# Final Year Project Report

**Full Unit - Final Report**

---

# CS3821 - Building a Game

Christopher Buss

---

A report submitted in part fulfilment of the degree of

**MSci Computer Science (Software Engineering)**

**Supervisor:** Dave Cohen


Department of Computer Science
Royal Holloway, University of London

26th March 2021

# Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 15696

Student Name: Christopher Buss

Date of Submission: 26th March 2021

Signature:

# Contents

# Abstract

Triple-A (AAA) is an informal classification that is used to describe games developed by larger teams, typically with higher than average development and marketing budgets [1]. Due to this, the approaches used by large scale teams can differ highly in comparison to a smaller games studio. This project intends to provide an overview of how AAA studios are developing games in contrast to miniature games and what approaches are used to expand further an evergrowing 160 billion dollar industry[2].

This project's core contributions are an understanding of why game engines are built in the way they are, how traditional software engineering approaches apply and do not apply to video games, and to develop a game that reflects on that which discussed in the report.

# Project Specification

**Aims:** to research how AAA games studios have different priorities compared to other software industries and to implement key concepts in a game using the Unreal Engine

**Background:** The Unreal Engine is one of the world's most powerful game engines and is also becoming increasingly popular for use in Architecture, Film and Television, and Training and Simulations. Despite this, there is very little information that focuses on the Unreal Engine and software engineering principles. This project aims to provide an overview of software engineering principles used to create better games using the Unreal Engine.

A successful project will give a user of the Unreal Engine a better understanding of how to approach game development using mainstream software engineering methodologies. A successful project will also give a software engineer a base on applying critical topics of their discipline when learning to build games using the Unreal Engine.

### Final Deliverables

The final game will at least:

1. have the key game **design patterns** described within the report
2. be **well documented** and tested using **software engineering testing** approaches applicable to the Unreal Engine
3. have a full **object-oriented implementation** using the full **software engineering lifecycle** to implement and develop features

The report will:

4. analyse the technical differences between different game engines and their approaches towards enabling game development
5. explore the major **software engineering methodologies, tooling, and processes** used by the industry, namely version control, different approaches to software testing and continuous integration
6. show examples of crucial game **design patterns**, how they apply to the Unreal Engine, and where applicable, how the pattern was implemented when creating the Unreal Engine

# Chapter 1: **Introduction**

Many different tools and methods are used to create fast-paced experiences, vast open-worlds, and story-driven adventures, but not all software engineering approaches can be applied to game development. Many factors go into making games. Art, music, software development and everything in between is an equally important part of the game development process. This process means that the techniques used to complete games can differ from traditional software development procedures.

## 1.1 Aims and Goals

This project is intended to be used to gain knowledge and information regarding the practices and methodologies used within the AAA games industry. This will be done by:

- examining the different priorities of AAA game studios in contrast to smaller studios
- seeing what tools and methodologies are used to facilitate these differences
- understanding how software engineering processes allow for these discrepancies to be carried out
- implement these methodologies into the Unreal Engine building a small game to get a better understanding of how they are used
- building a simple "game of tag" using Unreal Blueprints, Unreal C++ and Unity to compare and contrast the merits and approaches used in different engines

The final goal of the project is to build a game in the Unreal Engine that puts into practice elements analysed and discussed within this report.

## 1.2 Project Motivation

This project is focused on AAA game development, and as such, is intended to be helpful for those who are interested in pursuing a career in software engineering within the video games industry. The information provided will also be helpful for Computer Science students and faculty who wish to have an understanding of how Computer Science principles can be applied to the Unreal Engine and the differences between the Games industry and the software engineering industry at large.

## 1.3 Survey of Related Literature

This section intends to review existing literature that has been used as a basis for the research of this report. It covers some of the most important and common sources that were used while also giving a critical analysis of the background material that has been provided by the authors of said sources.

The use of Perforce has been widely associated with the AAA gaming industry since its inception, mainly due to its efficiency in the versioning of binary files. In this report, it was set out that a comparison would be made between that of Perforce and Git to find the significant reasons as for why Perforce is typically used and if there is still a case for its use in the present day. The State of Game Development Report by Perforce (2020)[3] states that 56% of participants used Perforce's Helix Core for their version control needs. However, this is a report taken by Perforce themselves, who have a strong interest in the success of their product. It is also a report likely to have been carried out by participants who already had prior knowledge of Perforce. In light of this, it was paramount to find sources from competing companies that had stakes in differing version control software or those who had nothing to gain from the success of either platform. Several articles (e.g. [4]) have been posted by Atlassian (owner of Bitbucket, a Git VCS platform) on their blog covering the demerits of Perforce, which help to provide a counterargument to the use of Perforce. Nonetheless, there is a lack of information on the merits of Perforce versus Git (aside from brief opinion pieces) from sources that do not have a strong interest in the success of either version control software.

