Mas of the rover. The result is presented in **Figure 31** below.

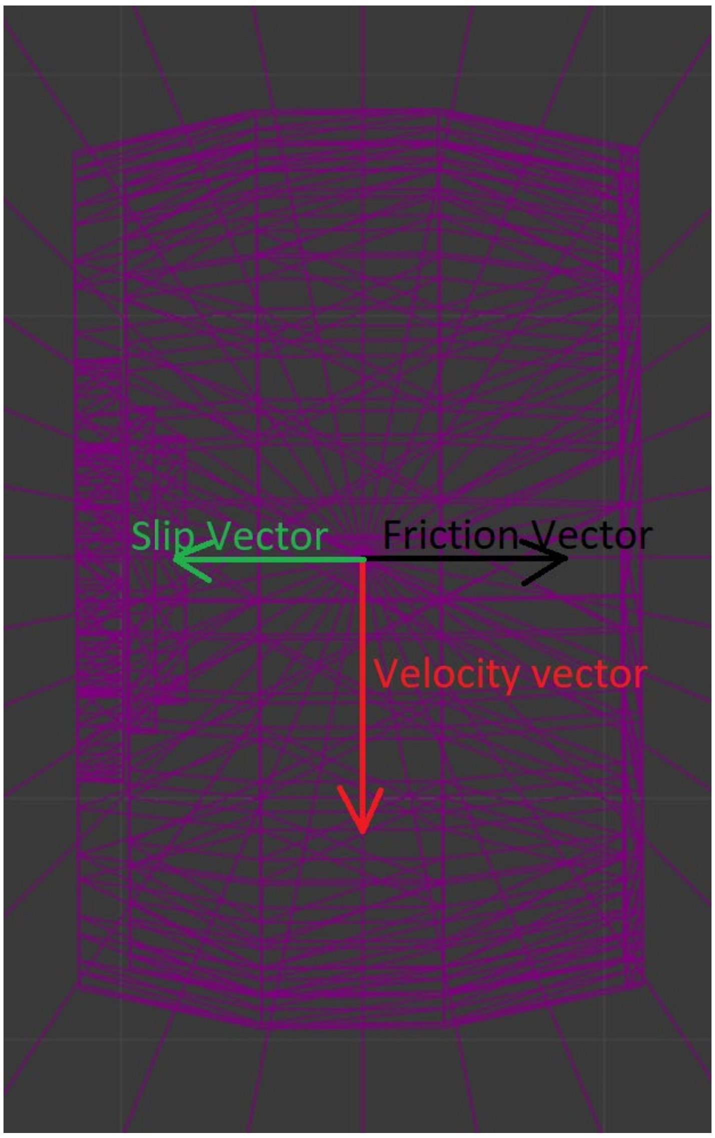


Figure 31: The Forces graph Slip Vector = Friction Vector

7.2 Character Swap

The idea of character swap was necessary to give the player possibility to fly around faster and to see everything from a different angle. The movement and control system is the same on flying sphere and Opportunity, which gives the comfort of remembering only one input setup. The flying sphere is perfect for VR players without motion sickness as it can fly up and down, so for some, it can cause the motion troubles. Character Swap operation was inspired by watching **Kyle Dail – "UE4 Blueprint Character Switching".** The tutorial series was really helpful with accomplishing the character swap task, although in Mars Mission Control not all of the solutions could be used. The main difference is the "Game State" and "Game Mode", the tutorial uses the Game State as the blueprint to assign the pawns to swap, in Mars Mission Control creating any other game states led to the Hard crashes of the engine. The next difference is using specified pawn types reference instead of generic ones – it avoids the confusion while passing the reference from different blueprints.

The process requires three pawns to use the same interface specified in the Blueprint folder. It helps to pass the message of the swap across the pawns. The process is relatively simple: Check which character is possessed -> get the characters camera and spawn the SwapPawn at the location of the camera -> find the second character -> transfer the camera to the location of the second one -> poses the character -> destroy the Swap pawn.

