

Nguyen Bao

# Developing a Game Debugger with Unreal Engine 4

```
----- Game Debugger / Cheat Sheet -----  
Cheat Typing Instruction (no square brackets): [CheatCommand] [Value]  
Debug.AttackTimingBar [] _____ Debug toggle  
Debug.BlockingBar [] _____ Debug toggle  
Debug.Camera [] _____ Debug toggle  
Debug.Controller [] _____ Debug toggle v  
Debug.DodgeTimingBar [] _____ Debug togg  
Debug.Hitboxes [] _____ Debug toggle v  
Debug.LockingAlgorithm [] _____ Debug toggle  
Debug.Notifications [] _____ Debug toggle N  
Debug.RageMode [] _____ Debug togg  
Debug.RangedMode [] _____ Debug togg  
Debug.Sidestepping [] _____ Debug toggle  
Debug.Sprinting [] _____ Debug toggle
```

Bachelor's Degree in Business Information Technology

Game Programming

Autumn 2021



**KAMK • University  
of Applied Sciences**

## **Abstract**

**Author(s):** Nguyen Bao

**Title of the Publication:** Developing a Game Debugger with Unreal Engine 4

**Degree Title:** Bachelor's Degree in Business Information Technology

**Keywords:** game development, game programming, game testing, debugging tool, game debugger

The objective of this thesis is to build a game debugger tool for a game studio called Still Running, which is based in Helsinki, Finland. The tool is developed within Unreal Engine 4, along with their current project. For programmers, it needs to be robust and modular to effortlessly expand for more functionalities. For users (other developers and testers), it should be easy to understand and utilize the provided features in it.

Debugging is a challenging process to identify the bugs and fix them, which can potentially lead to a slower process of game development, as well as negative effects on the player experience. Therefore, the demand for a tool that can help to minimize those bad effects is essential, which enforce programmers to build a tool called a game debugger as soon as possible.

To support prototyping and developing this debugging tool, researches were conducted to give more insight into the game debugger, from definition to example. They also included many important concepts and ideas for this practical development of game debugger. The acknowledged study and information varied from elementary to complicated, which gradually improved the utility of this tool in each version.

Besides the console commands, the final result of this practical development is a functional and friendly debug menu, in which "players" can click on buttons to perform certain cheat actions that fit their needs. This game debugger is also modular, maintainable, and can be expanded as big as developers want without any frustrating process.

## Table of Contents

1	Introduction.....	1
2	Game debugger .....	2
2.1	Theory background .....	2
2.2	Meaning and purpose .....	3
2.3	Types of game debugger .....	4
2.4	Game debugger examples.....	5
2.4.1	Patapon 3 .....	5
2.4.2	Pokémon Emerald.....	5
2.4.3	The Legend of Zelda: Majora’s Mask .....	6
2.5	Technologies involvement for practical development.....	8
2.5.1	Unreal Engine 4 .....	8
2.5.2	Rider for Unreal Engine .....	10
2.6	Example of a built-in debugging tool .....	11
3	Principal concepts and frameworks in UE .....	13
3.1	Common parent classes .....	13
3.2	C++ and Blueprint.....	14
3.3	Console Commands.....	16
3.4	Cheat Manager.....	17
3.5	UE’s Reflection System.....	17
3.6	Unreal Motion Graphics.....	19
4	Implementation of Game Debugger.....	20
4.1	Integrating custom Cheat Manager .....	20
4.2	Forming custom commands.....	21
4.3	Printing list of commands .....	23
4.3.1	Command text and values .....	24
4.3.2	Command descriptions .....	25
4.3.3	Final printing .....	27
4.4	Transferring to UI menu.....	28
4.4.1	WBP_CheatMenuButton.....	29
4.4.2	WBP_FunctionButton.....	30
4.4.3	WBP_GameDebugger.....	31
4.5	Setup input .....	34

4.6	Result.....	35
5	Conclusion .....	37
	References .....	38

## List of Symbols

AI	Artificial Intelligence
BP	Blueprint – visual scripting system in Unreal Engine
C++	A programming language that is used in Unreal Engine 4
EQS	Environment Queries System
IDE	Integrated Development Environment
UE	Unreal Engine
UHT	Unreal Header Tool
UI	User Interface
UMG	Unreal Motion Graphics
WBP	Widget Blueprints

## 1 Introduction

Game development is a complicated and intensive process of building and publishing games, which brings a joyful and immersive experience to players. Depending on the type of game, developers can have many different goals to plan and accomplish during the creation stage. This process may take up to several years, and potentially require hundreds of people to get involved, based on the company's scope and budget. Many development roles are fundamental in the game industry to achieve that, such as Programmer, Artist, Designer, Sound Engineer, Producer, Tester, etc. [1]

It can be said that it is a satisfied and joyful feeling whenever a game team finishes their passionate product and presents it to the players. For each individual, this achievement could be rewarded with a substantial improvement of knowledge, skills, and experience. For studios or companies in general, they could gain a significant increase in profit, with the possibility of enhancing their popularity and reliability. However, during the challenging development stages, many obstacles can appear and negatively affect the pace and the outcome of those games.

That being said, there is still a solution to completely get rid of them, or at least minimize most of the bad effects. Therefore, the aim of this Bachelor's thesis will concentrate on developing a convenient and useful system, called a game debugger. This systematic tool is built for a game studio in Helsinki, Finland, which is named Still Running. It is being utilized for their working-in-progress title in Unreal Engine 4.

To achieve the goal, some research is made to give a better understanding of the situation, then bring out optimal techniques for this practical development. They will have a more in-depth analysis of the theory, along with some visual examples about the game debugger in different games. Furthermore, an example of debugging tool in a game engine is also mentioned.

Before the implementation part, the thesis will make sure to go through every principal concept and idea. This would offer a clearer vision of what should be done in order to build a game debugger. Therefore, the content of the implementation part could be easier to understand and follow along.

The chosen topic is interesting because of its various benefits. For the individuals who develop the game debugger, they can gain more valuable knowledge and experience. For the company, they can boost their development process much faster and more efficiently.

## 2 Game debugger

### 2.1 Theory background

In programming, one of the hardest activities for programmers is debugging. According to Paul Knickerbocker [2], the process of eliminating faults and unexpected behavior in software is called debugging. In general, the nature of programming can produce many small issues, which rise from typing mistakes or unintentional consequent of designing choices. The more complicated the program, the more difficult it is to identify and fix the bugs.

If those bugs and errors are not removed from the game, they can have a significantly bad impact on the player experience. There are many types of bugs that a game can produce under certain circumstances. The worst problem is crashing, which makes the game unplayable, as it can freeze or shut down the entire system completely. Another example could be a door that is not opening, when it is supposed to open for the player to proceed to the next level. There are also minor bugs that are worth mentioning, including collision issues (Figure 1), glitching visual, broken sound, illness motions, etc. [3]



Figure 1. Dog collision bug in Fallout 4 [4]

It could be argued that debugging may take up enormous time and efforts, which potentially slow down the whole development process. As a result, game studios could essentially need a certain tool to support this irritating process, which is the game debugger.

This is not necessarily required for every company to have, especially for the ones with small-scale games. However, developers of bigger-scale products could be expected to build a game debugger, which would be more beneficial in the long run.

## 2.2 Meaning and purpose

In this specific topic, a game debugger (as known as a debug menu, or a debugging tool) is a hidden part of a game that allows developers and testers to perform inaccessible actions, which are not supposed to exist in the final shipping product. It contains simple menus with various buttons for the users to select, according to what they prefer (Figure 2). [5]



Figure 2. A debug menu from Super Smash Bros. Melee. [5]

Versions with debugging options attached are considered debug versions, which are not exposed to the public. Generally, most of them are removed by developers in the published version. However, there is a risk that players can still access this tool by hacking or taking advantage of system glitches and leaks. [5] Hence, it is also considered that game studios could take more advanced security into account, as they should not let the end-user or player get control over it. People with bad intentions can ruin the whole gameplay and possibly make a significant loss of profit for the game. Nevertheless, this kind of subject is not covered in this thesis, since the main objective is still about researching and developing the game debugger tool.



Technically, programmers are responsible for building and developing it, then utilizing it for their workflow. Designers can also take advantage of this tool to brainstorm a new idea, while testers can playtest with it, make sure the game mechanic and feel fit the design, and report bugs if available.

Consequently, game debugger plays an important role in making sure the game product works perfectly as intended, especially in bigger game companies. The earlier this tool is acquired, the more efficient the development process will become.

### 2.3 Types of game debugger

Technically, there are 3 crucial types of debugger menus involved in game development. The first one is Crash Debugger, which is only available while crashing the game by performing a certain combo of keys or buttons. This debugger reveals principal statistics of the game about the player, level, lives, etc. In addition, it can also show the version of the game as well as which date it was made. [5]

Secondly, Gameplay Debug Menu (or Debugging Console) is a tool that can affect the gameplay based on what developers want. It could be modifying the character's health, or changing gravity in a level, etc. [5] This thesis will also concentrate on developing this kind of debugging tool, with both console (Figure 3) and menu versions.

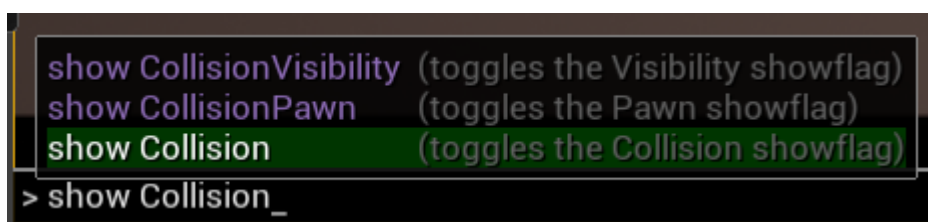


Figure 3. A screenshot showing collision commands in Unreal Engine's console

Finally, the System Debug Menu can produce adjustments in the game. Every cutscene, sound effects, and animation test are included in it. The game version and its published date are also no exceptions. [5]

## 2.4 Game debugger examples

### 2.4.1 Patapon 3

One of the examples that a game debugger can get accessed with a hack cheat code, is from a game called Patapon 3 (Figure 4). It is noticed that their debug menu is enormously big, with various debugging options, such as adding items to inventory, unlocking and enabling features, setting levels for units, adjusting skill experience, completing all missions, toggle debug camera and sound, changing weather, controlling units, etc. [6]



Figure 4. The debug menu in the field in Patapon 3 [6]

### 2.4.2 Pokémon Emerald

The next example is Pokémon Emerald, which has many debugging options taken from the English Ruby and Sapphire version. Its debug menu includes 8 pages leading to other submenus, mostly with Japanese (Figure 5). Although there are some selections in the menu with unknown mechanics, the remainders still offer powerful commands (Figure 6) that can significantly change the gameplay. [7]



Figure 5. Trainer card in Pokémon Emerald [7]

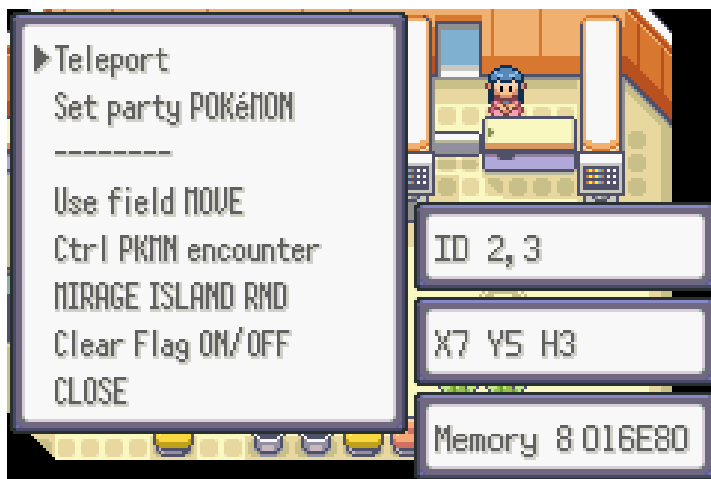


Figure 6. One of the debug menu in Pokémon Emerald can teleport the player to last visited Pokémon Center [7]

### 2.4.3 The Legend of Zelda: Majora's Mask

Another one is The Legend of Zelda: Majora's Mask, which had a leaked debug version in 2010 (Figure 7). This version contains the whole debugging features used by the game testers. Moreover, there are a diversity of features, including general functions, map select, inventory editor, memory editor, camera editor, environment editor, audio debugger, event editor, and console output strings. [8]



Figure 7. Debug title in The Legend of Zelda: Majora's Mask [8]

In addition, this version of The Legend of Zelda also has a test map (Figure 8), which is not typically reachable from the map selection. This map is the same as the normal map, but the contents are fairly different from each other. [8]



Figure 8. A test map in The Legend of Zelda: Majora's Mask [8]

## 2.5 Technologies involvement for practical development

This chapter will briefly cover the essential technologies for building a custom game debugger for the project. Besides a game engine, a coding tool will also be utilized for programming the mechanic and features of the debugging tool.