Developers are conditioned to continuously strive towards improving their practices, as well as being encouraged to adopt new approaches that can help to develop new systems and software. The use of Design Patterns in the software engineering community is well-established; despite design patterns being around since 1977 as an architectural concept by Christopher Alexander, Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (1994)[5] is an essential discussion regarding the potential and drawbacks of object-oriented programming, as well as describing 23 of the most common design patterns. However, as by design, the examples in the book can be used when tackling any software engineering problem in any language, as these design patterns are generic enough that they can be applied to any software engineering principle. The consequence of this is that it can be harder to use them in context, specifically in regards to that of the Unreal Engine. This report is designed, in part, to bridge the gap between the genericness of the software engineering world with the specifics of the Unreal Engine. Another consequence of being generic is that some design patterns that can be applied to the game development space are not present, and as such, books like The Game Programming Pattern's book by Robert Nystrom (2014)[6] need to exist to help cover the gaps missed.

The Unreal Engine is considered to be a problematic piece of software to learn. As a fully-fledged solution to AAA game development, it was primarily designed to be used by industry professionals who know how game development typically works and also have a strong understanding of game engines and their development. Despite this, the Unreal Engine has been opened up to public use so that anyone who has a desire to make games can download the engine and jump straight in. Paul Gestwicki's Unreal Engine 4 for Computer Scientists' series (2017 - 2020)[7] is one of many tutorials online that helps those who do not have experience with the Unreal Engine to get the tools and knowledge that they require and is especially useful for those with a strong understanding of Computer Science principles. However, as it is designed to be used by Computer Scientists, the series is not intended to be used by those who have no experience with object-oriented programming and computer science notions. This series has been chosen as it matched the target audience of this project, even if other tutorials are available that are better suited for those with no prior computer science background.

This report is presented as a dive into how software engineering principles can be applied to the Unreal Engine. The amount of advocated for principles are plentiful; therefore, further research could be done to contrast and compare different approaches that are not used in the design of the Unreal Engine, such as object-oriented programming versus data-driven development, or how an entity-component system contrasts with the Unreal Engine's approach to data management and the merits and demerits of each. Additional research could also be done for those of non-technical backgrounds, such as designers and artists, and how their prior professional experiences can be applied to the Unreal Engine.

# Chapter 2: **Game Engines**

A game engine is a software development environment that allows people to build video games. It exists to abstract the details of doing everyday game-related tasks, like rendering, physics, and input, so that developers (such as artists, designers, programmers) can focus on the details that make their games unique[8].

This chapter intends to explore:

- the reasons why many games companies have moved away from proprietary engines
- the comparison between two of the largest game engines, namely Unity and the Unreal Engine

## 2.1 The Move Away From Proprietary Game Engines

In the past, many game studios created proprietary in-house game engines crafted purely to support the needs of their product. Though because of increasing player demand, the cost of developing new engines has increased, and as such, the use of pre-existing engines such as Unity or the Unreal Engine is becoming more commonplace[9]. These engines exist with dedicated teams, allowing for studios to not allocate time to engine development and instead can execute their creative vision for their games more efficiently with dedicated tools designed for the job.

An example of a company swapping from an in-house engine is Riot Games. Their main game, League of Legends (LoL), is one of the most played PC games in the world, with a predicted number of over 100 million active users every month[10]. LoL is built on a proprietary engine using C++, Lua, C# and a few other languages[11], which started development in 2006. The full game released in 2009.

For LoL's 10th Anniversary, Riot Games announced League of Legends: Wild Rift - a recreation of the original game, but with mobile (iOS and Android) being the target platform. For the release of this game, Riot Games decided to ditch their engine and build the mobile version of the game in Unity (see *section 2.2.3*). Brian Cho, Head of Corporate and Business Development said[12]:

> "Unity's technology enables us to focus on delivering the beloved League of Legends experience to as many players across as many platforms as possible. We want to meet our players where they are, and Unity's world-class tools and platform optimization help us achieve that. With the expert support provided by Unity, we've been able to unlock the power of Unity's technology like never before."

