Bachelor’s thesis

Information Technology

2019

Aaro Salonen

USER EXPERIENCE TESTING OF VISUAL SCRIPTING

– Using Blueprints and PlayMaker

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR’S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information Technology

Completion year of the thesis 2019 | 39 pages

Aaro Salonen

USER EXPERIENCE TESTING OF VISUAL SCRIPTING

Using Blueprints and PlayMaker

The objective of this thesis work was to provide a statistical and practical evaluation to visual scripting tools with user experience test providing the viewpoint of a target user, and compare two different tools on two separate game development engines.

This thesis was commissioned by Turku Game Lab and is based on the request of the commissioner to look into the visual scripting tools found in Unreal Engine and on Unity.

The goal was to study Unreal Engine 4 and its internal visual scripting tool, Blueprints, in direct comparison to Unity’s plugin, PlayMaker. The author outlined definitions for tasks and attempted to complete similar projects on both engines utilizing only the visual scripting tools while contrasting user story method with the data.

The focus of the thesis remained on the comparison between the two visual scripting languages and on the data collected through task completion success rates. Both game engines and scripting languages are similar to one another but the languages had different approaches to problem solving, such as variable and node connection counts. Additionally, the thesis shows how inexperience with programming can be one of the greatest difficulties for a beginning developer faces and provides statistical analysis on these claims.

KEYWORDS:

Visual Scripting Language, Visual Programming, Unity, PlayMaker, Unreal Engine, Blueprints, User Experience, User Story

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Informaatio Teknologia

Opinnäytetyön valmistumisajankohta 2019 | 39 sivua

Aaro Salonen

VISUAALISEN SKRIPTAUKSEN KÄYTTÄJÄKOKEMUSTESTAUSTA

- Käyttäen Blueprinttejä ja PlayMakeria

Tämän opinnäytetyön tavoiteena oli antaa tilastollinen ja käytännöllinen arvio visuaalisille skriptityökaluille käyttäjäkokemustestauksella kohdekäyttäjän näkökulmasta ja vertailla kahta eri työkalua kahdella erillisellä pelikehitysmoottorilla.

Opinnäytetyön tilasti Turku Game Lab ja se perustuu tilaajan pyyntöön tutkia Unreal Enginestä ja Unitystä löytyviä visuaalisia skriptityökaluja.

Tavoitteena oli tutkia Unreal Engine 4 sisäistä visuaalista skriptityökalua nimeltä Blueprints ja verrata sitä suoraan Unityn ladattavaan lisäosaan nimeltä PlayMaker. Kirjoittaja määritteli mahdollisimman samanlaiset tehtävät ja yritti suoriutua niistä käyttäen ainoastaan molempien pelimoottoreiden skriptityökaluja, luoden käyttäjätarinan analysoitavaksi tuloksien rinnalle.

Opinnäytetyön painopiste keskittyi kahden visuaalisen skriptikielen vertailuun ja asetettujen tehtävien onnistuneesti suoriutumisista kerättyyn tietoon. Molemmat moottorit ja kielet olivat hyvin saman tyyliset, mutta muuttujien ja graafisten noodien määrät vaihtelivat samojen ongelmien ratkaisuissa. Tutkimus osoittaa, mitkä voivat olla suurimpia vaikeuksia aloittavalla pelikehittäjälle ja tarjoaa tilastollisen analyysin osoittamaan niitä.

ASIASANAT:

Visuaalinen skriptauskieli, visuaalinen ohjelmointi, Unity, PlayMaker, Unreal Engine, Blueprints, Käyttäjätarina, Käyttäjäkokemus

CONTENTS

LIST OF ABBREVIATIONS 6

1 INTRODUCTION 6

2 KEY CONCEPTS 8

2.1 Game engines 8

2.2 Visual scripting 9

2.3 Programming and scripting 10

2.4 Topic selection 12

3 PRACTICAL APPROACH 13

3.1 Evaluation and metrics 13

3.1.1 Game definitions 13

3.1.2 Game object definitions 13

3.1.3 Project sequence 15

3.2 Evaluation 16

4 PARALLEL COMPARISONS 19

5 PLAYMAKER PROCESS 22

5.1 Confined space 23

5.2 Controllable object 23

5.3 Avoidable object 24

5.4 Optional object 24

5.5 Required object 25

5.6 Desired object 25

5.7 Extra constraint 25

5.8 Additional notes 26

6 BLUEPRINTS PROCESS 27

6.1 Confined space 29

6.2 Controllable object 29

6.3 Avoidable object 30

6.4 Optional object 30

6.5 Required object 31

6.6 Desired object 31
6.7 Extra constraint 31

7 DATA REFLECTION 32

8 PROJECT REFLECTION 35

9 CONCLUSION 37

REFERENCES 38

FIGURES

Figure 1. Microsoft visual programming language flow graph. (Microsoft, 2012) 8
Figure 2. Node based VSL in comparison to nativized code in UE. (Unreal Engine, 2019) 10
Figure 3. The workflow of the project tasks. 17
Figure 4. Object creation. 20
Figure 5. Detail editor. 20
Figure 6. Lists of game objects and their visual scripting representations. 21
Figure 7. Project view on Unity. 22
Figure 8. Project view on UE. 27
Figure 9. Blueprints UI. 28
Figure 10. Success rates of each FSM for Unity project and its components. 32
Figure 11. Success rates of each Blueprint for Unreal Engine project and its components. 32
Figure 12. PlayMaker variable count by game object. 34
Figure 13. Blueprints variable count by game object. 34

LIST OF ABBREVIATIONS

2D	Two-dimensional
3D	Three-dimensional
FSM	Finite-state machine
UE	Unreal Engine, game engine
UI	User interface
Unity	Game engine
UX	User experience
VPL	Visual programming language
VPT	Visual programming tool
VSL	Visual scripting language
VST	Visual scripting tool

1 INTRODUCTION

This thesis provides a point of view to the visual scripting tools from a non-coder by creating a set of game elements with two methods on separate game development engines. The goal was to understand why game engines are so popular and show what was successfully created without external help and what was not. The tasks in the thesis were not part of any official project and will not become published.

This topic was chosen because game development has been the field of interest for the author for years, and because the game industry has managed to stay relevant in the entertainment industry long enough to start providing development applications to the consumers. This thesis was commissioned by Turku Game Lab and is based on the request of Taisto Suominen to look into the visual scripting tools found in Unreal Engine and on PlayMaker plug-in for Unity.

As both engines have much in common, the thesis helps explain the differences between them and runs a test on the two visual scripting additions, evaluating success rates of the tasks and providing user stories on each task to help understand the experience and thought processes. The purpose of these tools is to provide an easier path to game development industry for those who have no experience or knowledge on how to program. This thesis provides examples and issues found in the tools from the perspective of the author, who has a very basic understanding on programming and gaming logics but has never created a running program outside the basic programming courses on C# provided by the school.

There are multiple other published works and projects that utilize some form of visually aided programming, such as Collaboration Between C++ and Blueprint in Unreal Game Engine, or Linnea Torn World game demo production, but they are usually used as a platform to develop something, rather than examine the platform itself. (Salminen, 2019; Tiilikainen, 2014) In this thesis, the author defines a basic game concept outlines and creates a game scheme while providing the success rates of each task on the engines, showing how much inexperienced coder can achieve with the internal visual tools and how often he needs to rely on the external community’s help.

The thesis goes through the background of game development engines, what visual scripting is, and how it differs from visual programming. It also explains how programming and visual scripting interact inside the provided engines.

The thesis was a single person project and the author is responsible for all the work conducted in the thesis.

2 KEY CONCEPTS

2.1 Game engines

It is often debatable when, where, and how the video game industry started, but the core principles have not changed. The concept is business combined with games, and both have been around for a long time. What makes a game a video game can have varying definitions but ultimately, it is the transition into the newest medium of our time; the virtual medium. All games need a platform to be developed on, and eventually they moved from pen and paper to virtual environments in research labs of scientists. (History.com Editors, 2017) Video games became such a large market that companies strived to make them faster and cheaper. This naturally led to an environment that could be used to develop games. By the time most households had a computer, the game industry had grown into such a large scale that it found its way to most household through them. When this happened, companies started to provide tools for the masses to develop games by providing a tool accessible for as many as possible. The easier it is for beginners to start learning, developing, and eventually publishing their own games, the more they could profit from them. (Chikhani, 2015)

Because there were not any game engines in the beginning, developers had to make their own. Subsequently, when a game proves successful, the format is repeated. By creating their own engines, the developers guarantee that their engines will do exactly what they want them to do, optimizing the performance by letting them focus on the essentials. Unlike them, the modern game engine is for everyone. At least it tries to be available for as broad audience as possible. The modern game engines are designed to offer as much flexibility in terms of development so that users can use the same engine for as many projects as possible. The key is finding a balance between flexibility and beginner friendliness. (Brodkin, 2015)

2.2 Visual scripting

Visual scripting stems from visual programming, which was originally an idea to make a text-based system easier to understand. Canfield-Smith published a thesis in 1975 that proposed that part of the complexity of programming is mentally constructing a model of a program state in your head, and he wanted to be able to show all the states on-screen, so that it could be observed all at once. This was because people simply cannot handle big capacities of complex info as computers do. This lack of capacity leads to unanticipated errors.