- Game engine: Unreal Engine 4
- Coding tool: Rider for Unreal Engine

### 2.5.1 Unreal Engine 4

Unreal Engine 4 (UE4), which is developed by Epic Games, is a well-known and powerful game engine for developers to build games. It is completely free to use and also has big documentation and community. One of the reasons for its popularity is the multi-platform development support including PC, mobile, and consoles. Moreover, UE is being utilized by many modern and famous games, such as Fortnite (Figure 9), Rocket League, etc. [9]



Figure 9. Fortnite, Epic's battle royale game [10]

UE is utilizing C++ for its core programming language. Competent programmers can write C++ code in their scripts to develop their game within the engine. For more amateur ones, they can

use Blueprints (Figure 10), which consist of multiple visual and interactive blocks of code to connect with each other, to perform a similar game-building process without having to be good at traditional text coding. [9]

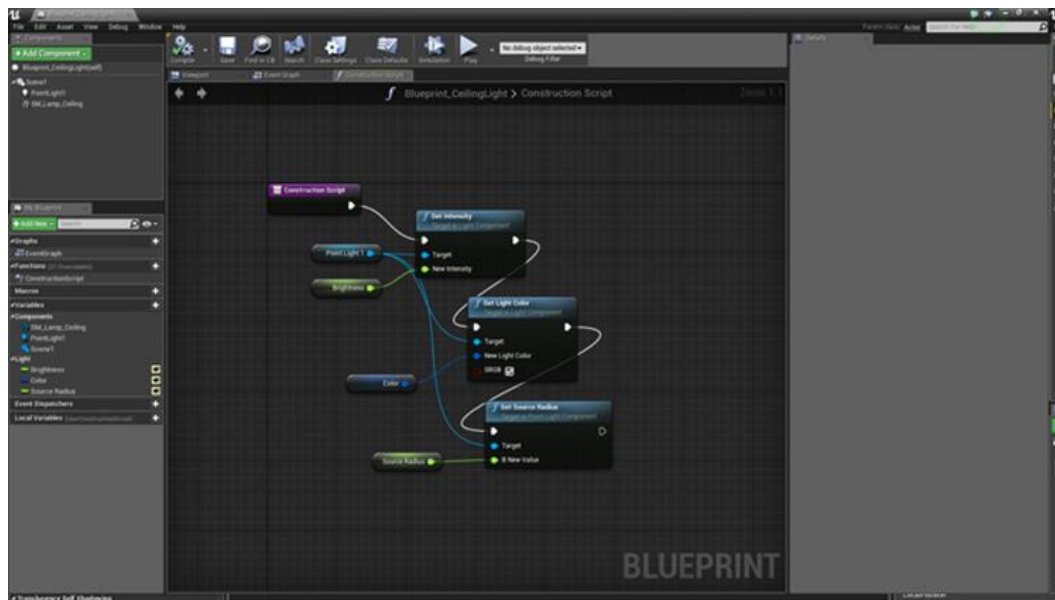


Figure 10. A Blueprint window in Unreal Engine 4 [9]

For artists and designers, UE also has various tools to modify UIs, materials (Figure 11), animations, lightning, particle effects, etc. They could be complex and intimidating at first, with various properties and parameters to adjust. However, with proficient experience, they are extremely powerful and efficient to achieve what developers are focusing on. [9]

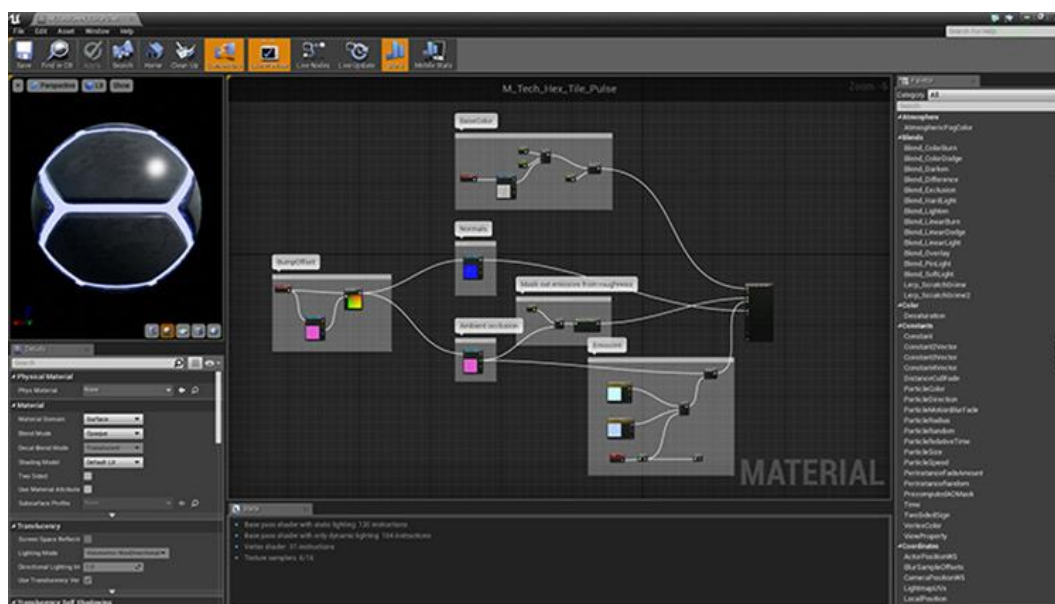


Figure 11. Material editor in Unreal Engine 4 [9]



For those reasons, Unreal Engine 4 is chosen for this practical development. In addition, it has numerous features and concepts to support building the game debugger more advanced and efficient. They will be mentioned and explained in detail in the further part of this thesis.

To support the production of this tool, many of the research sources are from Unreal Engine 4 Documentation (Figure 12). They are proficient, reliable, and straightforward to the basic information. However, for more in-depth pieces of knowledge, some tutorials and blogs are needed to improve the tool, make it more advanced and better for programmers to develop.

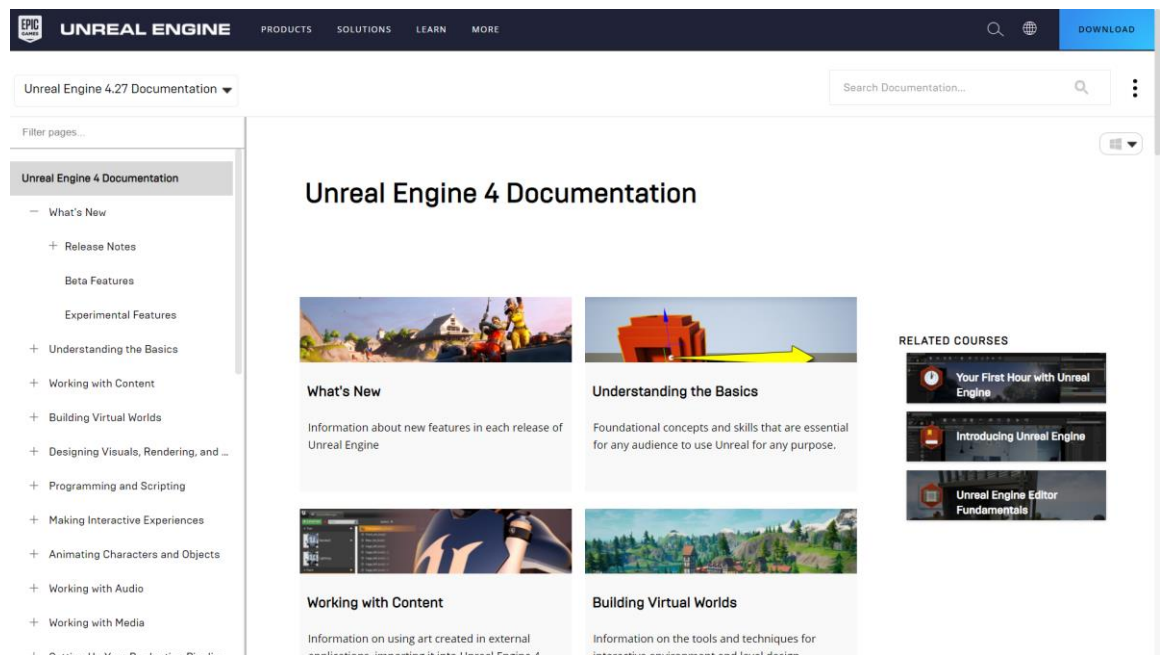


Figure 12. Screenshot of the Unreal Engine 4 Documentation website [11]

### 2.5.2 Rider for Unreal Engine

Rider is an IDE with a powerful and fast performance when working with Unreal Engine and Windows C++ development. It offers the developers the possibility to see Blueprint's data. Moreover, Rider can assist with coding style and correctness, as well as UE's reflection mechanism. [12]

An integrated development environment (IDE) is a software program consisting of necessary tools for programmers to build other applications with code. It primarily has a code editor for writing and editing source code, with a compiler to convert those codes into a form that can be executed by the computer. Moreover, the IDE also includes its own debugger, and some build automation tools to save developing time. [13]

As for now, Rider for Unreal Engine (Figure 13) is at the Early Preview stage, so game developers can use it for free until it is released. There are various categories of Rider to help to develop a wide range of applications. Besides Unreal Engine games, it also offers the production of Unity games, along with desktop applications, libraries, web applications, etc. [12]

```

38     UPROPERTY(EditInstanceOnly, Category=Brewery)
39     TWeakObjectPtr<AStrategyBuilding> RightSlot; ⚙️ "Wall_EmptySlot_Brewery_C_4" (Brewery_C_2)
40
41     /** cost of each requested spawn */
42     UPROPERTY(EditDefaultsOnly, Category=GameConfig)
43     int32 SpawnCost; ⚙️ "50" (Default_Brewery_C)
44
45     /** initial resou
46     UPROPERTY(EditDef
47     int32 ResourceIni
48
49     /** initial resou
50     UPROPERTY(EditDef
51     int32 ResourceIni
52
53     /** initial resou
54     UPROPERTY(EditDef
55     int32 ResourceIni
56
57     /** delegate to broadcast about finished wave */

```

The context menu is open over line 43, showing options: Generate, Definitions, Constructor, Copy and move operations, Getters and setters (highlighted), Missing members, Overriding members, Equality operators, Relational operators, Stream operations, Hash function, and Swap function.

Figure 13. Rider IDE with Unreal Engine support [12]

## 2.6 Example of a built-in debugging tool

This subchapter will briefly look into an existing useful tool for debugging in one of the most popular game engines: Unity.