Typically a game engine is developed catering to the specific needs of a game. Over time, features are added to an engine that helps to streamline the development process and provides features to add new and exciting game aspects. The LoL engine does not provide functionality for porting to another platform - it never needed to before. Either the engine could be improved to accommodate, or a different game engine could be used. Riot Games decided that it was in their best interest to use Unity as a platform and recreate the game from scratch, rather than to modify their existing engine to support mobile.

Another one of Riot Games' announcements was a new First Person Shooter (FPS) code-named Project A, later renamed to Valorant. Riot Games already had teams of software engineers

that could use the existing LoL engine, with Valorant targeting the same platform as LoL (Windows, Mac, Linux), but instead opted to use the Unreal Engine as their game engine of choice.

The three main reasons for not using proprietary engines are cost, talent, and time. A blog article talking about 'The Future of League's Engine[13]' talks about how LoL's engine is a 'mess', and about the fact that how in the pursuit of shipping many things quickly, enough compromises were made to the engine that the core engine is neither engine-heavy or engine-light. Because of this, the core engine exposes both high-level constructs and low-level primitives to the scripting engine, leading to complex systems having been built on top of a simple scripting language. Riot Games plans to sort of their engine architecture, but this will take time and is costly. If instead of using their engine, they had used a licensed engine, architectural issues with the engine would be resolved by the company producing the engine - saving the games company money.

Another major reason is that LoL and Valorant cover two different game genres. Their current engine is not general-purpose enough to support the different priorities of a game with a different focus. By using the Unreal Engine, they can use an engine that was specifically designed to be used for multiple game genres and will have the tools needed for them to develop without constraint. For when the tools provided by the Unreal Engine is not enough, the Unreal Engine source code is also available on GitHub[14] (see *section H.2*), so developers can easily modify the engine to suit their game and companies needs. Moving to a different engine also allows for new talent to be hired who specifically have experience with the Unreal Engine to work on the game. Instead of having to get new developers up to date and running with a new set of tools, they can easily start working on the project, increasing productivity and reducing costs incurred subsequently.

Overall, there are many reasons for moving away from proprietary engines. Although there is a cost to using another companies engine, this cost can be much less than the cost of developing and maintaining an engine in-house. It is also easier to hire new staff that have previous experience using a certain engine, as it reduces the amount of time needed to get to grips with an engine, as well as time is not needed to be spent developing and maintaining an engine in-house.

## 2.2 An Example - Game of Tag

There are many different engines and many different approaches to game development. In this section, a small *game of tag* (inspired by *Paul Gestwicki's Unreal Engine 4 for Computer Scientists*[7] series, where he builds a game using Behavior Trees in the Unreal Engine) will be used to explore the different approaches to game development using both the Unreal Engine and Unity. This will allow for a deeper look into the advantages and disadvantages of the approaches of each engine, as well as getting a better understanding of why engines are used in the first place.

### 2.2.1 Description of Game

The game that will be implemented in each approach is simple. It uses a main 'TagPlayer' that is controlled by the user and then has three different 'TagCharacters' that are AI-controlled pawns. When the game starts, one of the TagCharacters will be set to a 'tagged' state. This character will then identify another actor in the game, and then it will path towards the target to tag them so that they are no longer the tagged player. The user-controlled character can

be tagged just like the AI, and all actors have the same interactions with each other.

For a more comprehensive overview of the implementation using images, and a step-by-step of the interactions between characters, see *Appendix B*.

### 2.2.2 The Unreal Engine

The Unreal Engine is a game engine developed by Epic Games, first showcased in the 1998 first-person shooter game Unreal[15]. The Unreal Engine allows developers to use a licensing model to create and develop games for platforms such as PC, Xbox, Playstation, and others. The Unreal Engine supports three main approaches to game development, a Visual Scripting system, the use of C++ with custom functionality, or most commonly, a mixture of both.

The following section will discuss the advantages and disadvantages of C++ and Blueprints and how to use the Unreal Engine to create games using both. It will also detail the difference between both implementations and why approaching games using both is a typical practice.