Because visual processing is much more effective than abstract structure thinking, our world is filled with icons and shapes to help us process things faster. This is all applied to coding in visual programming, often to help beginners understand how coding works, or to make things easier for unprofessional programmers. (Canfield-Smith, 1975) VPLs use graphical elements to display the flow of the code so that working with the whole system is displayed clearly, and they are often represented in block-based or flow-based systems. Most notable example of a block-based VPL is Scratch, which is a visual programming language. Flow-based diagrams can represent multiple different flows, like state or logic flows, and are often visual scripting languages. See Figure 1 for an early VPL example.

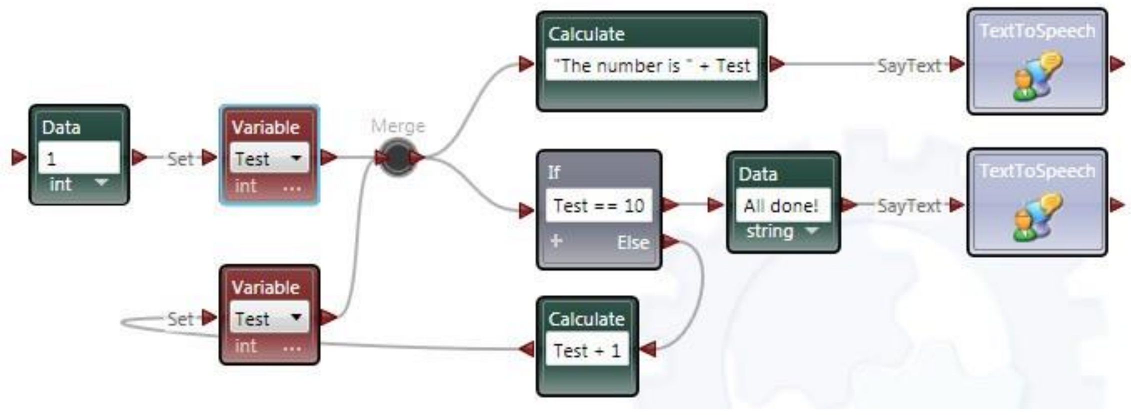


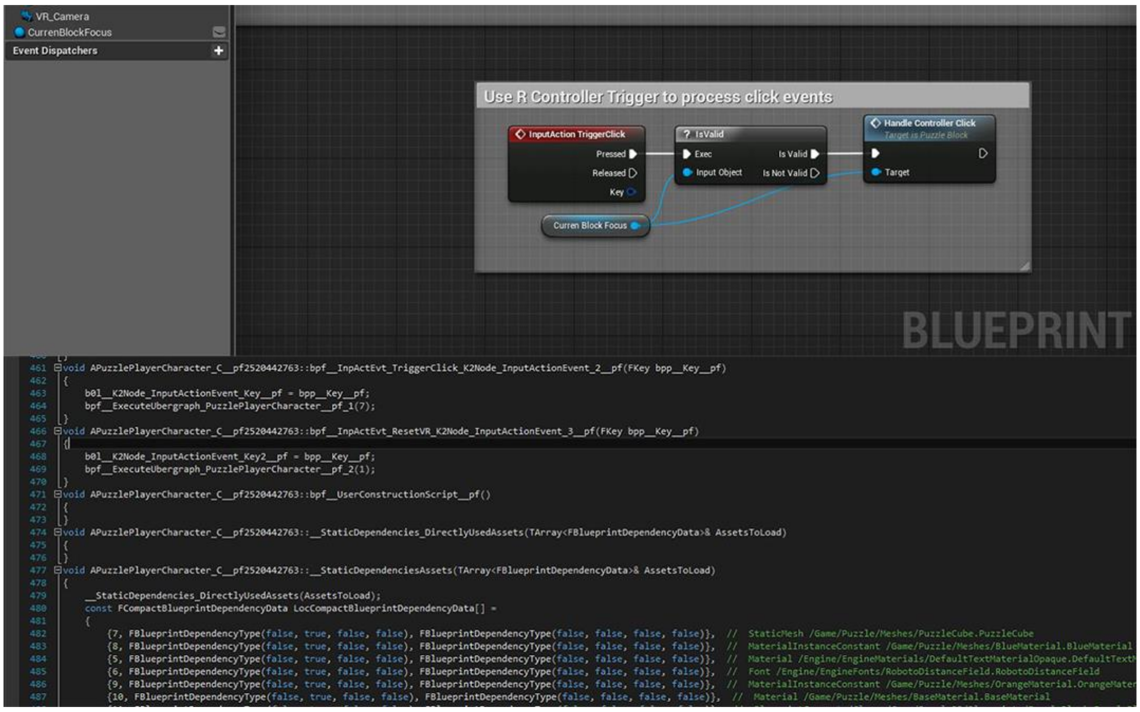
Figure 1. Microsoft visual programming language flow graph (Microsoft, 2012)

2.3 Programming and scripting

Visual scripting languages are not to be confused with visual programming languages even though they are both graphical programming environments and often share similar flow chart-like diagramming structure. VSLs are a subcategory for programming languages and the key difference between the two is that programming requires a compiler and scripting requires an interpreter. If a piece of code needs to be expressed in a VSL tool, a programmer would need to compile the code first, for example with Microsoft Visual Studio, and then import the compiled script. (Geeks For Geeks)

Most of the confusion between VSL and VPL stems from situations where people intermix classifications and call scripting languages programming languages. They are not wrong because scripting is coding, but the fact that scripting has different functionalities than regular programming, it creates confusion. All scripting is programming, but not all programming is scripting.

While Playmaker cannot directly convert scripts into C#, the user can import your C# scripts into PlayMaker. This essentially allows users to keep using PlayMaker visual FSM editor and simultaneously create their own actions to do anything the editor doesn’t provide. Unreal Engine (UE) can provide similar co-creation in Blueprints, but the language used is C++. What Blueprints can do that PlayMaker doesn’t, is a feature called code nativizing. This allows the developer to convert their Blueprints into native C++ code. See Figure 2 for a comparison example. This generated code becomes very hard to read due to its unfriendly format because it needs to translate the nodes run in the virtual machine into C++ code, resulting in extra dependencies, machine instructions, and metadata that needs to get complied. This needs to be accounted for as it can decrease the performance of the game.



```
void APuzzlePlayerCharacter_C__pf2520442763::bpf__InpActEvt_TriggerClick_K2Node_InputActionEvent_2__pf(FKey bpp__Key__pf)
{
    b0l__K2Node_InputActionEvent_Key__pf = bpp__Key__pf;
    bpf__ExecuteUbergraph_PuzzlePlayerCharacter__pf_1(7);
}
void APuzzlePlayerCharacter_C__pf2520442763::bpf__InpActEvt_ResetVR_K2Node_InputActionEvent_3__pf(FKey bpp__Key__pf)
{
    b0l__K2Node_InputActionEvent_Key2__pf = bpp__Key__pf;
    bpf__ExecuteUbergraph_PuzzlePlayerCharacter__pf_2(1);
}
void APuzzlePlayerCharacter_C__pf2520442763::bpf__UserConstructionScript__pf()
{
}
void APuzzlePlayerCharacter_C__pf2520442763::__StaticDependencies_DirectlyUsedAssets(TArray<FBlueprintDependencyData>& AssetsToLoad)
{
}
void APuzzlePlayerCharacter_C__pf2520442763::__StaticDependenciesAssets(TArray<FBlueprintDependencyData>& AssetsToLoad)
{
    __StaticDependencies_DirectlyUsedAssets(AssetsToLoad);
    const FCompactBlueprintDependencyData LocCompactBlueprintDependencyData[] =
    {
        {7, FBlueprintDependencyType(false, true, false, false), FBlueprintDependencyType(false, false, false, false)},  //  StaticMesh /Game/Puzzle/Meshes/PuzzleCube.PuzzleCube
        {8, FBlueprintDependencyType(false, true, false, false), FBlueprintDependencyType(false, false, false, false)},  //  MaterialInstanceConstant /Game/Puzzle/Meshes/BlueMaterial.BlueMaterial
        {5, FBlueprintDependencyType(false, true, false, false), FBlueprintDependencyType(false, false, false, false)},  //  Material /Engine/EngineMaterials/DefaultTextMaterialOpaque.DefaultTextM
        {6, FBlueprintDependencyType(false, true, false, false), FBlueprintDependencyType(false, false, false, false)},  //  Font /Engine/EngineFonts/RobotoDistanceField.RobotoDistanceField
        {9, FBlueprintDependencyType(false, true, false, false), FBlueprintDependencyType(false, false, false, false)},  //  MaterialInstanceConstant /Game/Puzzle/Meshes/OrangeMaterial.OrangeMater
        {10, FBlueprintDependencyType(false, true, false, false), FBlueprintDependencyType(false, false, false, false)},  //  Material /Game/Puzzle/Meshes/BaseMaterial.BaseMaterial
```

Figure 2. VSL above, nativized code below. (Unreal Engine, 2019)

This thesis does not include a nativized example from the project files due its unnecessary nature on the project. Even though it is possible to write code and compile them into usable scripts in the engines used in this thesis, this thesis doesn’t include any custom C# or C++ scrips because this thesis does not test the tester’s skills in that area, nor is the tester that fluent in programming.