Utilizing a debugger of Unity offers the ability to inspect the C# source code while the game or application is running. Although different code editors can be paired with Unity, they all share the same functionality such as breakpoints, single stepping, and variable inspection. [14]

Programmers can set up breakpoints (Figure 14) and attach the code editor to the Unity Editor. Some code editors can offer the developer to choose which instance of Unity for debugging. When everything has been set, enter Play Mode in Unity Editor to test the breakpoint. The debugger will stop when the code at the breakpoint is executed. At that time, programmers can



check the variables step-by-step. The Unity Editor will temporarily freeze until the debugger is continued or stopped. [14]

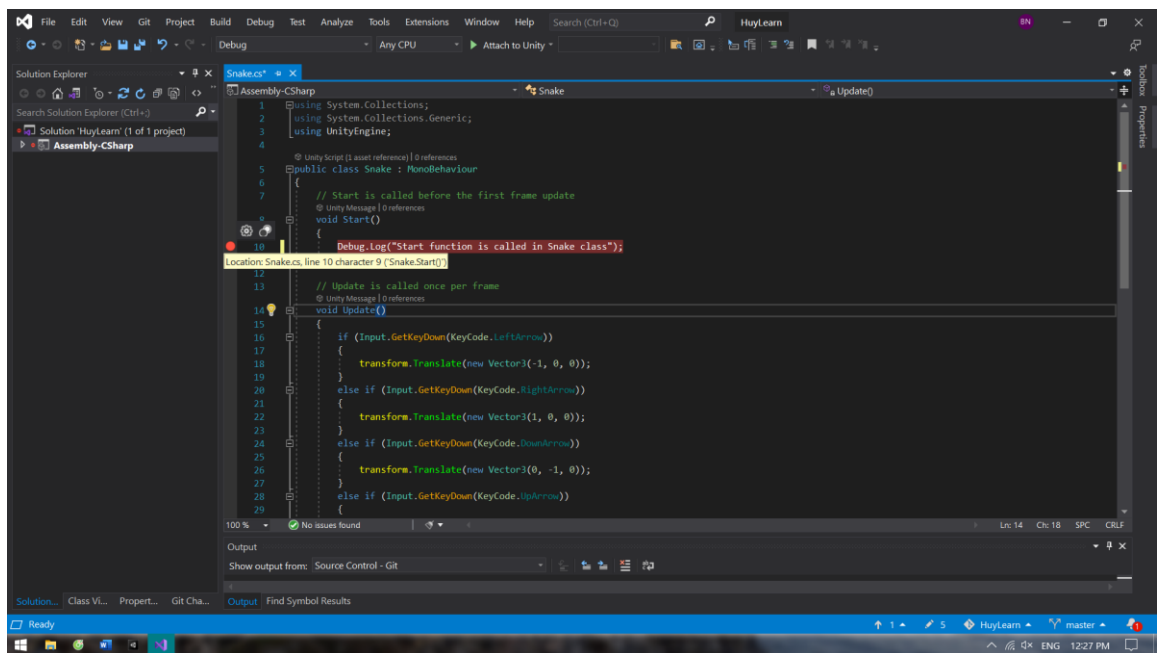


Figure 14. Setting a breakpoint in a line in Visual Studio, which is also an IDE. This line also prints the message in the Unity console

### 3 Principal concepts and frameworks in UE

This chapter will cover every vital idea for developing the game debugger in UE and Rider. However, it does not include any installation instruction of them, or explanation of basic theories in general programmings, such as variable, function, class, etc.

#### 3.1 Common parent classes

First of all, a brief clarification of some built-in UE classes is necessary. It introduces essential fundamentals in UE, so it would be easier to proceed to the next concepts which may be more advanced to understand.

When creating a new class with C++ or BP, developers need to choose the parent class. By specifying that, they are allowed to inherit properties from the parent, then use them in their new class. Figure 15 shows the most common classes in UE. [15]

Class Type	Description
<b>Actor</b>	An Actor is an object that can be placed or spawned in the world.
<b>Pawn</b>	A Pawn is an Actor that can be "possessed" and receive input from a Controller.
<b>Character</b>	A Character is a Pawn that includes the ability to walk, run, jump, and more.
<b>PlayerController</b>	A Player Controller is an Actor responsible for controlling a Pawn used by the player.
<b>Game Mode</b>	A Game Mode defines the game being played, its rules, scoring, and other faces of the game type.

Figure 15. Most common classes in UE, described by UE documentation [15]

As for GameMode, it can be set for each level, or reused in multiple ones. In addition, developers can change the default GameMode class in the Projects Settings in UE. Each level can also have its GameMode overridden, as shown in Figure 16. [16]

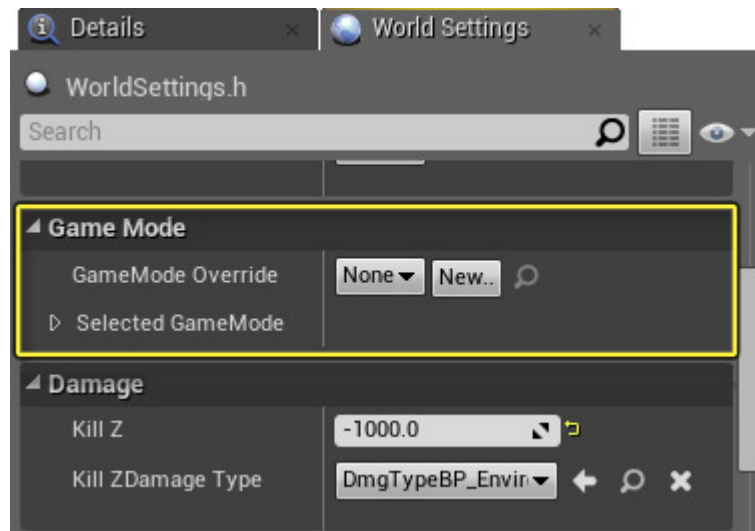


Figure 16. Overriding GameMode in the World Settings tab [16]

After the game starts, GameMode, Pawn (or Character), PlayerController, etc. objects will be spawned. There are options to modify which class should be used for them in the GameMode details panel. [16]

### 3.2 C++ and Blueprint

In addition to C++, UE also has another system for programming, which is called Blueprint Visual Scripting (BP). It is utilizing a node-based interface in the editor to construct gameplay logic. Programmers can set up a C++ base class with UE's reflection system, and then extend it with BP to expose a wide range of functions and properties to the editor. This powerful and flexible system offers the designers the virtual usage of many concepts and tools, which are usually only available to programmers. [17]

In basic words, BP contains blocks of code, where developers can simply drag and drop to connect them with each other, in order to build the mechanic of the game. Figure 17 demonstrates how a typical BP looks like with EventGraph, variables, tabs, etc. [18]

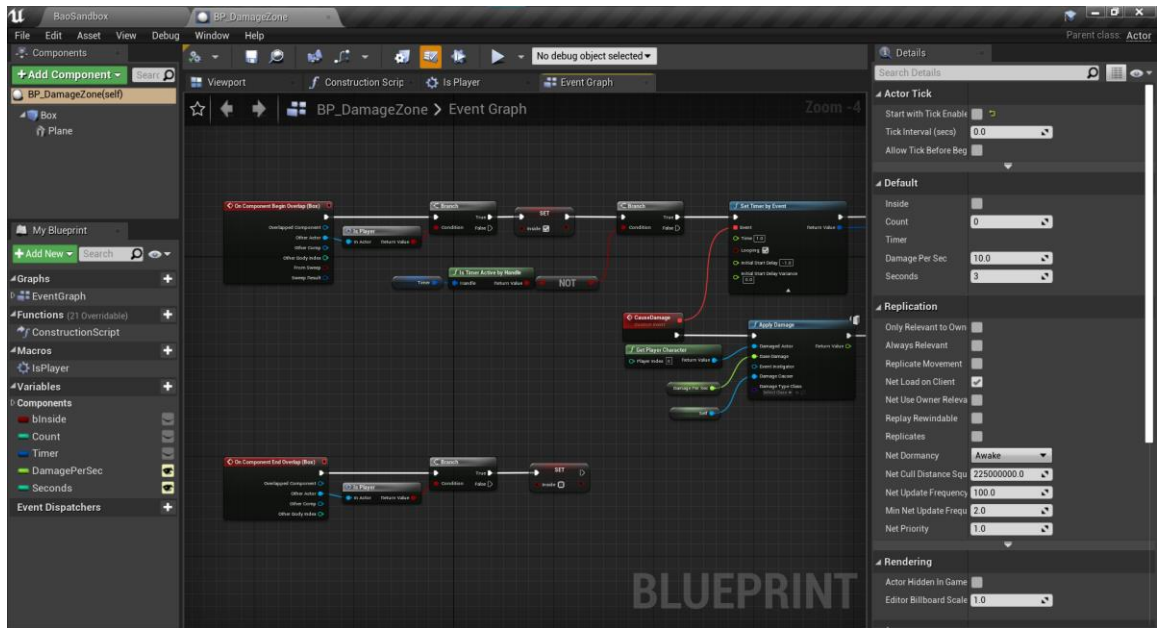


Figure 17. A screenshot of an Actor BP

There is no clear recommendation of which is better to use between C++ and BP, as they can have their own advantages, and they can be utilized together too. Moreover, each game and studio has its unique perspective and condition, so it is not straightforward to come up with the best selection. Fortunately, Epic Games suggests some general guidelines for the programmer to make better decisions and get the best of both worlds. [19]

One of the benefits when working with C++ is better performance in general when compared to BP. In addition, it provides more engine functionality and has more access to all other systems. C++ also offers more control over loading and saving data, as well as choosing which properties should be exposed to the editor. Furthermore, network bandwidth and timing are more efficient when being developed with it. Finally, C++ is much more compatible with version control, as its code and data are stored as text, which is easier to merge codes and work simultaneously between different programmers. [19]

On the other hand, BP offers much faster creation and iteration time. It can save a lot of time when creating a new BP, modifying and compiling it within a few seconds. That amount of time could potentially get more than a minute, performing the same action with a C++ class. Moreover, BP can visually represent the game flow and make it easier to implement the logic. Last but not least, artists and designers can even create and modify data in a BP, as it does not require as heavy in technical knowledge as C++. [19]

Consequently, this practical development of the game debugger will utilize both C++ and BP and combine them efficiently. Most of the functions or commands are implemented with C++, as it provides more engine and system functionality. Otherwise, the UI menu will be built with BP, as it can visually show the important elements.

### 3.3 Console Commands

In the programming paradigm, commands are technically coded functions that can be called in the console. The later parts will demonstrate how UE's built-in system can perform the function call as a console command.

Developers can bind which keys in the Project Settings (Figure 18) to bring up the command. They can also press the key twice to expand the console as shown in Figure 19.

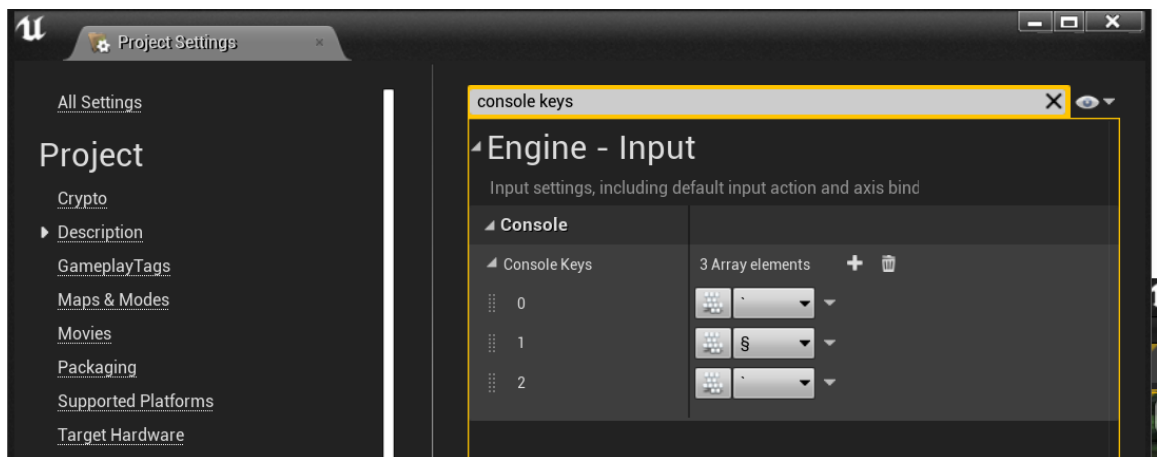


Figure 18. Setup keys to toggle UE's console in Project Settings

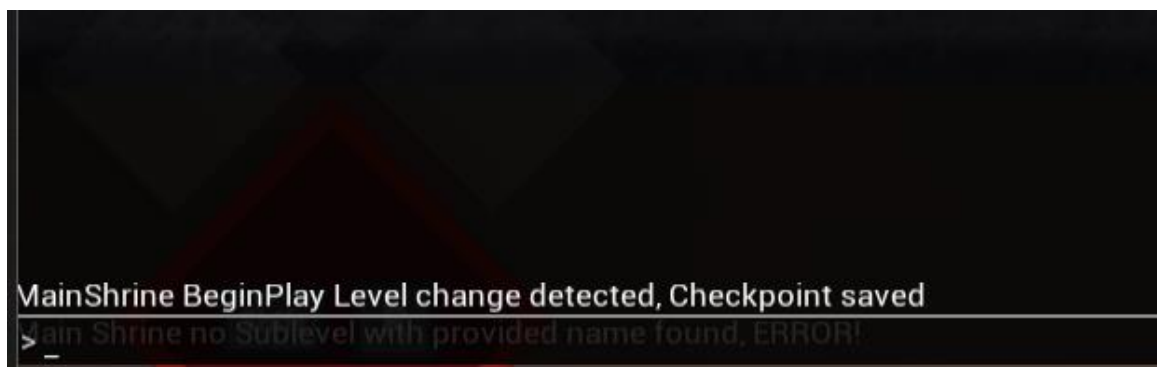


Figure 19. Bigger console in UE

### 3.4 Cheat Manager

As Unreal Engine 4 Documentation states, Cheat Manager is a fundamental blueprint for testing and debugging codes and behaviors, which are not included in the shipping game. The only purpose of Cheat Manager is just debugging, so there is no instance of it in the shipping version. [20]

With the above explanation, this is the best class to implement the game debugger, as it is not available in the final version, only in debugging one. It is possible to build this tool with other classes, such as GameMode. However, GameMode is an important class that comes with the shipping version. Therefore, CheatManager should handle the debugging system.