#### Unreal Engine Blueprints

As defined in the Unreal Engine wiki[16], "The Blueprints Visual Scripting system in Unreal Engine is a complete gameplay scripting system based on the concept of using a node-based interface to create gameplay elements from within Unreal Editor. As with many common scripting languages, it is used to define object-oriented classes or objects in the engine".

As the entirety of the Unreal API is available using Blueprints, full games can be developed using Blueprints only. The Blueprints system is also context-aware. This means that when a node is pulled off in the Blueprints system, only functions that can be applied to the current node will show. This helps to avoid situations where a developer tries to make functions that cannot interact with each other interact. Features like this are what make Blueprints powerful for quickly developing games.

The main issue with Blueprints is their performance. The Unreal Engine Wiki states[17], "When teams script gameplay with Blueprints, they're creating new UClasses without having to write or compile native C++ code. As a result, Blueprints work well for game teams that don't have the technical expertise of native C++ programmers. The reason that non-programmers can work strictly with blueprints because Blueprint nodes run in a virtual machine (VM), enabling them to call native C++ functions. Unfortunately, relying on a VM that translates Blueprints into native C++ code comes at a cost; and as you can imagine, translating Blueprint nodes into native C++ functions can potentially slow your game's performance per frame."

This can be somewhat alleviated through the use of The Unreal Engine's Blueprint Nativization tool. This tool can be used when cooking a game to create an executable that reduces VM overhead by generating native C++ from the project's Blueprints. This generates non-reader friendly code that typically is not suitable for reusability due to the nature of how the project is compiled. The advantage of this is that by generating native C++, the project no longer needs to use the VM to execute Blueprint code, increasing the performance of a project.

Despite Blueprint Nativization being effective, it also comes at a cost. The project setting provides two options: Inclusive or Exclusive Nativization. Inclusive means that all Blueprint classes in the project are converted to native C++. The downside to this is that the executable may not remain a size that is acceptable for the projects intended platform. The other option is to use Exclusive Nativization. This limits the final size of the executable by actively preventing unused assets from being converted. It also gives developers the ability to prevent Blueprint

assets from being converted explicitly.

From a traditional software engineering perspective, Blueprints have the negative side effect of being binary data. This means that version control systems cannot track the changes made into VCS, making it harder to revert changes where necessary. This also means that blueprint aspects of a project cannot be viewed from within a VCS and instead must be accessed from within the Unreal Engine itself, making code review difficult. For more on version control, see *Chapter 3.3*.

### Unreal Engine C++

The Unreal Engine Wiki defines the use of C++ as being "used as a base for Blueprint classes, and in this way, programmers can set up fundamental gameplay classes that are then sub-classed and iterated on by level designers."[18]. From the very start, Unreal Engine 4 has been designed with Blueprints in mind when developing on its platform.

Although Blueprints are integral to the Unreal Engine workflow, all the functionality that Blueprints call is initially implemented in C++. Due to this, all features of the engine can be achieved using pure C++, however impractical at times. This provides unparalleled flexibility when it comes to working with the engine, as even elements of the engine that are locked down can be modified to the developer's whim. A blog article from Riot Games[19] talks about how they modified the engine itself to better suit the support that they needed for their game. Features like modifying the forward renderer of the engine are not possible using Blueprints, and this is where knowledge of C++ can provide many benefits for development teams. Another main advantage of using C++ is performance. Despite Blueprint Nativization existing (see *Chapter 2.2.2*), it still does not outperform that of correctly written C++. Blueprints can also become messy when connecting many features due to the visual node system.

The major downside to C++ is the difficulty of implementation. In comparison to Blueprints, C++ is hard and more error-prone, where a simple mistake can take a long time to track down and solve and is more likely to crash the engine if an error occurs. This can add up over time when features are needing to be added, causing delays in a games development cycle. These crashes that are easy with C++ are hard with Blueprints, making Blueprints a much safer development environment.

### Unreal Engine Blueprints and C++

The main purpose of Blueprints is its simplicity, which gives the ability to create gameplay features quickly and easily. The Blueprints system built into the Unreal Engine is perfect for a game designer prototyping and iterating over gameplay ideas. By building elements of the game quickly in Blueprints, concepts can be rapidly iterated over to understand if they will improve the overall quality of the game. Blueprints are also perfect for elements of the game that are not performance-critical.

The Blueprints system then allows software engineers to build complex systems in C++ and for these to be exposed to Blueprints for easy modification. This allows game designers not to need to understand how to program using C++. By using this approach, game designers can iterate over ideas fast, while software engineers can simultaneously spend time on getting the most out of performance-critical elements of the new features.