A game developer will generally need to learn a coding language. Realistically, often multiple languages if they want to be professional coders in the industry. Visual scripting languages provide an easy way for developers to start working on their programs with the intuitive user interfaces and preset narrowed down tools, but they are limited. Because programming is not limited, it drives developers out from the limited environment because they want to create something more complex that the VSL cannot provide, or to create shortcuts that the VSL tools limit. Sometimes, the dependence of a developer on a public game engine’s most recent update can create issues.

The aim of VSLs is not to help developers publish games, but to lower the threshold for beginners to start their own project, and through that, learn the logic behind it. By making coding visual, the game engine developers increase the number of game developers. Many of those stick to the programs they learned it with. (Mazaika)

2.4 Topic selection

Visual Scripting tools are usually promoted towards certain types of target audiences; beginning developers with an idea but no assets or coding experience to make it concrete, people with assets they have made themselves and want to prototype but have no coding experience, or people who have coding experience and want to automate parts of their code. The forums and the online communities of these tools also provide either free or purchasable content, such as pieces of code made by professional programmers, game assets such as art or animations, to those who rely on placeholders due to their lack of visual experience, or even tutorials on how to do all that from scratch. (Unreal Engine, 2019)

Because the tester fit into most of these categories, he was considered the target audience for these video game engine developers. The tester has a long history of traditional arts, so he was often tasked with the visual aspects in the projects he took part in. In addition, an equally long history of playing games has given him an urge to develop a game himself. Due to the testers lack of programming experience, he fit the described target audience for game engines visual scripting tools. This intended targeting led the author to take a practical approach towards the tools and see how far comprehending basic concepts can take a user.

3 PRACTICAL APPROACH

3.1 Evaluation and metrics

The tester has no proper experience developing games with a game engine. In the past, the tester has created a few different scenes with Unreal Engine with plain object meshes and tried importing a sprite sheet to create a sprite animation with Unity. But the visual scripting languages are new to him. The tester has a basic idea of what the visual scripting tool looks like in Unreal Engine but has never used it.

3.1.1 Game definitions

Deriving a large inspiration from The Art of Game Design, a game is meant to provide entertainment, and that a game can be won or lost. This means that a game needs to have a goal which defines how you win, and rules that define how you lose. Based on the suggestions found in The Art of Game Design, the different objects are labelled as different objectives based on relatively wide variety of game types. (Schell, 2008)

3.1.2 Game object definitions

Because the concepts of gameplay mechanics vary immensely depending on the type and target audience, intended platform, and intended time consumption, the gameplay elements were narrowed into distinctly different ones based on some of the most popular games currently available.

The first two object types were labelled as primaries. Without them the game could easily not work or would be so limited not to provide enough gameplay.

A1) Controllable object, the player

The object the player directly has control over. These were considered to be the car in a racing game, or an avatar on a board game like a pawn in chess.

A2) Desired object, the goal

First of the few object types that really define the game itself. It can be a score, a destination, or making everyone else lose first, in which the goal is technically the player.

The second three object types were labelled as secondaries because they bring extra elements into the game such as difficulty and chance through obstacles and additional rules.

B1) Required object, the key

This object type is limited only by the imagination of the designer but usually derives from the gameplay and goal type. If the goal is a destination, the required object could be a door key or a vehicle that will grant access to otherwise inaccessible goal. The nature of this object makes the game more puzzle oriented but can easily be mixed with the goal, if acquiring the required object leads directly to the goal.

B2) Environmental object, the constraint

This object type increases the difficulty through more abstract natured gameplay element choices, such as having a limit like time, health or space, or by introducing an enemy. Environmental hazard could be managed by a random number generator introducing chance, like having randomized effects take place such as weather in open field action adventure games or having randomly generated traffic in a racing game. These all limit the actions the player can make and introduce the possibility for risk and reward.

B3) Optional object, the aid

Similar to the required object, the optional object often heavily depends on the goal and gameplay. If the goal is destination, the optional object could be a speed boost, the player can avoid possible non-player character racers or time limits, or it can be an extra life in case you fall into a bottomless pit and have to start over on a side scrolling platformer. But because of the helpful nature of the object, it is often behind environmental or avoidable objects, away from the goal.

From the categories above the final list of tasks is as follows:

1. Create the confined space that operates as the scene’s spatial restrictions (B2.1)

2. Create the player as a controllable object (A1)

3. Create the goal as a winning condition upon reaching or finding it (A2)

4. Create the key, so that the goal is not reachable from the start (B1)

5. Add time limit (B2.2)

6. Create an enemy as another spatial restriction that tries to find the player (B2.3)

7. Create the aid that removes the enemy (B3)

The game prototype needs to provide the following conditions and objects to qualify. Some games don’t have all of these and some might not even have most of these, but with the tester’s personal level of gaming experience, the author can acknowledge these as something that repeat in most games in some form or another.

Some games consider environmental objects and avoidable objects as the same thing. For example, in virtual chess, all pieces the player controls could be considered optional, environmental, and controllable objects, except the King, which would be a required object. And the enemy pieces the enemy controls could be considered avoidable, environmental, and optional objects, except the King, which would be a desired object. Regardless of how these objects could be classified, these were the object types used in the thesis.

3.1.3 Project sequence

Once the main components had been decided, the task sequence for the project was planned. The tester was adviced that learning how to do one thing could help complete another, so the object creation sequence was designed in such a way that it could help the tester as much as possible. If the author wanted to see how difficult certain tasks were from the beginning, the tasks would each need to be first attempts, but with one test that is not possible. Therefore, all previous attempts were expected tp aid the tester as much as possible by using what he would learn on the preceding tasks.

The results show how the sequence flow was affected by the success rates of the tasks, and how multiple objects ended up sharing variables, linking them together.

3.2 Evaluation

Conducting usability metrics can be fairly difficult to measure, as they can have a wide variety of requirements, dependencies, and be affected by intuitiveness. What gives the best results, are statistics, but they are hard to get because they require a lot of data from many comparable viewpoints to prove valuable. And that kind of data requires a lot of time and money. Because this work is not host a sponsored or professionally approved wide workshop, the test focused on the user experience story.

Because usability can be tested in so many ways, the metrics are narrowed down into two categories that are kept track of, and reflect on the reasons why something was or was not successful:

1. Success rate, whether the tester had to look it up

2. Error rate, how many times the tester tried and failed

Many other metrics could be considered, such as time in relation to errors, or how optimal the choices were, but these would be much more valuable if the author coult test people with categorized backgrounds and similar levels of experience to create a comparison between the results. For final progression flow, see Figure 3.

Using time as one of the main metrics was considered but because the focus was on seeing whether the job can be completed without assistance or not, it would create uneven results as some of them can take a lot longer time than others through sheer amount of clicks required to complete them. Furthermore, speed is not a performance specification that VST counters. They provide as low and as informative learning curve as possible while making the tool itself as flexible as possible, rendering time relatively useless in terms of usability design.

Usability metrics for satisfaction, whether it was task or engine specific, could been evaluated if the VSTs were large in numbers and they had to compete for the same user space on the market. Additionally, that kind of test would require a fair number of users.

Instead of using completion rate and effectiveness through time spent on individual tasks, it was chosen to see if the tasks were completed with or without online help, and how many errors were made. As the correct solution can be achieved in different ways and can vary depending on my basic concepts of coding and game logic, and how the correct solution is not known in advance, the ratio between number of steps completed successfully from the total number of steps undertaken can’t be determined. However, the number of key functionalities each task contained, was compared.

Results explain the thought process behind the choices, as the previous successes enhanced the success rate of the tasks that followed. Additionally, a closer inspection on project definitions can be found in the reflection chapter.