An instance of CheatManager is created by the PlayerController. Therefore, it is straightforward to change the CheatManager class in the Details panel of the PlayerController blueprint, as shown in Figure 20.

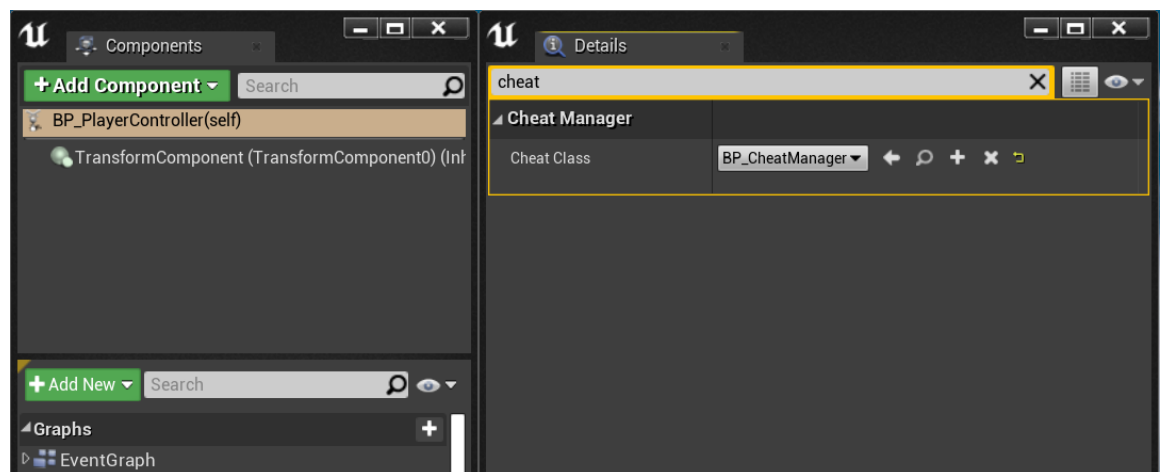


Figure 20. Changing Cheat class in PlayerController

By default, CheatManager class has a lot of powerful functions already. For example, DestroyAll can destroy all objects of a certain class, God is an invulnerability cheat, PlayersOnly can freeze everything in the level except for players, etc. [20]

### 3.5 UE's Reflection System

The capability for the program to inspect itself at runtime is called reflection. It is a principle technology of Unreal Engine, which is substantially helpful for driving many systems such as editor's

detail panels, serialization, garbage collection, replication over the network, and communication between Blueprint and C++. Nevertheless, reflection is not natively supported in C++. Thus, Epic Games developed their own system in Unreal Engine to gather, query, and control information about C++. [21]

The requirement to implement this reflection system is to add a special line (`#include "FileName.generated.h"`) on the top of the C++ script. This allows Unreal Header Tool (UHT) to recognize that it shall consider this file as reflected, then gather all necessary information when the project is compiled. [21] By default, after creating a C++ class or script within UE, that line should be included already and ready to use, except for some accidental modification.

Depending on the types and properties, several macros can be used before declaring them in the header file. In addition, these macros can consist of extra specifier keywords, which provide more features to the variables and functions. For example, a property can be exposed and modified in any detail panels, a function can be declared or called in a BP graph, etc. based on which keywords come along with the macro (Figure 21). Moreover, they can be divided into various categories, so developers can easily find them in BP. [21]

```
UPROPERTY(EditAnywhere, BlueprintReadOnly)
    struct FEnemyStats Stats; Changed in 11 blueprints
UPROPERTY(EditAnywhere, BlueprintReadOnly)
    TEnumAsByte<EEnemyTypeEnum> Enemytype = ET_UNKNOWN; Changed in 10 blueprints
UPROPERTY(EditAnywhere, BlueprintReadOnly)
    float ExpRewardModifier = 1.0f; "5" (BP_EnemyBaseFlesh_Kevin_3)
UPROPERTY(EditAnywhere, BlueprintReadOnly)
    bool CanBeTargeted = true; Changed in 2 blueprints
UFUNCTION(BlueprintPure)
    bool GetIsWounded() const;
UFUNCTION(BlueprintPure)
    float GetHealthPercentage() const;
```

Figure 21. A screenshot showing how the macros look like with the properties and functions

For the development of game debugger, this is a useful system by UE since it can help to expose the data to the menu UI BP. In addition, other special keywords with the UFUNCTION macro are mandatory for this too. The implementation part of this document will cover them in depth.

### 3.6 Unreal Motion Graphics

Unreal Motion Graphics (UMG) is an authoring tool that is utilized for creating UI elements such as menus, heads-up displays, or other graphical interfaces depending on what developers want. Widgets, which are the central part of UMG, are a collection of pre-built functions that can be utilized for constructing and designing interfaces, including buttons, sliders, bars, checkboxes, etc. They can be modified in Widget Blueprint (WBP), which consists of 2 tabs for creation: Designer tab and Graph tab. [22]

As already mentioned, the ultimate objective of this thesis is to integrate the game debugger into an interactive menu. Unreal Motion Graphics (UMG) is an essential tool in UE to accomplish that. It is technically a BP that is inherited from User Widget class, so it can acquire important information from the custom CheatManager, thanks to the reflection system. Then, those data will be categorized and shown on multiple buttons, offering users a vast selection of debugging options or cheats. Figure 22 illustrates the window of the debugger menu in UMG, without the functional buttons. They will be generated after the game starts.

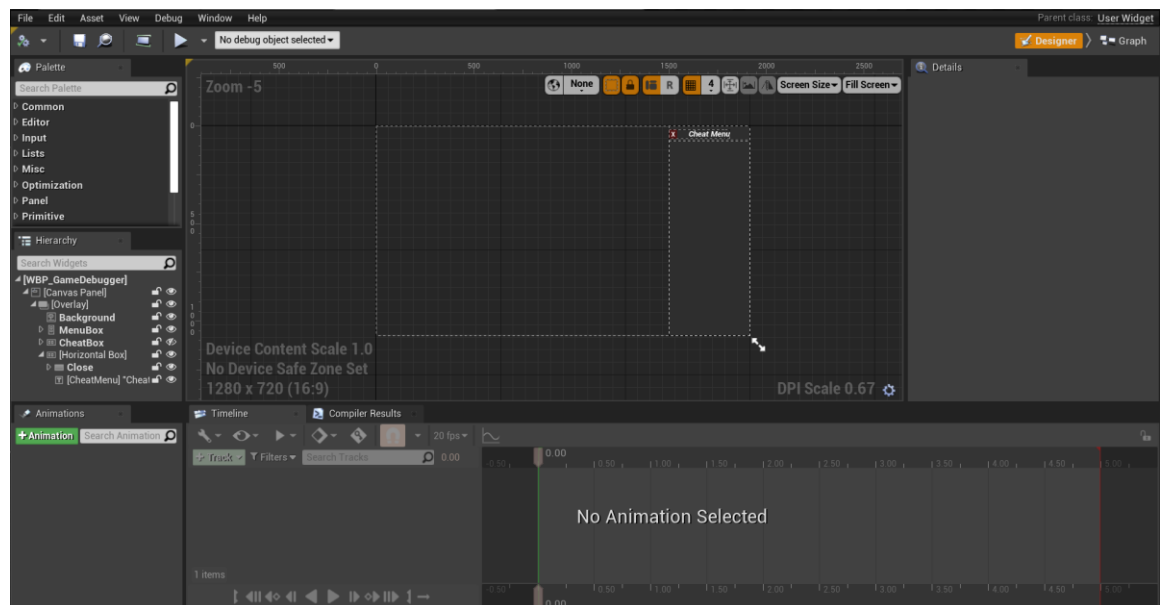


Figure 22. A screenshot of the UMG window of the game debugger

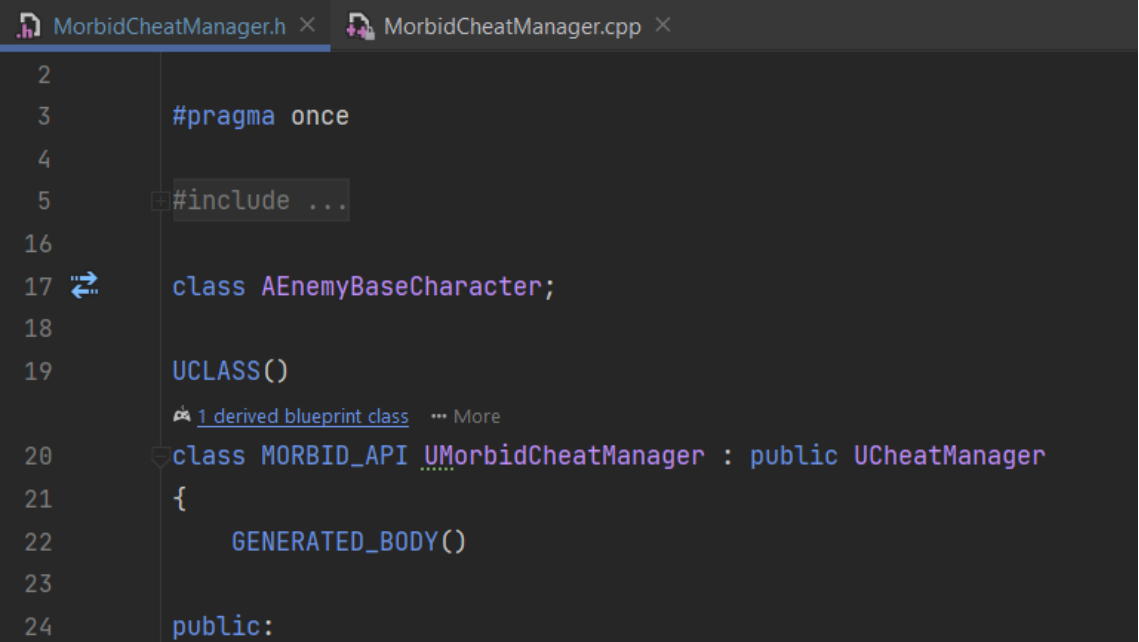


## 4 Implementation of Game Debugger

In the prototyping period, the tool is expected to be a built-in console with a diversity of commands. Users can type a certain command to perform their chosen action, with a value if applicable. Moreover, there is a special command that can print a list of usable command lines. Each line also contains a description of what that command can perform.