The animation of this is possible by adding the Timeline and transferring it over within one second.

7.3 Day and Night cycle, Sky creation

Day and Night Cycle has been created by using "Level Sequencer" Tool **[27].** The sequence is relatively simple: Rotate the Sun and Moon around the Pivit – which is the sphere located in the point (0,0,0). The rotation is the change of the location of the spheres, and rotation to adjust the light direction. It is set to definitive loop, so the process will never end. To the sequencer is added Atmospheric Fog, which is a simple effect of the foggy environment, and sequencer changes the value of it – the intensity of the fog: **Figure 32.**

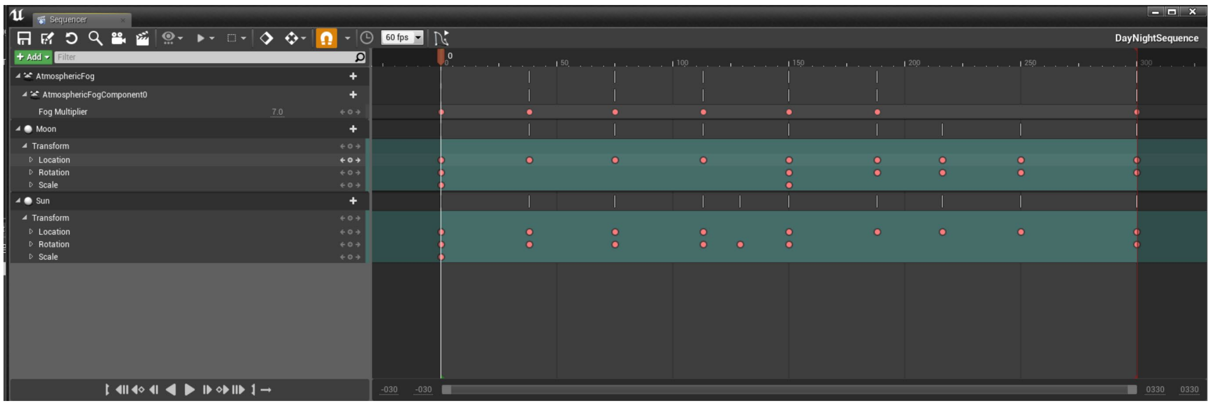


Figure 32: Day and Night Sequence

We can see that the sequence remembers the "new location" at the particular frame of it and between the remembered points it relocates the object. Selecting Sun and Mon will show the route of the objects during the full cycle **Figure 33.**