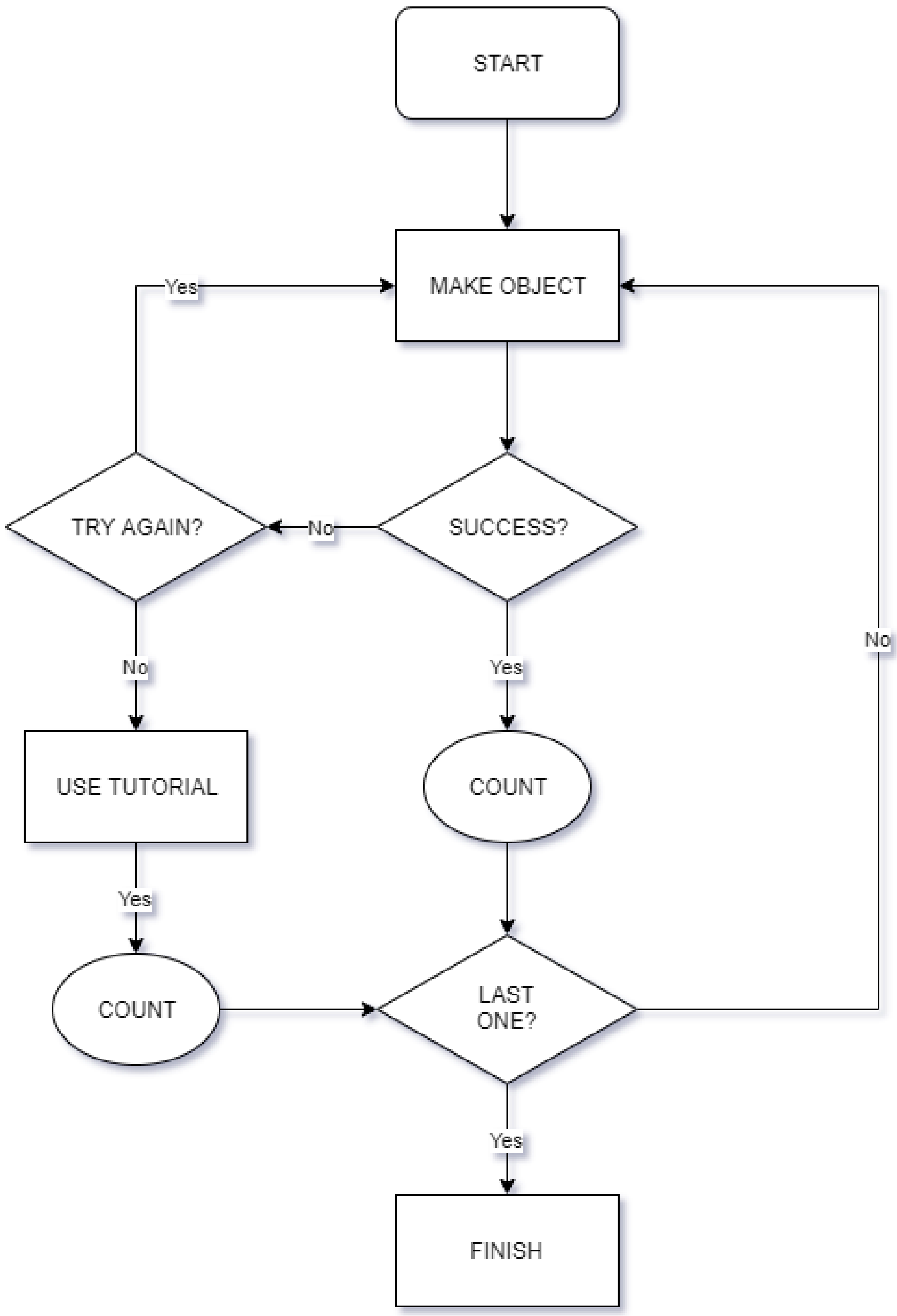


Figure 3. The workflow of the project tasks.

4 PARALLEL COMPARISONS

The purpose was to examine the reasoning behind choices and discoveries, so that the same task setting can be duplicated and directly compared to this thesis. All quotations in this thesis are thoughts of the tester.

> I found that one of the main features in both Unity, and the Unreal Engine game engines, was how most assets could be placed into the scene as placeholders with no scripting required. And testing out different options in the inspector tab that appear when clicking on the objects, providing massive amount of adjustable details. On one hand, this Unreal Engine's user experience felt very welcoming due to its toybox feel, where one could pick up and place wide selection of objects in to the scene I wanted, but on the other hand, the amount of info previewed on the inspector/details tabs and the provided selections in action and component tabs felt extremely overwhelming. Unity was more beginner friendly, as it didn't include all variations of information, only the bare essentials.

See Figure 4 for engine object creation comparisons and Figure 5 for detail editor comparisons.

> By following the principle of visual scripting, I attempted to place all objects into the scene at once so that I could visualize the project better and preview as many objects in the project as possible. I added a placeholder for all required objects, except the timer, because it needed to be built separately as it was an abstract object in the scene that was not a singular game object, but rather tied to the canvas. The canvas is considered what the main camera can see and depending on the location and settings of the camera, the timer can easily end up lost in the scene view.
>
> I renamed all my objects to represent the what they will be. What I found most useful about these objects, was that they had a lot of components by default, like colliders, which are required on all objects that can affect each other through normal collision. Browsing through the component lists, I found a lot of different colliders but going through them all individually was not necessary, due to these provided default colliders. Developer could easily swap out the default one and replace it with their preferred collider type.

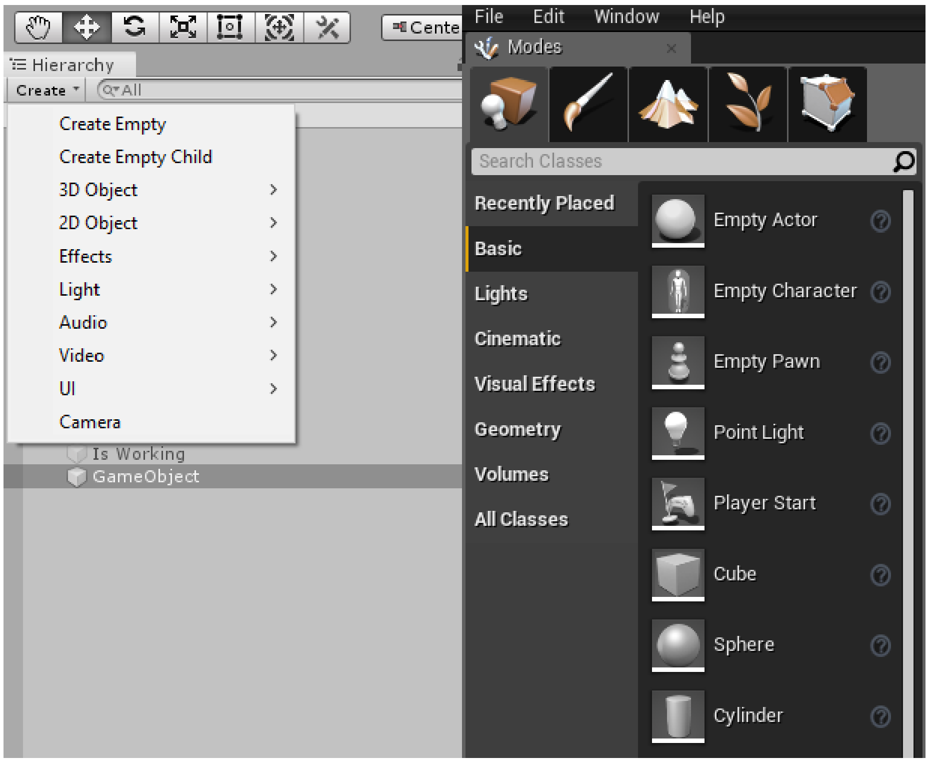
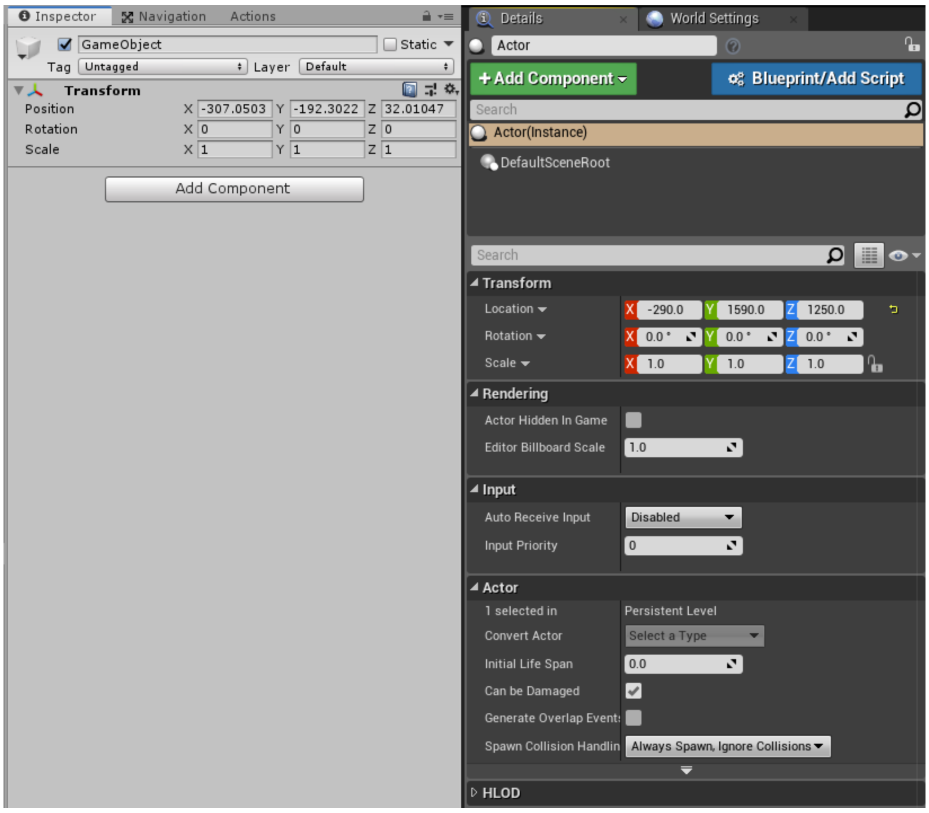


Figure 4. Object creation. Unity on the left, UE on the right.