### 4.1 Integrating custom Cheat Manager

The first step is to set up a C++ script that is inherited with UCheaterManager class (Figure 23) to get access to most of the methods and properties of that class. For the most convenience, it is suggested to create a BP (Figure 24) that is inherited with that custom C++ script, to expose necessary properties to the editor and quickly prototype new functions.



```

2
3     #pragma once
4
5     #include ...
6
16
17     class AEnemyBaseCharacter;
18
19     UCLASS()
20     class MORBID_API UMorbidCheatManager : public UCheatManager
21     {
22     GENERATED_BODY()
23
24     public:

```

Figure 23. Custom C++ class that inherited with UCheatManger, header file

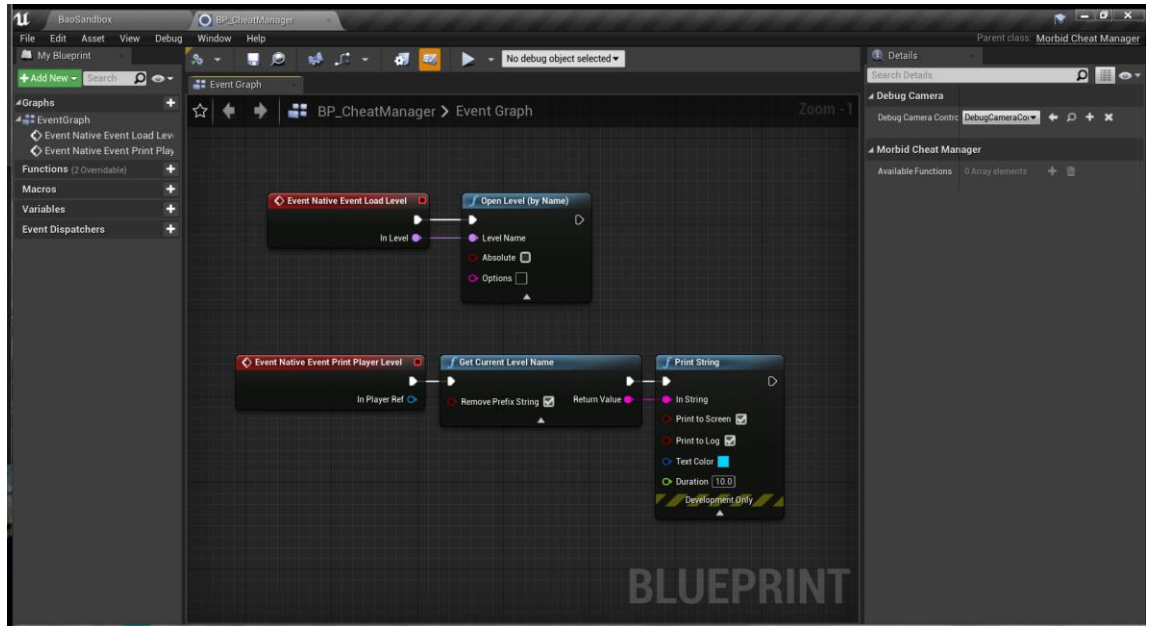


Figure 24. A BP with parent class of above script, with some prototyping functions

Last but not least, it is extremely important to remember to set which CheatManager class to use within the PlayerController, as mentioned in Figure 20 above.

#### 4.2 Forming custom commands

Based on the information in chapters 3.3 and 3.5, a command is technically a C++ function with UE's macro containing special keywords. In this case, the most important one is Exec (Figure 25), which allows a function to be executed in the in-game console.

```

UFUNCTION(Exec)
    void SetPlayerHealth(int32 NewHealth = 100);

```

Figure 25. Exec keyword in UFUNCTION macro

The function also takes one parameter with a default value of 100. With that setup, it is now possible to type the function to the console as a command. Figure 26 illustrates that UE also takes the new command into account and puts it in the console auto-complete system.



Figure 26. Typing the new command in the console

Furthermore, the console can display the NewHealth parameter and its type. If the user presses enter without any given value, UE will automatically accept the default value, which is 100 in this case, then execute the command. Otherwise, it can take a different value if the user types it as shown in Figure 27.

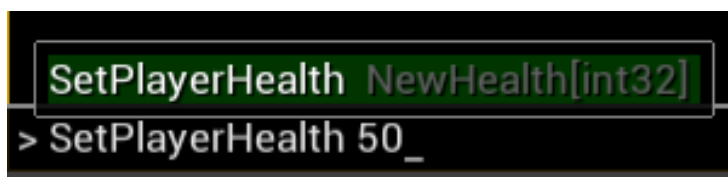


Figure 27. Optional value for the same command

It could be said that this would be a completion of the game debugger since developers can add as many commands as they want. However, if there is an enormous amount of them, it could be significantly challenging to maintain and memorize all of the commands. As a result, they should be split into various categories with a dot, to keep the commands clean and synchronized.

Fortunately, UE's macro combining with metadata specifiers allow programmers to define how properties and functions can interact with the editor and engine. [23] The essential keyword for categorizing purpose is `OverrideNativeName`, which can change the command into whatever the developer wants, with the ability to include any special characters and symbols.

With the same example, Figure 28 indicates how to use the `OverrideNativeName` specifier. `SetPlayerHealth` is now transformed into `Player.SetHealth` as a command. This offers the developer to classify the diversity of debugging types as they demand, such as `Player`, `Enemy`, `Item`, etc. Figure 29 below shows how they look in the in-game console.

```
UFUNCTION(Exec, BlueprintCallable, meta = (OverrideNativeName = "Player.SetHealth"))
void SetPlayerHealth(int32 NewHealth = 100);
```

Figure 28. Metadata specifier with the keyword of `OverrideNativeName`

```

Player.UnlimitedStamina
Player.UnlimitedAmmo
Player.Size                Multiplier[float]
Player.SetSpeed            NewSpeed[int32]
Player.SetHealth          NewHealth[int32]
Player.RecoverFullPostureStamina
Player.LaunchUp           LaunchUpForce[int32]
Player.LaunchForward      LaunchForwardForce[int32]
Player.DamageImmunity
Player.CinematicMode
Player.AlwaysRiposte
> Player.AlwaysRiposte

```

Figure 29. List of "cheats" within Player category

#### 4.3 Printing list of commands

Commands are now categorized into various sections. However, the next problem of the game debugger is to let the user know which commands are available for usage, and how they work in gameplay.

One way to solve this is to manually write a document to list all of the commands and describe each of them. Nevertheless, this could lead to another issue, which is uncomfortably confusing to keep both the code and the document to be in sync with each other, as the developer can easily lose track of a huge list if a command is added, removed, or modified.

Thanks to the UE's powerful reflection system, programmers can iterate through all functions with the UFUNCTION macro and utilize them with their data. [21] Therefore, it is possible to store the functions in an array (Figure 30) and get their essential information. These steps are performed within the GetAllFunctions function.

```

void UMorbidCheatManager::GetAllFunctions()
{
    for (TFieldIterator<UFunction> FIT (InStruct:StaticClass(), EFieldIteratorFlags::ExcludeSuper); FIT; ++FIT)
    {
        // Store functions for later use
        UFunction* Function = *FIT;

        if (Function->GetName() == TEXT("GetFunctionInfo")) continue;

        // Remove BlueprintNativeEvents using a naming scheme filter eg.
        // BlueprintNative event defined NativeEventLoadLevel contains NativeEvent
        // Also hide any function with name Hidden in it
        if (Function->GetName().Contains(SubStr:TEXT("NativeEvent")) || Function->GetName().Contains(SubStr:TEXT("Hidden")))
        {
            continue ;
        }

        AvailableFunctions.Add(Function);
    }
}

```

Figure 30. Use TFieldIterator and store them in AvailableFunctions array, with some cases are not taken into account

#### 4.3.1 Command text and values

Figure 31 describes the way to print the cheat on the console. It gets the name of the stored function, then again loop through its parameters and add them between the square brackets (Figure 32). The codes below the comment “Check longest name” are not vital here since it is just for decoration. Finally, the final text is added into an FString array called FunctionLines.

```

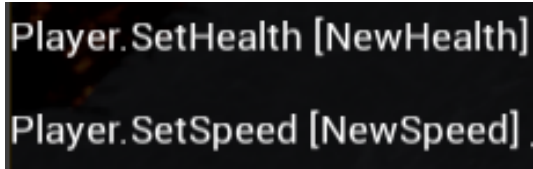
// Get function parameters and add to name
FString Name = Function->GetName() + TEXT(" [");
for ( TFieldIterator<FProperty> It(Function); It && (It->PropertyFlags&(CPF_Parm)) == CPF_Parm; ++It )
{
    Name += It->GetName();
    if (It->Next != nullptr)
    {
        Name += TEXT(", ");
    }
}
Name += TEXT("] ");

// Check longest name
if (Name.Len() > LongestName)
{
    LongestName = Name.Len();
    //UE_LOG(LogTemp, Warning, TEXT("Longest is now %i"), LongestName);
}

FunctionLines.Add(Name);

```

Figure 31. Get function name and its parameters



Player.SetHealth [NewHealth]  
Player.SetSpeed [NewSpeed]

Figure 32. How they are printed in the console

It is worth mentioning again at this stage, there are 2 different arrays but of the same size, and they also work parallel with each other. The first one is AvailableFuntions which contains the Ufunction. The second one is FunctionLines which stores the whole line of commands and their descriptions.

#### 4.3.2 Command descriptions

With the parallel arrays, it is safe to loop one with an index and get the information from the other one with the same index. In this particular case, a for-loop is used for the FunctionLines with index I, as demonstrated in Figure 33.

```
// Add spaces and descriptions
for (int32 I = 0; I < FunctionLines.Num(); I++)
{
    int32 ActualSpaces = LongestName - FunctionLines[I].Len() + LineSpaces;
    for (int32 J = 0; J <= ActualSpaces; J++)
    {
        FunctionLines[I] += TEXT("_");
    }
    FunctionLines[I] += TEXT(" "); // Add a space in the end

    FString Description = AvailableFunctions[I]->GetToolTipText(bShortTooltip: true).ToString();
    FunctionLines[I] += Description;
}
}
```

Figure 33. Add spaces and descriptions

Within each line, a description is collected through an element of AvailableFunctions with the same index I, by the support of the GetToolTipText method. This method can return the comment texts that programmers put above each function (Figure 34). In addition, underlines are added to make more space between a command and its description (Figure 35).

```

// Morbid Commands
/** Help command to print list of cheats */
UFUNCTION(Exec, BlueprintCallable, meta = (OverrideNativeName = "M2.HELP"))
    void MorbidHelp();
/** Activate Morbid cheat, this should be done FIRST before anything */
UFUNCTION(Exec, BlueprintCallable, meta = (OverrideNativeName = "M2.ACTIVATE"))
    void ActivateCheat();

// Player Commands
/** Change into Cinematic Mode, which basically just removes enemies from the level */
UFUNCTION(Exec, BlueprintCallable, meta = (OverrideNativeName = "Player.CinematicMode"))
    void CinematicMode();
/** Change Player's size to default size * [Multiplier]. Default is {2} (double size) */
UFUNCTION(Exec, BlueprintCallable, meta = (OverrideNativeName = "Player.Size"))
    void PlayerChangeSize(float Multiplier = 2) { ChangeSize(Multiplier); }
/** Launch Player upward with [LaunchUpForce]. Default is {1000} */
UFUNCTION(Exec, BlueprintCallable, meta = (OverrideNativeName = "Player.LaunchUp"))
    void LaunchPlayerUp(int32 LaunchUpForce = 1000);
/** Launch Player forward with [LaunchForwardForce]. Default is {1000} */
UFUNCTION(Exec, BlueprintCallable, meta = (OverrideNativeName = "Player.LaunchForward"))
    void LaunchPlayerForward(int32 LaunchForwardForce = 1000);
/** Set current Player's HP to [NewHealth] amount. Default is {100} */
UFUNCTION(Exec, BlueprintCallable, meta = (OverrideNativeName = "Player.SetHealth"))
    void SetPlayerHealth(int32 NewHealth = 100);
/** Set current Player's speed to [NewSpeed] amount. Default is {1000} */
UFUNCTION(Exec, BlueprintCallable, meta = (OverrideNativeName = "Player.SetSpeed"))
    void SetPlayerSpeed(int32 NewSpeed = 1000);

```

Figure 34. The comment line above each function acts like a tooltip, used for generating the command's description

M2.ACTIVATE []	Activate Morbid cheat, this should be done FIRST before anything
M2.Freeze [Delay]	Pause the game for [Delay] seconds. Default is {1}
M2.HELP []	Help command to print list of cheats
M2.TimeSpeed [NewTimeSpeed]	Set [NewTimeSpeed] as desired. Value > 1 is faster, value < 1 is slower. Default is {1} (unchanged)
Player.AlwaysRiposte []	Set always Riposte when Player is blocking
Player.CinematicMode []	Change into Cinematic Mode, which basically just removes enemies from the level
Player.DamageImmunity []	Set damage immunity for Player
Player.LaunchForward [LaunchForwardForce]	Launch Player forward with [LaunchForwardForce]. Default is {1000}
Player.LaunchUp [LaunchUpForce]	Launch Player upward with [LaunchUpForce]. Default is {1000}
Player.RecoverFullPostureStamina []	Recover full posture and stamina for Player
Player.SetHealth [NewHealth]	Set current Player's HP to [NewHealth] amount. Default is {100}
Player.SetSpeed [NewSpeed]	Set current Player's speed to [NewSpeed] amount. Default is {1000}

Figure 35. Results of printing the list of commands

There was an attempt to decorate and make the descriptions vertically in-line. However, the sizes of letters in UE's console are not the same and uniform, so the aesthetical aspect is not too respectable.