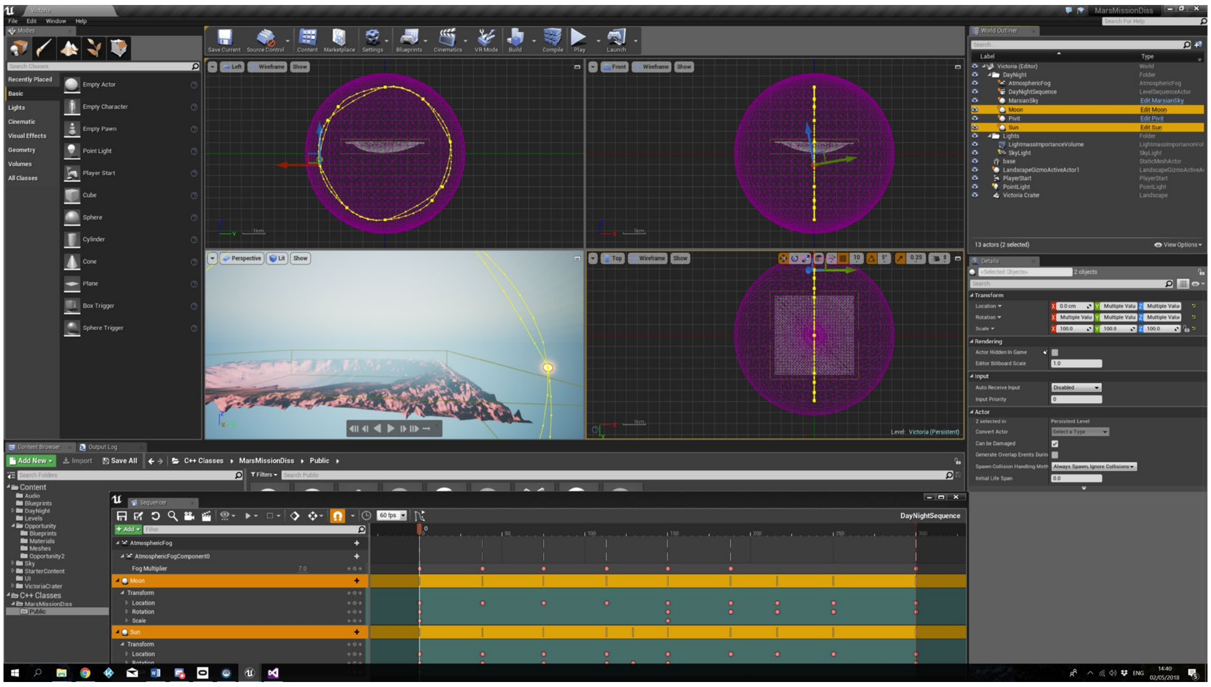


Figure 33: The Moon / Sun Cycle

<u>Sky Creation:</u>

The sky is a simple texture created in Spacescape **[28]** – Software created by Alex Peterson. The software gives a wide range of opportunities to create nebulas and skies. For this task, only basic functionality was needed to recreate similar Mars sky. The texture was imported to the UE4 and set for the lowest compression settings to sharpen the image. The skybox is a sphere with the material applied outside and inside, and stretched out, as the game level is on the inside of it. The material is relatively simple as it takes the texture sample and multiplies it by the constant variable to adjust the brightness and light intensity.

8. Tests and current bugs

Software Testing:

Game tests are different than software tests. The games are tested by the beta-testers, people who are playing the game and checking the stability of the game, and report bugs. The testing was done by myself – Lukasz Wrzolek, as during the development it was the best option to test the solutions. Unreal Engine is not the best tool for testing the software as it is really unstable and every mistake done in the code or blueprint causes a hard crash – the engine turns off itself. I have put many hours to play the game and check how everything works together and there are not many things that could be improved. The engine is a powerful and sensitive tool and doesn't forgive any mistakes, that's why usually people hire beta testers to find the problems and report them.

Current problems and bugs:

The final game is not free from various problems and bugs. The crucial and the most important problem is driving the rover across the Victoria Crater. The rover is a static mesh, so the wheels are not rolling and while the player drives around it can collide with invisible edges of the landscape. The Landscape is built from the grid of squares and I think that driving through the edge of the square can cause a collision with the mesh.

Spawning at the beginning of the game is the second bug. The rover always spawns facing the exit of the base, while the camera can spawn facing different points. The problem is that the camera is placed on the gimbals and it's an independent component of the spawned pawn. Although, the camera should always spawn on the generic position – behind the rover. Currently, there is no reasonable explanation for that bug and it has been reported to the Epic Games.

9. Critical Evaluation

Mars Mission Control is the project that represents Victoria Crater and allows the player to drive around/fly around – explore the crater. It is the representation of NASA Mars Mission and it allows to control the Opportunity rover. The main task of the project was to give the real representation of the crater with all the details and I think that I have accomplished it. The recreated crater has 800m diameter like original one, the shape is also the same, and by using tools provided by Unreal it has natural look. The waves in the middle of the crater have "Waving Sand" material applied to them to give the feeling of little wind traversing through the crater.