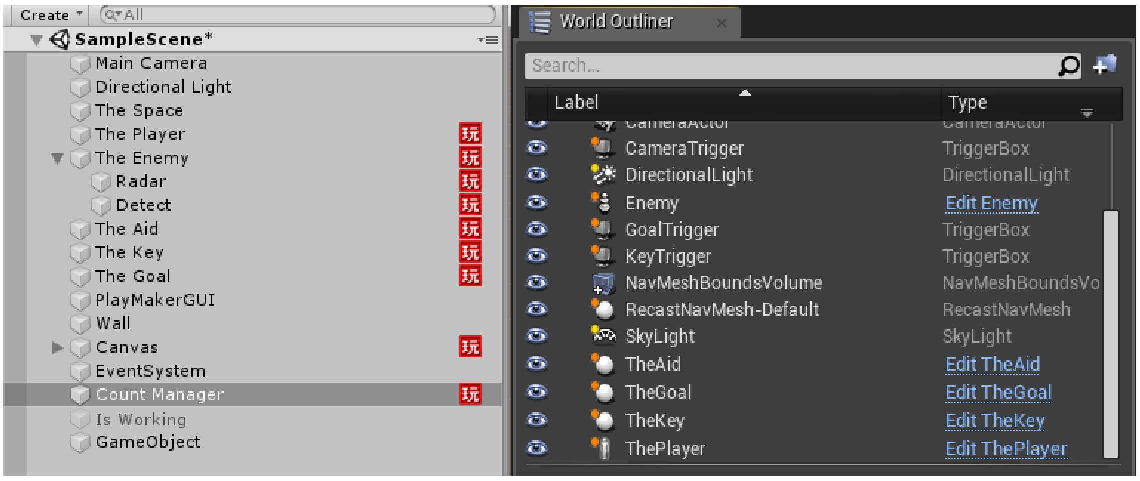
GameObject
Tag Untagged Layer Default
Transform
Position X -307.0503 Y -192.3022 Z 32.01047
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Actor
Transform
Location X -290.0 Y 1590.0 Z 1250.0
Rotation X 0.0° Y 0.0° Z 0.0°
Scale X 1.0 Y 1.0 Z 1.0
Rendering
Actor Hidden In Game ☐
Editor Billboard Scale 1.0
Input
Auto Receive Input Disabled
Input Priority 0
Actor
1 selected in Persistent Level
Convert Actor Select a Type
Initial Life Span 0.0
Can be Damaged ✓
Generate Overlap Events ☐
Spawn Collision Handlin Always Spawn, Ignore Collisions

Figure 5. Detail editor. Unity on the left, UE on the right

By adding an FSM to the game object, the engine provides a start and state 1 in the FSM editor, where the logic flows instantly when the game starts. They light up to help guide the flow of the logic, making it very easy to understand. Renaming and commenting on the states makes it even more easy. When a game object is given a finite-state machine (FSM) component, it is visible on the scene’s object list and indicated by a small red icon. The icon is also visible on all game objects in the scene itself. Unreal Engie informs of added blueprints on game objects by turning them into highlighted links.

See Figure 6 for comparisons between game object lists.



Create All
SampleScene*
Main Camera
Directional Light
The Space
The Player
The Enemy
Radar
Detect
The Aid
The Key
The Goal
PlayMakerGUI
Wall
Canvas
EventSystem
Count Manager
Is Working
GameObject

World Outliner
Search...

Label	Type
CameraActor	CameraActor
CameraTrigger	TriggerBox
DirectionalLight	DirectionalLight
Enemy	Edit Enemy
GoalTrigger	TriggerBox
KeyTrigger	TriggerBox
NavMeshBoundsVolume	NavMeshBoundsVo
RecastNavMesh-Default	RecastNavMesh
SkyLight	SkyLight
TheAid	Edit TheAid
TheGoal	Edit TheGoal
TheKey	Edit TheKey
ThePlayer	Edit ThePlayer

Figure 6. Lists of game objects and their visual scripting representations. Unity on the left, UE on the right.

Since the Blueprints are not a separate plugin, there is no need to install anything. And in direct comparison to Unity, UE has a library of game assets for different game types that can be selected depending on the game type. This alone makes UE more attractive for beginners. But these assets are not used in this project. Additionally, UE provides categorized internal tutorials that the user can utilize instead of looking information online.

5 PLAYMAKER PROCESS

The test started with Unity because the tester had less experience using it.

> Because my initial component placements did not to help me achieve complete game objects, I started adding components from the inspector tab. And once the PlayMaker plugin was installed to the project, I was able to add it to the project view and find the associated components in the provided list. Once an FSM was added as a component to a game object, the editor would automatically set up a default state which executes as the game begins to run.

See Figure 7 for project layout in Unity. PlayMaker workspace can be placed in any location.

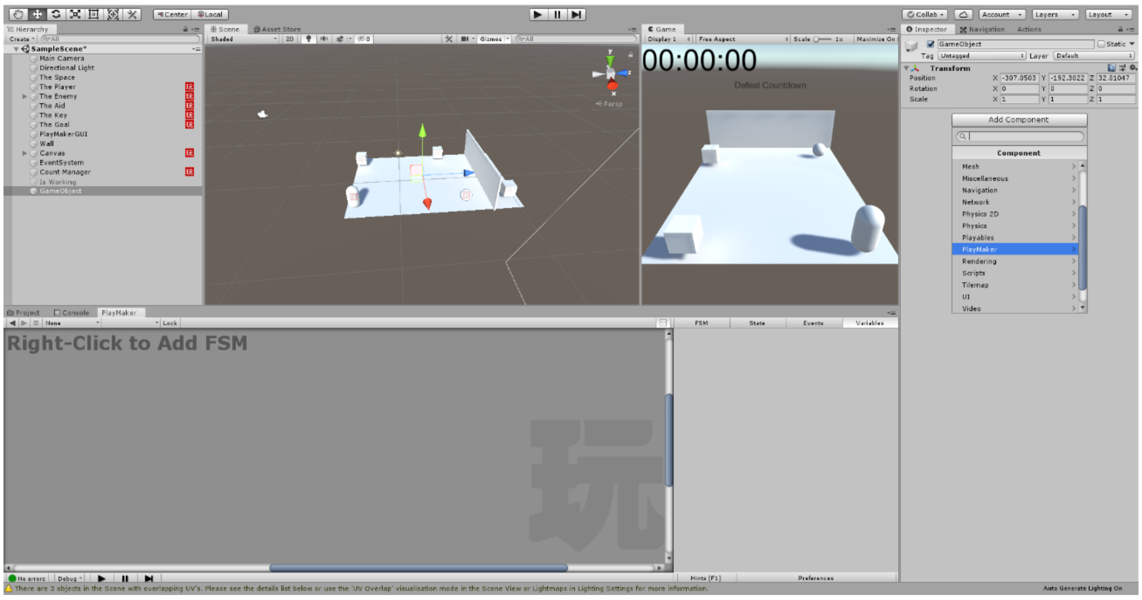


Figure 7. Project view on Unity.The box on the bottom-left is the PlayMaker workspace.

5.1 Confined space

Create the confined space that operates as the scene's spatial restrictions (B2.1).

> The plane was easier to place in the scene than assumed. It was found in the default list of objects and was easily manipulated. Right clicking the hierarchy window felt like the intended UX but the same list is found in the toolbar. To add functionality to the confined space in order to tie the visual scripting to it, I had the space separated into two separate areas, of which the latter has the goal, and which could only be accessed by jumping on the required object. Once the space was laid out, it was time to add scripts and more components to the objects.

The task was considered a success.

5.2 Controllable object

Create the player as a controllable object (A1).

> Scripting could be done in two ways; by creating a new C# script for the project and adding those scripts as components to the objects, or by adding finite-state machines directly to the objects with PlayMaker plugin. This project does not create any new scripts with C#, so the only options are FSMs and components.
>
> Initially, I attempted to create mouse-based movement scheme for the player object. The mouse-based navigation was easier to make due to having less variables. But as the other game objects ended up requiring the use of mouse, I switched to using arrow key based navigation system. I was not able to make the logic work without online help.
>
> The tutorial revealed that setting global variables first is required, which then are found by set actions, and translate that info. This could be done in few simple ways, for example through button clicks that set X and Z axis values, or by setting properties to 3D vector values that follow mouse clicks.
>
> I also attempted to create an FSM that would indicate the enemy collision, effectively creating a health bar for the player, or indicating that the player is hit and therefore, has lost the game. I was able to set the player as a global variable

> for the other game objects to refer to but was not able to make create the logic flow from the player object to the enemy object and vice versa.

The task was considered as two successes and two failures.

5.3 Avoidable object

Create an enemy as another spatial restriction that tries to find the player (B2.3).

> I was not able to make the enemy object seek out the player object from the get-go. After consulting an online tutorial, I was able to make a finite-state finite-state machine called findObject that could locate objects on the scene. But once I understood how the mechanic behind it worked, I was able to make identifyObect FSM to tell the player object apart from the other objects in the scene. Enemy object ended up having two extra game objects, labelled as Radar and Detect. Both FSMs are children to the enemy parent class, and the both had different variables.

The task was considered as one success and one failure.

5.4 Optional object

Create the aid that removes the enemy (B3).

> This game object was one of the tasks that were simple to make due to their lack of variable usage. When selecting the starting state for the FSM, the plugin provides a default list as a quick selection of common transitions between states. That list was very easy to understand and the events simple to label. Once I learned how this game function was achieved, I was able to use it to complete other tasks. The aid has to be clicked, and that input leads to an event that destroys the enemy object.

The task was considered as a success.