#### 4.3.3 Final printing

All the codes above, which formatted and stored every command in the FunctionLines array, are scripted within the GetAllFunctions method. Nevertheless, it has not been called anywhere yet. The best place to execute this function is in InitCheatManager (Figure 36), which is a virtual function of UCheatManager class that is called when the cheat manager is created.

```
void UMorbidCheatManager::InitCheatManager()
{
    Super::InitCheatManager();

    GetAllFunctions();
}
```

Figure 36. Calling function

All the text in Figure 35 is printed thanks to a support function called PrintMessage (Figure 37), which utilizes an inherited method to print texts to the in-game console.

```
void UMorbidCheatManager::PrintMessage(FString Message)
{
    GetOuterAPlayerController()->ClientMessage(Message);
    GetOuterAPlayerController()->ClientMessage(S::TEXT(" ")); // New line
}
```

Figure 37. PrintMessage function

With all things set up, the ultimate command to print all of them is called M2.HELP (one of the functions in Figure 34). It will loop the cached FunctionLines array (Figure 38) and print all of them as formatted (Figure 39).



```

void UMorbidCheatManager::MorbidityHelp()
{
    // Now print everything
    PrintMessage("");
    PrintMessage(TEXT(" ----- Game Debugger / Cheat Sheet ----- "));
    PrintMessage(TEXT("Cheat Typing Instruction (no square brackets): [CheatCommand] [Value]"));
    for (FString Line: FunctionLines)
    {
        PrintMessage(Line);
        UE_LOG(LogTemp, Warning, TEXT("%s"), *Line);
    }
}

```

Figure 38. Implementation of MorbidHelp function for M2.HELP command

```

>>> M2.HELP <<<
----- Game Debugger / Cheat Sheet -----
Cheat Typing Instruction (no square brackets): [CheatCommand] [Value]
Debug.AttackTimingBar [] _____ Debug toggle Attack Timing Bar visual element on UI
Debug.Camera [] _____ Debug toggle Camera debug messages printing
Debug.Controller [] _____ Debug toggle visibility of information about the Controller devices on UI
Debug.DodgeTimingBar [] _____ Debug toggle Dodge Timing Bar visual element on UI
Debug.Hitboxes [] _____ Debug toggle visibility of Attack Hitboxes in game
Debug.LockingAlgorithm [] _____ Debug toggle LockingAlgorithm visibility
Debug.Notifications [] _____ Debug toggle Notifications UI debug panel visibility to count Prompts
Debug.RageMode [] _____ Debug toggle Rage Mode debugging info
Debug.RangedMode [] _____ Debug toggle Ranged attack algorithm visibility
> _

```

Figure 39. After entering M2.HELP

#### 4.4 Transferring to UI menu

With all the information already gathered from the last subchapters, this step will be about building the UI system and utilizing them as data. Besides the main WBP\_GameDebugger, it also requires 2 more components to make it complete (Figure 40). One is WBP\_CheatMenuButton, which acts as a category and will bring up another menu when it is clicked. The other one is WBP\_FunctionButton that is the real command and will execute its mechanic after clicked.

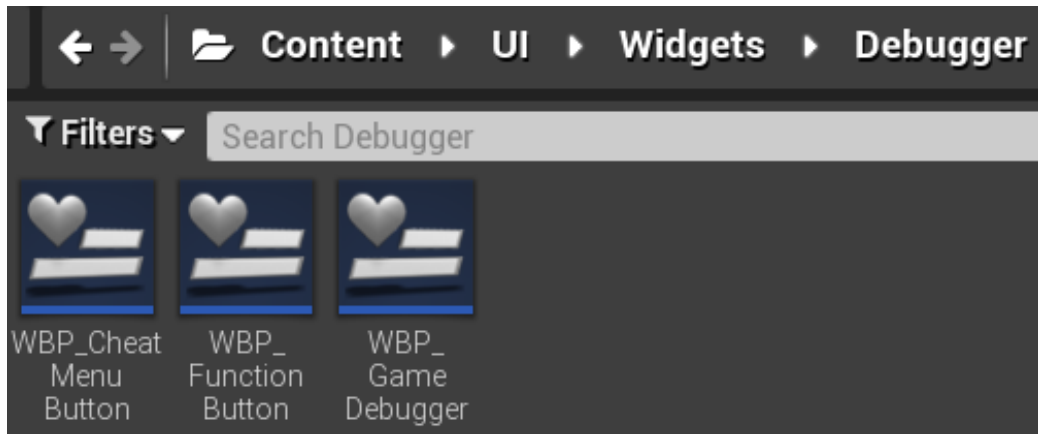


Figure 40. Widgets needed to build a debugger menu

#### 4.4.1 WBP\_CheatMenuButton

Figure 41 visually shows how the WBP\_CheatMenuButton looks like. It simply contains a button with some text in it that can be changed with BP codes. The button's On Clicked event will let the main debugger menu know and generate a list of WBP\_FunctionButton as commands, as shown in Figure 42.

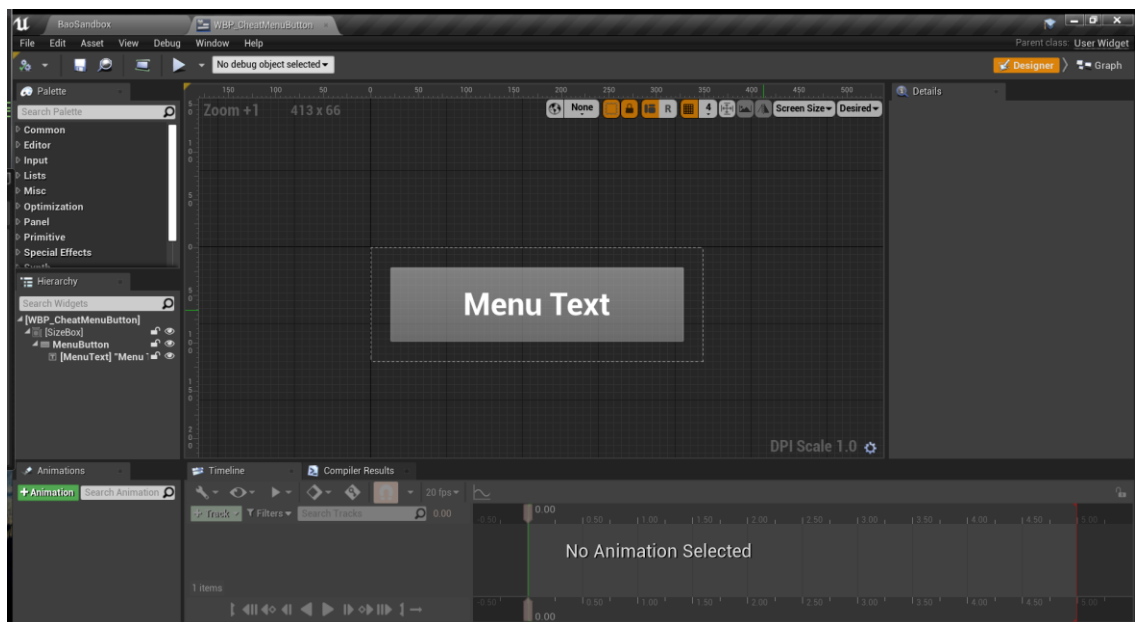


Figure 41. WBP\_CheatMenuButton widget editor

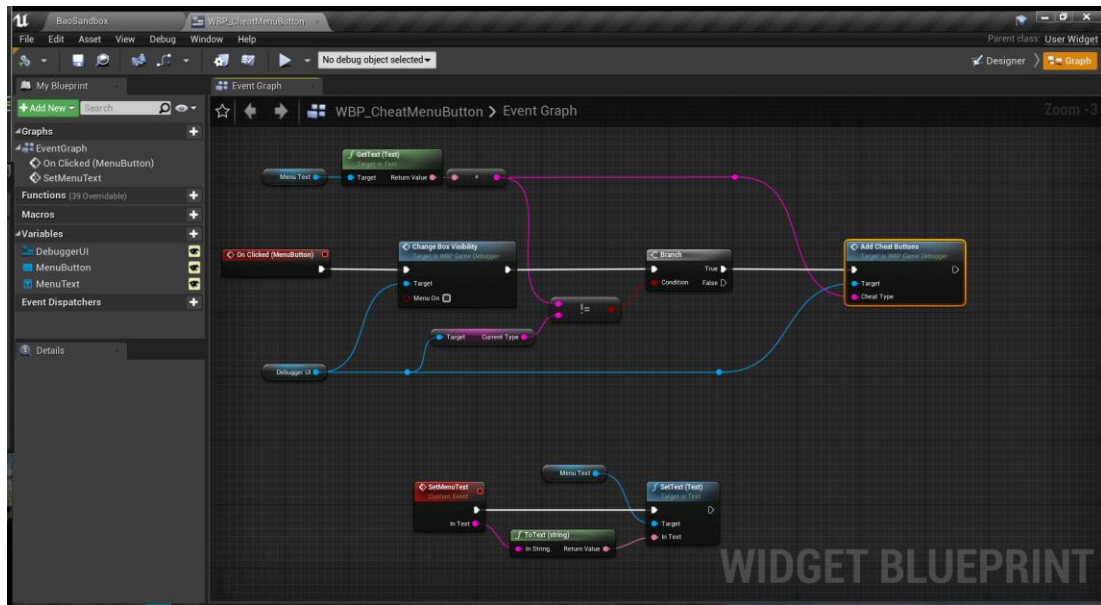


Figure 42. Event Graph of WBP\_CheatMenuButton

#### 4.4.2 WBP\_FunctionButton

Figure 43 illustrates the necessary UI elements for WBP\_FunctionButton. It includes a button with text, a text of parameter name, and a text box that can receive input value from the user. The On Click of that button will take the cheat text and its parameter value, combine them into one string and run it with a method called Execute Console Command (Figure 44).