There are many elements of making a game that does not need to be performant, and adding such features is drastically simplified through the use of the Blueprints system and engine interfaces. This is the main reason for both typically being used together in tandem. The

following examples help to demonstrate why this approach is useful.

### Setting the Default Pawn Class

One such example of this is setting the default pawn class (the character the user controls). Using the Unreal Engine, it is as easy as picking a pawn from a drop-down list (see *figure 2.1*). This compares to the implementation in C++, which requires six lines of code to add the same simple feature (see *figure 2.1*). In this code example, the location of the pawn class is hard-coded. If the pawn were ever to move its folder or be renamed, then the default pawn would no longer load. By using the blueprint dropdown menu, the Unreal Engine knows to track the location of the asset, and then if it gets refactored to another location, the engine will deal with this and tell the handler the location of the asset. This is a much better solution than hardcoding assets in C++.

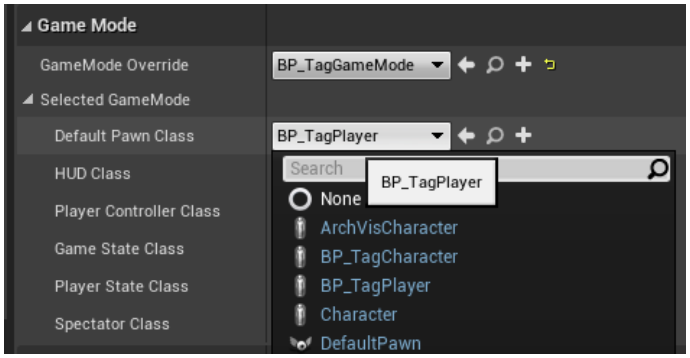



Figure 2.1: The Unreal Engine drop-down picker

```
1        AGameOfTagUnrealCPPGameMode::AGameOfTagUnrealCPPGameMode() {
2            // set default pawn class to our Blueprinted character
3            static ConstructorHelpers::FClassFinder<APawn> PlayerPawnBPClass(
4                TEXT("/Game/ThirdPersonCPP/Blueprints/BP_TagPlayer"));
5            if (PlayerPawnBPClass.Class != nullptr) {
6                DefaultPawnClass = PlayerPawnBPClass.Class;
7            }
8        }
```

Listing 2.1: Implementing the Default Pawn Class in C++

### Visual Complexity of Blueprints

One of the major drawbacks to Blueprints is the complexity that can arise from the number of nodes needed to represent parts of the project. *Figure 2.5* and *Figure 2.3* show the blueprint implementation for the `FindTargetToTag` function, which is a part of the AI implementation and is used to find another actor in the world that can be tagged. This actor cannot be itself (the currently tagged actor), and it also cannot be the previously tagged actor.

As blueprints are visual, the ability to view what is happening in a blueprint function can quickly become difficult due to an overwhelming sea of nodes. For simple tasks, this is not a problem, but as complexity arises, certain elements that can be represented in code easily can end up taking up lots of space on a Blueprint graph.

*Figure 2.3* especially shows the visual complexity that can arise from using blueprints. This figure uses 8 nodes to do a quick check to see if another actor is a possible target. This very same snippet can be represented in just *three* lines of code (see *Listing 2.2, lines 45 - 47*). Although the function call in *Figure 2.5* is simple (a singular node to call the function), the implementation itself uses up a considerable amount of space for something represented simply in code.