5.5 Required object

Create the key, so that the goal is not reachable from the start (B1).

> The key was one of the other two objects that shared the structure style with the aid, in such way that the aid was waiting for a mouse input that would transition into a state that destroys a game object. This time, the game object of choice was an additional piece of terrain, that blocked the player from reaching the goal directly from the start.
>
> This part enhances the previous design of the first task. I found a tab labelled Navigation where I was able to utilize a tool called Bake, which defined a navigational mesh for selected object, effectively not letting the player walk off the game space or around the wall that was placed into the scene. This wall blocked the player view from the goal and prevented the player from interacting with it in any way, providing more freedom to make the goal. The player would need mouse input on the aid object, triggering another destroy object command, this time targeting the wall.

The task was considered as a success.

5.6 Desired object

Create the goal as a winning condition upon reaching or finding it (A2).

> The goal was the third object that followed the same logic as the aid and the key but instead of destroying a game object, it activated a new one. The transition state of the FSM would reveal an invisible text object in the scene stating that the game has been won.

The task was considered as a success.

5.7 Extra constraint

Add time limit (B2.2).

> I was able to create a timer that counted time, but not a countdown that would indicate when the game was lost. The timer functioned with a few state actions that used defined time variables and converted them into strings that were placed into the UI as text. The counter manager required an action that compared integers and a separate property setting action that I was not able to come up with. Once the countdown reached 0, the game scene background turns red, indicating defeat.

The task was considered as one success and one failure.

5.8 Additional notes

The total amount of attempts for each task varies so much that they could not be counted. A large amount of time twas consumed rying to figure out how to make the enemy find the player, but extremely little time creating the goal due to learning to replicate a FSM style on multiple game objects to meet thr needs. Because the number of FSMs, states, events, and variables could be counted, they are compared with the results from blueprints.

6 BLUEPRINTS PROCESS

The tester has had previous experiences constructing a scene with the Unreal Game editor. Additionally, the experiences gained during the first part of testing enabled the tester to have a efficient start. See Figure 8 for the project layout in UE, and Figure 9 for the Blueprints window layout.

> Since the Blueprints are not a separate plugin, there was no need to install anything. I created a new project with no started content and once I found my project content browser below, I added a folder labelled as Blueprints. The editor had all the same elements as Unity, including game scene, game object list, object creation list, and the component information. The only clear difference was UE replacing the visual scripting editor with content browser, as UE handled the VSL in a separate space.

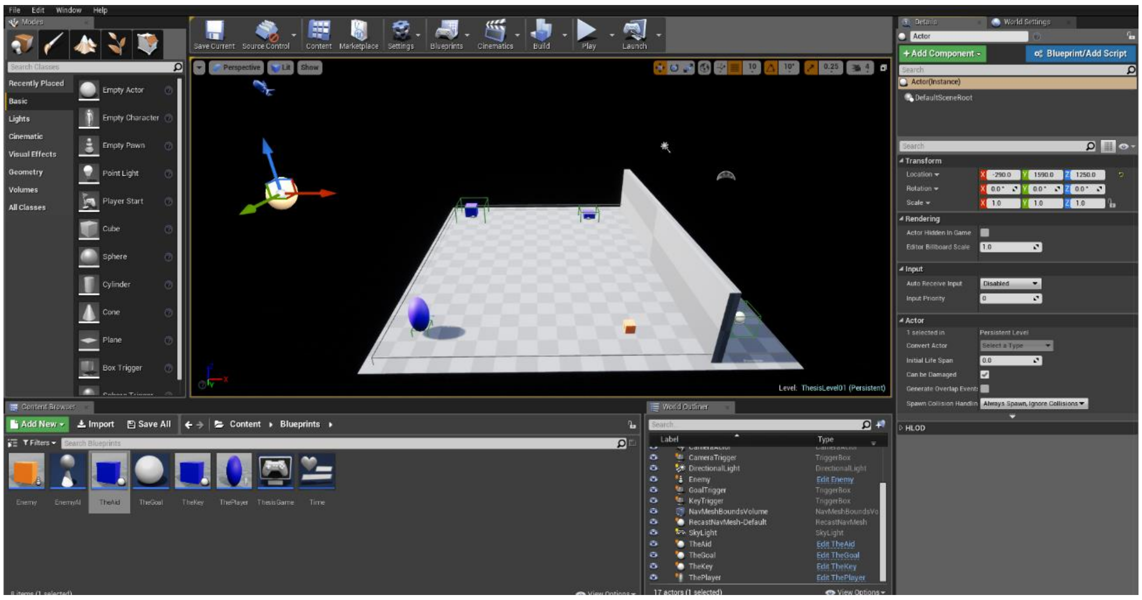


Figure 8. Project view on UE.

> The Blueprints opened in a separate window and allows the user to add all Blueprints as separate tabs in that window. With two monitors, having a separate workspace for this was very efficient, leaving more space for the UEs larger user interface. And arranging the nodes was very simple, allowing me to keep the graph neat and easy to read.

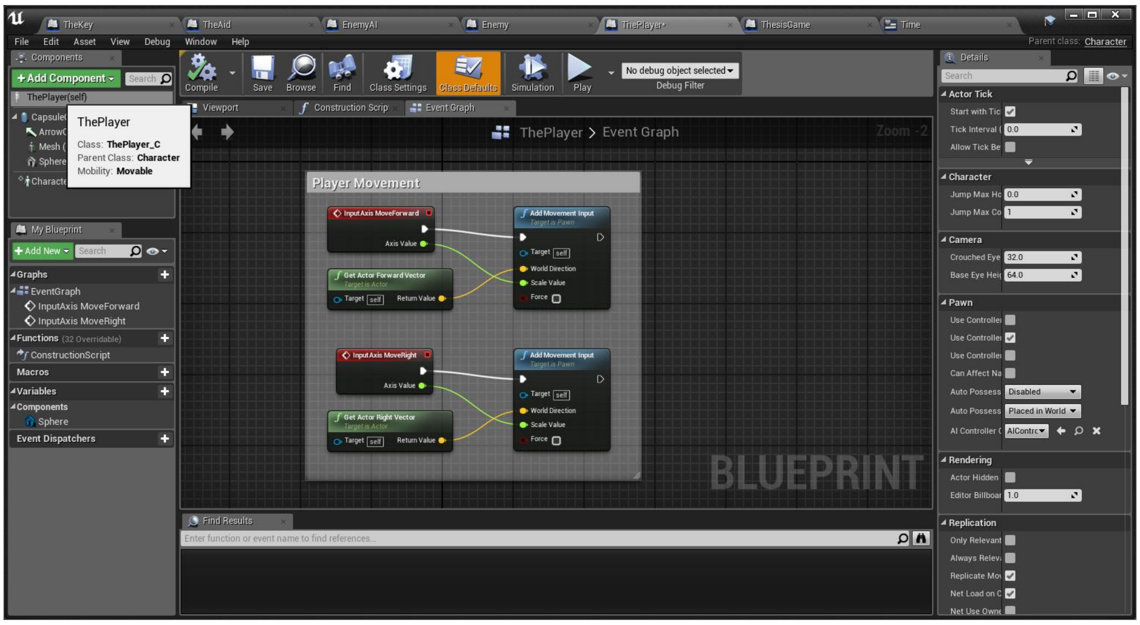


Figure 9. Blueprints UI.

6.1 Confined space

Create the confined space that operates as the scene’s spatial restrictions (B2.1).

> Creating the space was even faster than on Unity, as the game objects were presented on the default engine view. And with the base knowledge to seek out some level of navigation system, I was able to find a game object labelled as NavMeshBoundsVolume. At this stage, I did not know that it allowed certain types of object classes to have enabled pathfinding, but I figured it would work in a similar fashion.
>
> I followed the same principle of blocking the goal with a wall, that would get targeted by the key object’s Blueprint. I was also required to add an external camera to the scene where the game view would lock because the default playtest kept initiating a run based on where the view on the scene was at any given time, effectively not letting me work outside the plane or it would drop the player object instantly off the scene. But by adding an extra camera object to the scene, and later figuring out a trigger box object functions, I was able to instantly move the camera view to the added camera by placing the trigger box directly under the player object in the scene. This allowed me to create the same view as in the Unity project.

This task was considered a success.

6.2 Controllable object

Create the player as a controllable object (A1).

> From the knowledge gained from Unity, I was doubtful of successfully creating the player object. Especially, since the previous experiences with UE has contained the started materials which include a playable first-person view game object that allows you to test the environment. Surprisingly, the set-up was very easy to make with the previous information. I knew to add components for axis inputs, one that allowed movement forward with a positive or negative value, and another allowing movement to the right and left respectively. Additionally, I found that I can set these

values in the project settings, that allowed me to set input from a wide variety of game controller selections.

This task was considered a success.

6.3 Avoidable object

Create an enemy as another spatial restriction that tries to find the player (B2.3).

The enemy was divided into two separate Blueprints; the one that moves the object, and the one that hits the player. In contrast to PlayMaker, Blueprints allowed me to pick a direct game object in the scene to use as a reference point for the enemy object to move towards. And the details allowed me directly to set the speed of the action. It was all very simple.