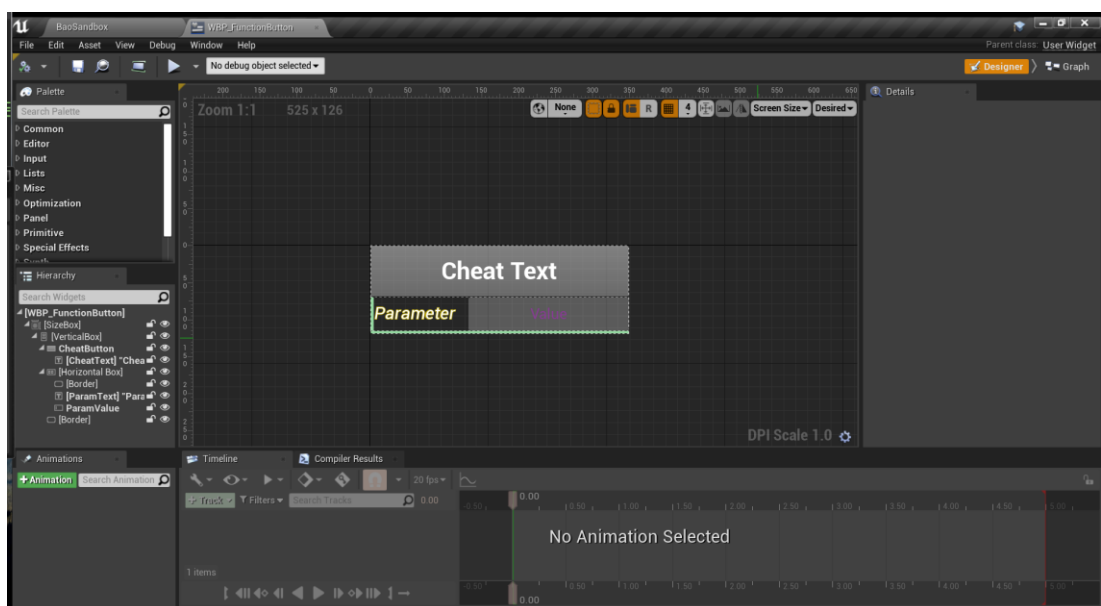


Figure 43. WBP\_FunctionButton widget editor

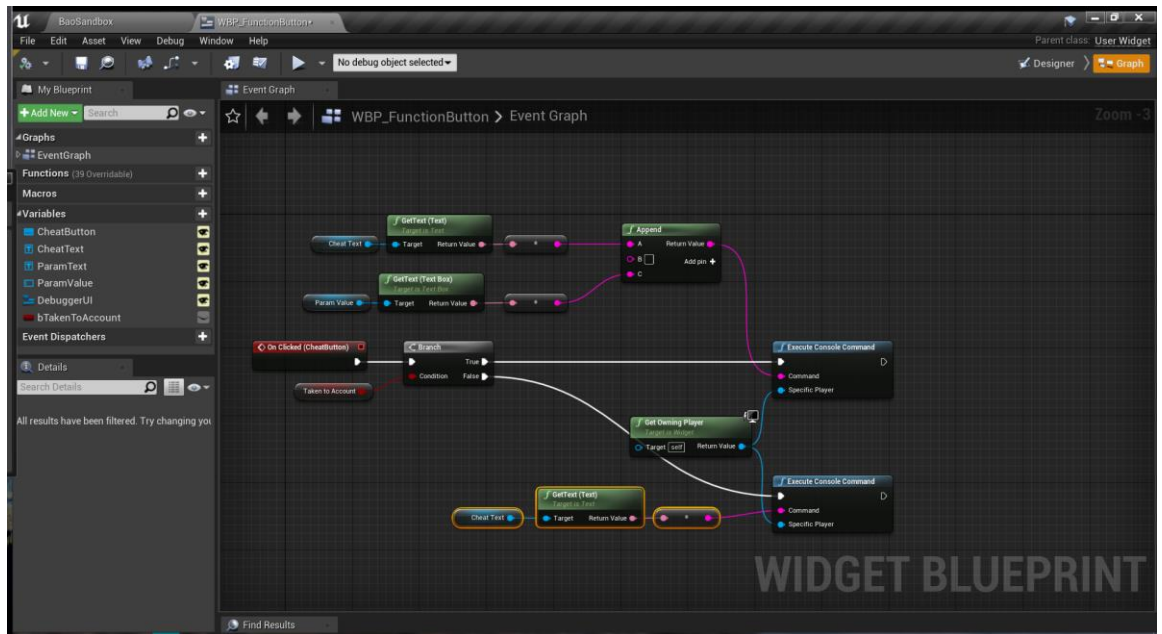


Figure 44. On Clicked event to execute a command

#### 4.4.3 WBP\_GameDebugger

This widget is the core menu of the game debugger (already shown in Figure 22). It consists of all the functions to automatically create buttons, which can lead to another categorized menu, or execute a console command.

Figure 45 below indicates a function that is called when the widget is constructed. It accesses the Cheat Manager reference and gets all the AvailableFunctions array. Then it loops that array and gets the function name of each element and stores it as unique in a new array. With that loop finished, it can be certain that the MenuTexts array will only include categorized texts. Subsequently, for each element during that array's loop, create a new widget of type WBP\_Cheat-MenuButton and assign the text to it.

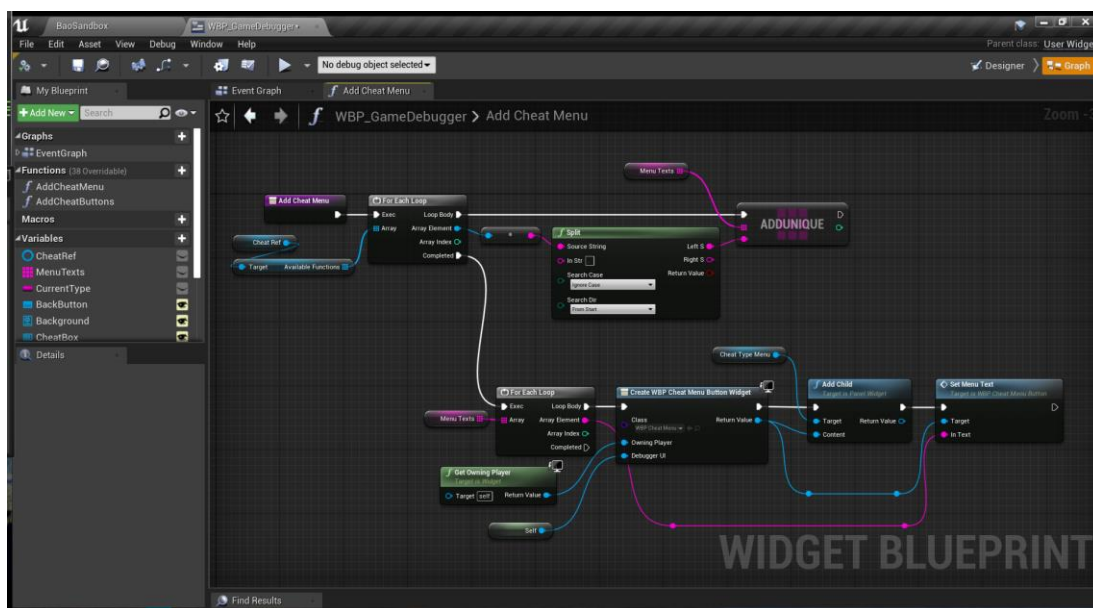


Figure 45. AddCheatMenu function

As stated in 4.4.1, the AddCheatButtons function will create a list of WBP\_FunctionButton. Technically, it gets the AvailableFunctions array in the Cheat Manager reference and loops it. For each element, it will go through another function in UMorbidCheatManager named GetFunctionInfo (Figure 46), which returns the command's parameters, values, and description.

```

void UMorbidCheatManager::GetFunctionInfo(UFunction* FunctionToCheck, TArray<FString>& Params,
TArray<FString>& Values, FString& Description) const
{
    // Get parameters
    for (TFieldIterator<FProperty> It(FunctionToCheck); It && (It->PropertyFlags&(CPF_Parm)) == CPF_Parm; ++It)
    {
        Params.Add(Item:It->GetName());
    }

    // Get Description and Param Value from the function line
    int32 FunctionIndex;
    if (AvailableFunctions.Find(FunctionToCheck, [&]FunctionIndex))
    {
        // Description
        FString Left;
        FunctionLines[FunctionIndex].Split(InS:TEXT("_"), &Left, RightS:&Description, ESearchCase::IgnoreCase, ESearchDir::FromEnd);

        // Param Value
        int32 StartIndex = FunctionLines[FunctionIndex].Find(SubStr:TEXT("{"), ESearchCase::IgnoreCase, ESearchDir::FromEnd);
        int32 LastIndex = FunctionLines[FunctionIndex].Find(SubStr:TEXT("}"), ESearchCase::IgnoreCase, ESearchDir::FromEnd);
        int32 Length = LastIndex - StartIndex;
        if (Length > 1)
        {
            FString Value = FunctionLines[FunctionIndex].Mid(Start:StartIndex + 1, Count:Length - 1);
            Values.Add(Value);
        }
    }
}

```

Figure 46. GetFunctionInfo method, which returns essential information of the input UFunction

The acquired information is now passed into `WBP_FunctionButton` and set everything accordingly, including the tooltip which describes the command's description, as demonstrated in Figure 47.

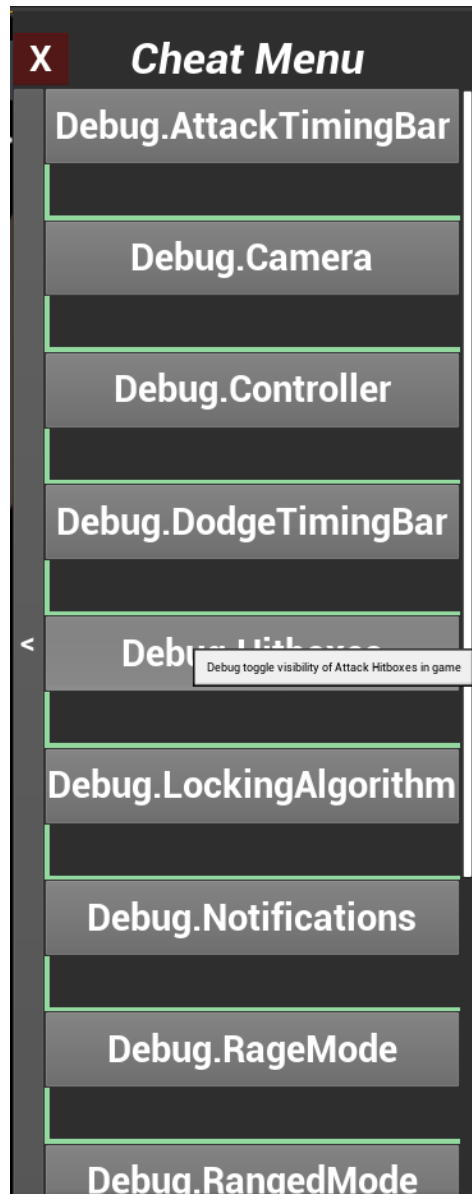


Figure 47. Cheat commands in Debug group, with a tooltip that shows a description when hovering the mouse

The figure above also shows that the parameter text and input value box are hidden since it does not take any arguments. Figure 48 below illustrates another group with commands that have a parameter.

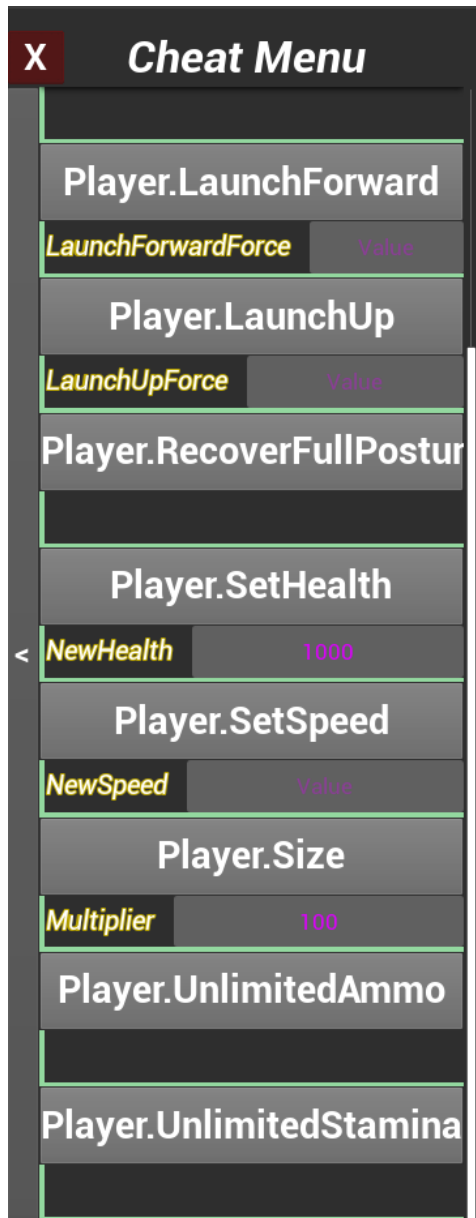


Figure 48. Player group of the debugging tool

#### 4.5 Setup input

It must not be forgotten that the final steps are to create the game debugger widget and bind the input to toggle it on and off. These should be done in PlayerController BP. CreateDebuggerUI (Figure 49) is executed in the BeginPlay event, while number 4 will be the input to toggle the menu (Figure 50).