The time that I had to develop my skills in various areas of Coding, Modelling, Unreal Engine knowledge, was really short and the final version of the software is not perfect as well. I thought that modeling is pretty simple – and it is, but rigging the model, and exporting it to the engine is not. There are various problems with the models: Different engine axis to the world axis, different vectors – all of these parts had a crucial impact to the functionality, and even with applied solutions and overall success it could be done better. The Opportunity rover should be a real representation of the rover with working suspension – and with my current knowledge and time, I could accomplish it. To do that I have created the skeleton of the Opportunity and imported it into the engine, re-created the physics assets – to see that at the end it is not working properly. If I would have more time, I am sure that the final product would be more realistic than it is now.

Game development is hard and takes hours to create something really good, and it is definitely tasked for more than one person. People specialized in creating models, designing the levels, functionality designers, coders etc. all of them are the specialists in the areas that I am only learning about.

Mars Mission Control helped me to understand the software development process and helped understand how hard is creating the actual game. Looking behind I am able to see how big progress I have made to develop the final version of the application, and how hard it was. Learning how to create a simple 3D model to creating Opportunity, reminding how to code in C++ to advanced solutions in a final project, and finally learning more and more about the Unreal Engine 4.

Overall I am really satisfied with what I have created and achieved.

10. Bibliography

1. Epic Games - https://www.unrealengine.com/en-US/what-is-unreal-engine-4
2. UE4 License and Distribution - https://www.unrealengine.com/en-US/eula
3. Blueprints - https://docs.unrealengine.com/en-US/Engine/Blueprints
4. Visual Studio - https://www.visualstudio.com/
5. Dr. Laurence Tyler – Dept. of Computer Science, Aberystwyth
6. Sand Material – The inspiration how to create procedural Sand in Blender: https://www.youtube.com/watch?v=eDuArvDdRu4 – Crea2000
7. Christian A. Lopez - https://nasa3d.arc.nasa.gov/detail/spirit
8. Metal Game Studios - https://www.youtube.com/watch?time_continue=102&v=TTsAYpqHU-c
9. Unreal Coding Standards - https://docs.unrealengine.com/en-us/Programming/Development/CodingStandard
10. Official NASA website: https://mars.jpl.nasa.gov/mer/home/
11. Lofi Hip Hop: https://www.youtube.com/watch?v=DMyudUrFlQQ
12. Too Good for TV: https://www.youtube.com/watch?v=YluWsdMil5Y
13. Red, Green, Refactor: http://blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html
14. Linetrace:http://api.unrealengine.com/INT/API/Runtime/Engine/Engine/UWorld/LineTraceSingleByChannel/
15. OnTheGround Event: https://answers.unrealengine.com/questions/198710/c-if-an-actor-has-a-collision-hit-onhit-how-do-i-c.html
16. EditDefaultsOnly:https://answers.unrealengine.com/questions/92344/upropertyeditdefaultsonly-editassetdetailsonly.html?childToView=93137#answer-93137
17. BlueprintImplementableEvent: https://www.youtube.com/watch?v=YrTn7xA-3jk
18. Ensure: https://docs.unrealengine.com/en-us/Programming/Assertions
19. Camera Azimuth and Elevation: https://en.wikipedia.org/wiki/Azimuth
20. Atheis91: https://answers.unrealengine.com/questions/289436/how-to-properly-define-output-parameters.html
21. Deprojection:http://api.unrealengine.com/INT/API/Runtime/Engine/GameFramework/APlayerController/DeprojectScreenPositionToWorld/index.html
22. SuggestProjectileVelocity:http://api.unrealengine.com/INT/API/Runtime/Engine/Kismet/UGameplayStatics/SuggestProjectileVelocity/index.html

23. Ryanjon2040: https://answers.unrealengine.com/questions/198710/c-if-an-actor-has-a-collision-hit-onhit-how-do-i-c.html
24. Gimbal System: https://en.wikipedia.org/wiki/Gimbal
25. Force: https://www.wikihow.com/Calculate-Force
26. Kyle Dail – Switch: https://www.youtube.com/watch?v=R3KdTxkIwyU
27. Sequencer: https://docs.unrealengine.com/en-us/Engine/Sequencer/Overview
28. Spacescape: http://alexcpeterson.com/spacescape/