I was not able to create the Blueprint that detected a hit on the player object. With online tutorial help, this script had a node called Sequence, that allows the developer to divide the event flow with conditions, and these conditions were hard to make. The sequence ended up containing an on-hit condition that destroyed the enemy object, turning it into a visible animation indicating player’s defeat.

The task was considered as one success and one failure.

6.4 Optional object

Create the aid that removes the enemy (B3).

The aid followed a similar path with the key and the goal, as they did on Unity. Except that on UE they had a trigger boxes placed on them, indicating a collision with the player object instead of using a mouse input detection. This is a key game play element difference between the two projects, indicating that the project can be approached from different angles. Once the player object enters the trigger box placed on the aid, it destroys the enemy component from the scene.

The task was considered a success.

6.5 Required object

Create the key, so that the goal is not reachable from the start (B1).

> The key functioned exactly like the aid and destroyed the wall component from the scene. No major difficulties.

The task was considered a success.

6.6 Desired object

Create the goal as a winning condition upon reaching or finding it (A2).

> The goal functioned exactly like the key and the aid, except that used a similar animation to indicate collision between its trigger box and the player object.

The task was considered a success.

6.7 Extra constraint

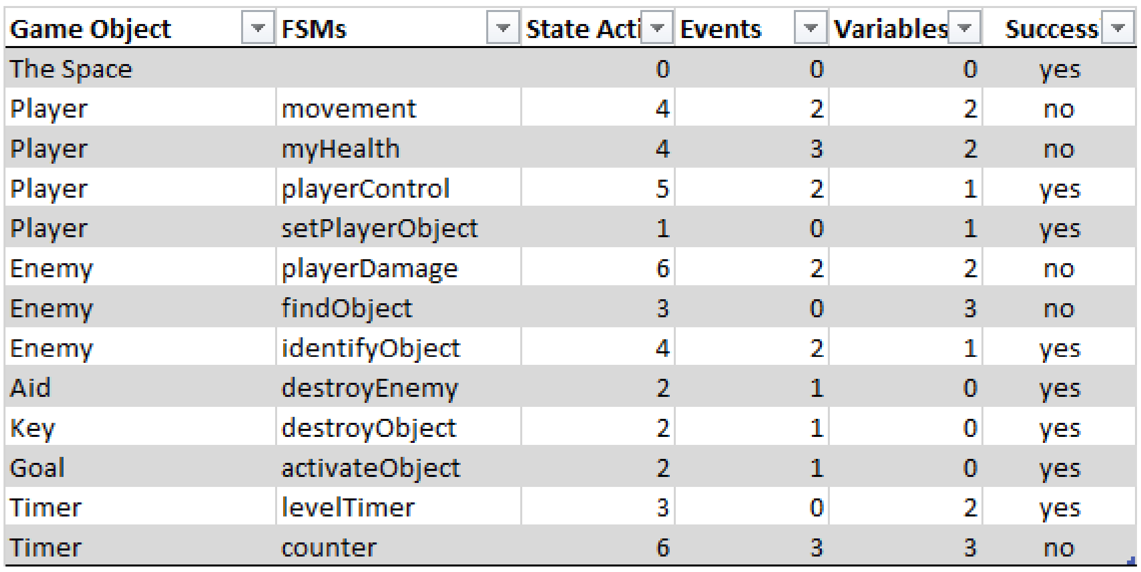
Add time limit (B2.2).

> The timer proved to be the most difficult part of the UE project. I was only able to add a timer widget to the scene but not create a Blueprint for it. The widget was easy to add through the Event BeginPlay suggestions, but I had to look online for a tutorial to create the event tick that counted and defined minutes and seconds for the timer. And because the timer already had used its Event BeginPlay, I did not know how to add another one. Tutorial showed me a way to do this through editing the level Blueprint itself, which I did not know existed. Though there, I was able to set the timer on a separate text block that was visible in the game view, similarly to the Unity project.

This task was considered as one success and two failures.

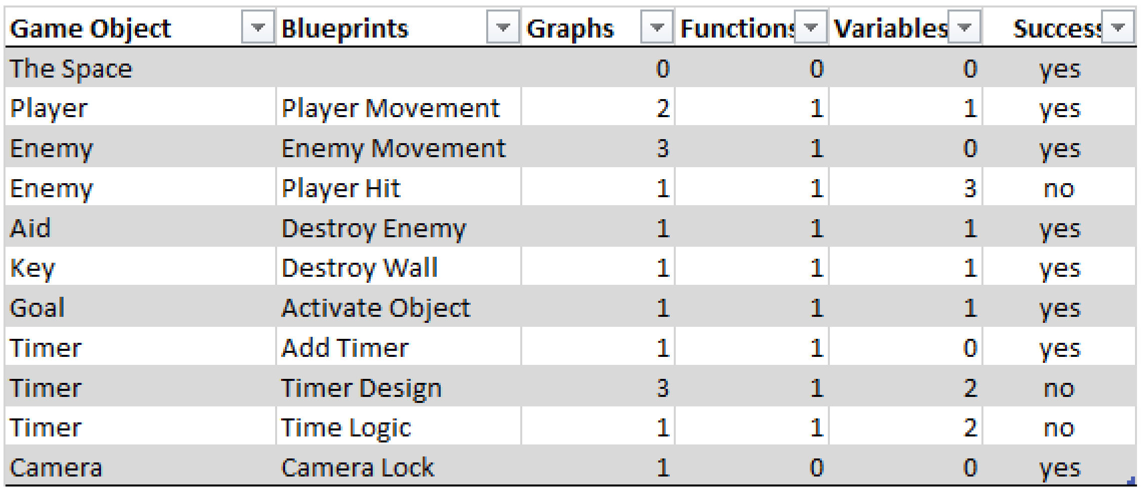
7 DATA REFLECTION

Even though both engines provided a different approach to visual scripting, they both provided comparable statistics. Unity had state actions and events, and UE had graphs and functions, and both used variables. The parts that functioned similarly on both engines were taken into consideration and compared the numbers with each other, resulting in very clear difficulty curve. See Figures 10 and 11 for the statistics.



Game Object	FSMs	State Acti	Events	Variables	Success
The Space		0	0	0	yes
Player	movement	4	2	2	no
Player	myHealth	4	3	2	no
Player	playerControl	5	2	1	yes
Player	setPlayerObject	1	0	1	yes
Enemy	playerDamage	6	2	2	no
Enemy	findObject	3	0	3	no
Enemy	identifyObject	4	2	1	yes
Aid	destroyEnemy	2	1	0	yes
Key	destroyObject	2	1	0	yes
Goal	activateObject	2	1	0	yes
Timer	levelTimer	3	0	2	yes
Timer	counter	6	3	3	no

Figure 10. Success rates of each FSM for Unity project and its components.



Game Object	Blueprints	Graphs	Function	Variables	Success
The Space		0	0	0	yes
Player	Player Movement	2	1	1	yes
Enemy	Enemy Movement	3	1	0	yes
Enemy	Player Hit	1	1	3	no
Aid	Destroy Enemy	1	1	1	yes
Key	Destroy Wall	1	1	1	yes
Goal	Activate Object	1	1	1	yes
Timer	Add Timer	1	1	0	yes
Timer	Timer Design	3	1	2	no
Timer	Time Logic	1	1	2	no
Camera	Camera Lock	1	0	0	yes

Figure 11. Success rates of each Blueprint for Unreal Engine project and its components.

These numbers clearly indicate that it was difficult for the test person to complete a task if it had multiple variables in it, and that translates directly into the inexperience of programming. In both projects, There were more successes than failures, but in terms of variable use, the statistics are close to even. Figures 12 and 13 show the sum of variables by game object and it is clear that each task containing at least one failure are the game objects that have higher count of variables. It can be seen on both figures how there were problems with the enemy and timer objects. Oddly enough, only the PlayMaker player object proved difficult despite its lack of variables in comparison to Blueprints.

Another fact proven by the statistics, is that it was possible to create over half of the intended tasks with ease on both engines. This is very good result when compared to traditional programming, which the tester had very little experience of.

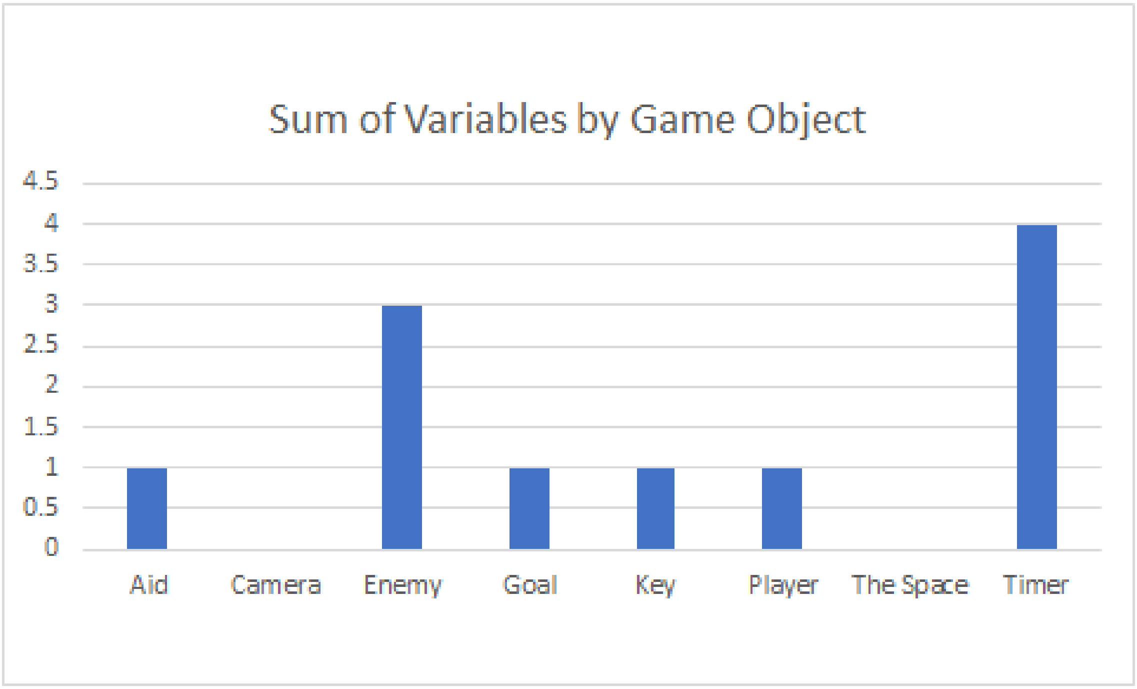


Figure 12. PlayMaker variable count by game object.