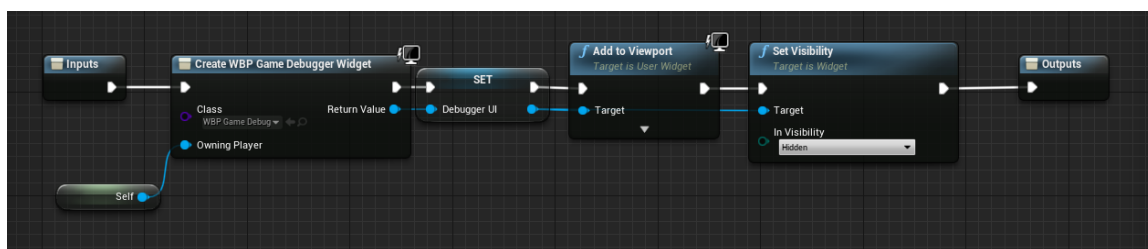


Figure 49. Creating WBP\_GameDebugger in BP\_PlayerController

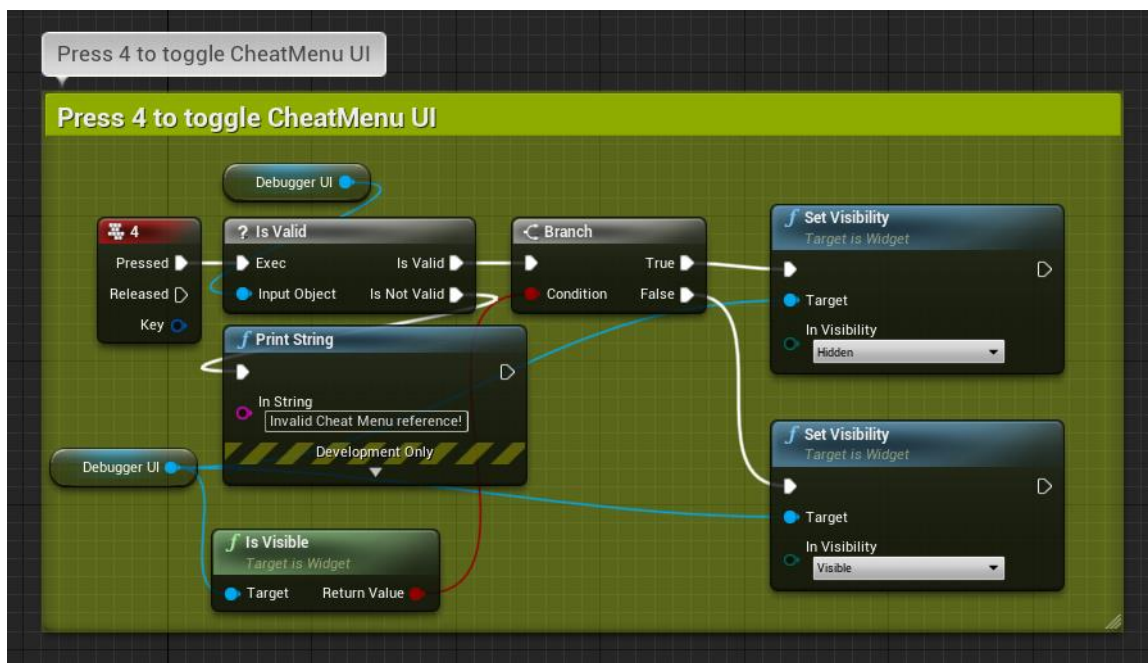


Figure 50. Bind number 4 to toggle WBP\_GameDebugger on/off

#### 4.6 Result

Finally, the desired product is an interactive and user-friendly menu (Figure 51) with multiple buttons, which provide several functional “cheats” for the users. They are divided into various categories (Figure 47 and Figure 48) to offer organized elements that are easier to use. In addition, a tooltip is also provided when users hover the mouse over a button (Figure 47), which can indicate the description of what should happen after that button is pressed.



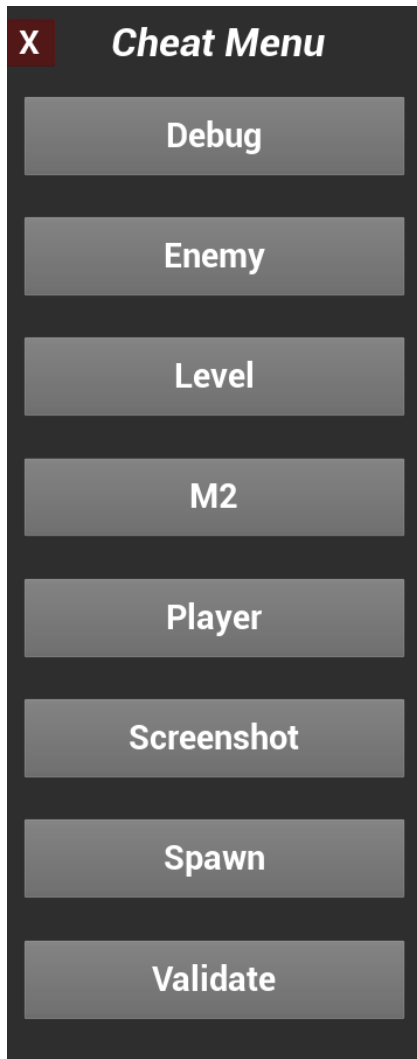


Figure 51. The main interface of the debugger menu, with various categories

Furthermore, the UE console that still exists, which is considered a prototype, can also be used in parallel with this debug menu. It depends on the user's choice, but the more advanced debug menu offers more quick usability and friendly interaction.

## 5 Conclusion

In summary, the game debugger plays an important role in the game development industry. It can offer a faster and more efficient production process of a game, especially for programmers, before publishing it on markets.

The knowledge and skills gained from this research and practical development are significantly valuable for the author. They covered a lot of important information about the background and definition of the debugging tool, as well as its purpose in making games. In addition, the author had a great chance to study and build a custom game debugger in Unreal Engine 4. Especially for an intern and a freshly graduate student, the real-world involvement has been delightful and pleasing, as the author has been acquiring more experience in developing a complex and modular system for a game company to utilize in their title.

For the duration, the prototyping stage of this game debugger was about 1 working week, and 2 additional weeks to achieve the desired result. After that, it has been getting developed to extend more features and functionalities for about 5 months, and is continued for as long as the game grows.

With the prior setups, the final product has been accomplished better than expected. Both the debugging console and menu are existing at the same time and can be chosen based on the user's favorite. The system is also extremely flexible and extensible for creating new commands. The program only needs to add a function and provide necessary macro keywords, then write some comments for the tooltip, which is not even compulsory for this progress. Afterward, everything will automatically work well on both console and menu, without any extra modification needed.

## References

- 1 Indeed. 7 Key Roles in Video Game Development [Internet]. 2021 [cited 2021 September 26]. Available from: <https://www.indeed.com/career-advice/finding-a-job/game-development-roles>.
- 2 Knickerbocker P. Game Maker Debugging [Internet]. 2009 [cited 2021 October 2]. Available from: <https://media.lanec.edu/cit/gamedev/CIS125G/Extra%20Tutorials/Debug.pdf>.
- 3 GameCloud. Look At Various Types of Game Bugs Found in Game Testing Methodologies [Internet]. [cited 2021 October 2]. Available from: <https://gamecloud-ltd.com/look-at-various-types-of-game-bugs-found-in-game-testing/>.
- 4 WIRED. Fallout 4 Is Full of Bugs, But Fixing Them Could Ruin It [Internet]. 2015 [cited 2021 October 2]. Available from: <https://www.wired.com/2015/11/fallout-4-bugs/>.
- 5 Fandom. Debug menu [Internet]. 2007 [updated 2015] [cited 2021 October 3]. Available from: [https://nintendo.fandom.com/wiki/Debug\\_menu](https://nintendo.fandom.com/wiki/Debug_menu).
- 6 The Cutting Room Floor. Patapon 3/Debug Menu [Internet]. [cited 2021 October 3]. Available from: [https://tcrf.net/Patapon\\_3/Debug\\_Menu](https://tcrf.net/Patapon_3/Debug_Menu).
- 7 Fandom. Proto:Pokémon Emerald/Debug Menu [Internet]. [updated 2021] [cited 2021 October 28]. Available from: [https://tcrf.net/Proto:Pok%C3%A9mon\\_Emerald/Debug\\_Menu](https://tcrf.net/Proto:Pok%C3%A9mon_Emerald/Debug_Menu).
- 8 Fandom. Proto:The Legend of Zelda: Majora's Mask/Debug Version [Internet]. [updated 2021] [cited 2021 October 28]. Available from: [https://tcrf.net/Proto:The\\_Legend\\_of\\_Zelda:\\_Majora%27s\\_Mask/Debug\\_Version](https://tcrf.net/Proto:The_Legend_of_Zelda:_Majora%27s_Mask/Debug_Version).
- 9 Denham T. What is Unreal Engine? [Internet]. [cited 2021 October 3]. Available from: <https://conceptartempire.com/what-is-unreal-engine/>.

- 10 Epic Games. Fortnite [Internet]. [updated 2021] [cited 2021 October 3]. Available from: <https://www.epicgames.com/fortnite/en-US/home>.
- 11 Epic Games. Unreal Engine 4 Documentation [Internet]. 2004 [updated 2021] [cited 2021 October 3]. Available from: <https://docs.unrealengine.com/4.27/en-US/>.
- 12 JetBrains. Rider for Unreal Engine [Internet]. 2021 [cited 2021 October 3]. Available from: <https://www.jetbrains.com/idea/rider-unreal/>.
- 13 Veracode. What Is An Integrated Development Environment (IDE)? [Internet]. [cited 2021 October 13]. Available from: <https://www.veracode.com/security/integrated-development-environment>.
- 14 Unity Technologies. Debugging C# code in Unity [Internet]. 2021 [cited 2021 October 28]. Available from: <https://docs.unity3d.com/Manual/ManagedCodeDebugging.html>.
- 15 Epic Games. Blueprint Class [Internet]. [cited 2021 October 14]. Available from: <https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/UserGuide/Types/ClassBlueprint/>.
- 16 Epic Games. Setting Up a Game Mode [Internet]. [cited 2021 October 14]. Available from: <https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/HowTo/SettingUpAGameMode/>.
- 17 Epic Games. Blueprint Overview [Internet]. [cited 2021 October 14]. Available from: <https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/Overview/>.
- 18 Epic Games. EventGraph [Internet]. [cited 2021 October 27]. Available from: <https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/Blueprints/UserGuide/EventGraph/>.
- 19 Epic Games. Balancing Blueprint and C++ [Internet]. [cited 2021 October 14]. Available from: <https://docs.unrealengine.com/4.27/en-US/Resources/SampleGames/ARPG/BalancingBlueprintAndCPP/>.

- 20 Epic Games. UHeatManager [Internet]. [cited 2021 October 13]. Available from: <https://docs.unrealengine.com/4.26/en-US/API/Runtime/Engine/GameFramework/UHeatManager/>.
- 21 Noland M. Unreal Property System (Reflection) [Internet]. 2014 [cited 2021 October 13]. Available from: <https://www.unrealengine.com/en-US/blog/unreal-property-system-reflection>.
- 22 Epic Games. UMG UI Designer [Internet]. [cited 2021 October 14]. Available from: <https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/UMG/>.
- 23 Epic Games. Metadata Specifiers [Internet]. [cited 2021 October 24]. Available from: <https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/GameplayArchitecture/Metadata/>.