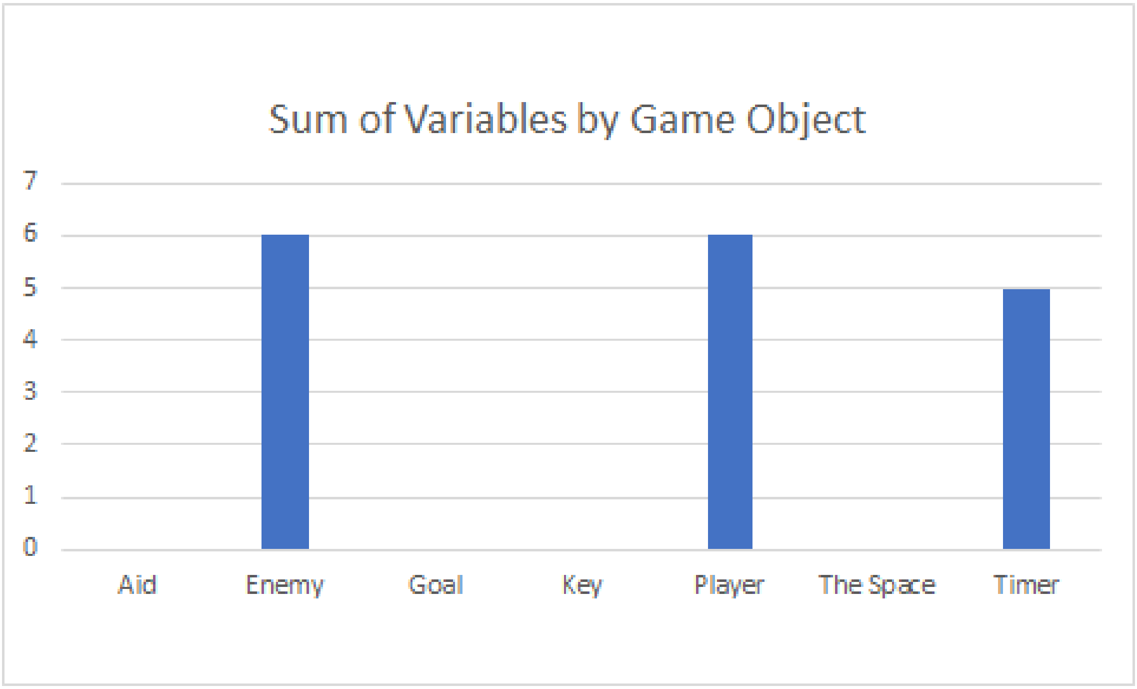


Figure 13. Blueprints variable count by game object.

8 PROJECT REFLECTION

Through this demonstration, it is evident that the issue was in the inexperience of programming. As the number of variables went higher, the number of successes went down. Variables are what hold the values which the events use. If the user doesn't have the required programming logic necessary to create a working program flow, it can turn into an insurmountable problem. But with the help of tutorials provided by the online community it can be done. But following tutorials won't help if the user doesn't understand the core concepts. Following a tutorial will be difficult without having an idea of what the tutorials are saying.

I would not recommend visual scripting to a person who has no prior knowledge on programming. But only if they wanted to develop a game only using a VSL. The logic is too complex without basic concepts. On the contrast, an artist who wants to test out a game asset they've made can very neatly execute it in this kind of environment. Adding an animation to a single object does not interact with multiple other events in the scene and remains a very simple test.

If a professional programmer using either of these tools wants to offload their work to the other members of the game development team, it is much faster to connect nodes together than it is to write code. Keeping the node flow clean is also much easier by quickly arranging them around, in comparison to writing clean code, which can be very difficult for an outsider to look at.

The experience with tutorials was overwhelmingly positive. It is not hard to imagine why these engines are so popular as they really help users test their imagination and the limits of the tools. Even though it is clear that my experience level does not match the amount required to develop a game, it certainly has advertently increased my interest in the field. As one of the target audience type for these VSL tools was a developer who wants to learn to use an engine, it is very easy to get involved through provided online tutorials and the flexible, yet slightly daunting UI.

My definitions for game objects can be considered rather limited as games can work in so many ways. As an example, one of the biggest names that used PlayMaker in its development is Hearthstone, a card game. And a card game that contains a deck with wide selection of cards is essentially a game based on luck and tactics. In that game, the goal would be getting enemy player’s health to 0 before they do the same to you, but the entire concept of having an aid or a key object is entirely lost. There isn’t even a proper player object. Games like that differ from the majority of game concepts and didn’t mix well with the other game object types.

Another PlayMaker example is an indie success called Hollow Knight. That game did provide all the elements I covered in my concepts. The only major difference was dimensions, as Hollow Knight is a 2D side scroller. This wouldn’t have changed anything else, except the camera placement.

There isn’t a very large selection of hit games developed with UE Blueprints. I think this is partly because the developers of UE, Epic Games, has focused on presenting the Blueprints as a supportive tool for the engine, not used exclusively. It’s hard to say why but I think it feels more like an educational tool for those who want to learn how to code, as advertised. Some sources had found the tool too limiting or slow to use professionally.

9 CONCLUSION

This thesis attempted to find out why the two chosen game engines offer a popular development environment using their visual scripting language tools and the results offer a very clear message. They significantly increase the possible progression when compared to traditional programming. Even though the author did not provide any related statistics, he stated that without VSL the tester would have instantly had to resort to online tutorials. As a result of using a VSL on either game engine, the tester successfully managed to create over half of the desired tasks without any external help. Considering how easy it was to consult online tutorials through the engines themselves, it is a very plausible claim that the tools help developers get into the field and learn coding.

If this project would develop further, it would most likely require a workshop set up with a range of skill and experience levels going though comparable tasks because the greatest short coming of this thesis is the data of one user. Equally important would be to add more variation to task evaluation, like time or user experience rating. There aren't enough observations to study the results in a significant scope in this thesis, but it could provide the push needed for someone considering the option. Regardless of how crucial understanding programming logic is from a game development standpoint, it is vital to have a drive to test things and learn in the process. Providing an environment for that is what these engines succeed at.

REFERENCES

Brodkin, J. (2015) Should Developers Still Pay for Game Engines

https://insights.dice.com/2015/05/01/should-developers-still-pay-for-game-engines/ [Last Access 4.12.2019]

Canfield-Smith, D. (1975) PYGMALLION: A Creative Programming Environment. Stanford University.

Chikhani, R. (2015) The History Of Gaming: An Evolving Community https://techcrunch.com/2015/10/31/the-history-of-gaming-an-evolving-community/ [Last Access 4.12.2019]

GeeksforGeeks documentation (n.d.) What's the difference between Scripting and Programming Languages?

https://www.geeksforgeeks.org/whats-the-difference-between-scripting-and-programming-languages/ [Last Access 4.12.2019]

Godot Engine documentation (2014-2019) What is Visual Scripting

https://docs.godotengine.org/en/3.1/getting_started/scripting/visual_script/what_is_visual_scripting.html [Last Access 4.12.2019]

History.com Editors (2017) Video Game History https://www.history.com/topics/inventions/history-of-video-games [Last Access 4.12.2019]

Mazaika, K. (n.d.) Why You Should Learn Multiple Programming Languages http://blog.thefirehoseproject.com/posts/2-1-learn-multiple-programming-languages/ [Last Access 4.12.2019]

Microsoft documentation (2012) VPL Introduction

https://docs.microsoft.com/en-us/previous-versions/microsoft-robotics/bb483088(v=msdn.10)?redirectedfrom=MSDN [Last Access 4.12.2019]

Salminen, M. (2019) C++:n ja Blueprintin yhteiskäytäntö Unreal-pelimoottorissa

Schell, J. (2008) The Art of Game Design. USA: Elsevier.

Tiilikainen, V. (2014) Linnea Torn World -pelidemon tuotanto

Unreal Engine documentation (2019) Blueprints Visual Scripting

https://docs.unrealengine.com/en-US/Engine/Blueprints/index.html [Last Access 4.12.2019]