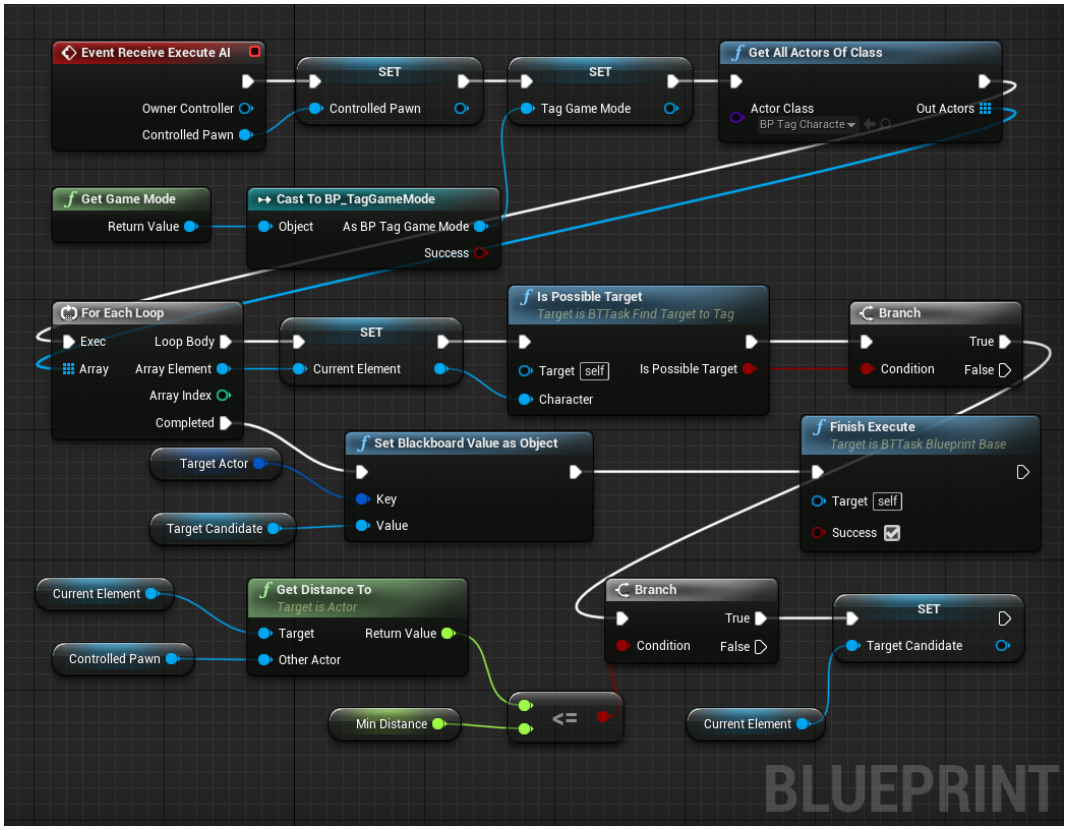



Figure 2.2: Blueprint nodes for the FindTargetToTag implementation

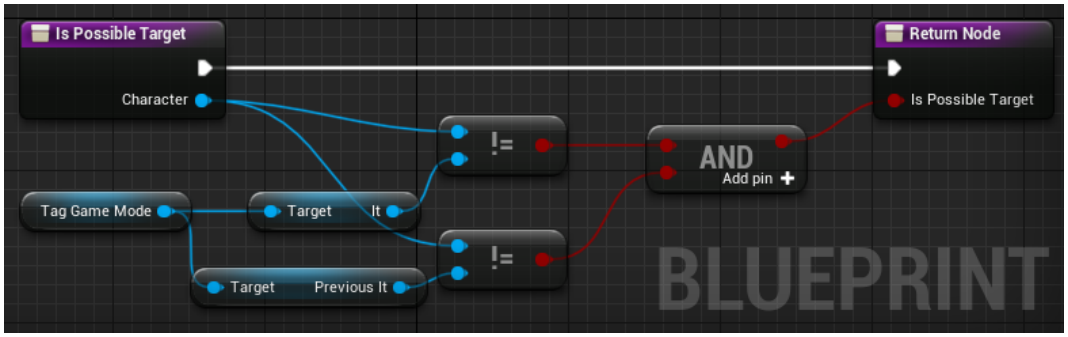



Figure 2.3: IsPossibleTarget function

Due to the visual complications of Blueprints, C++ can provide a more straightforward way to overview an entire class and its implementation all in one place, as many lines of code can be fit into more space on a screen than blueprint nodes.

```cpp
/**
* @brief Iterates through an array of all actors in the game, checks if
* they're a possible target and then sets the "TargetActor" blackboard to
* the found actor
* @return EBTNodeResult: Code if task executes successfully or otherwise
*/
EBTNodeResult::Type UBTTask_FindTargetToTag::ExecuteTask(
    UBehaviorTreeComponent& OwnerComp, uint8* NodeMemory
) {
    if (!(OwnerComp.GetAIOwner() && OwnerComp.GetAIOwner()->GetPawn())) {
        return EBTNodeResult::Failed;
    }
    // Get the AI pawn (self)
    APawn* ControlledPawn = OwnerComp.GetAIOwner()->GetPawn();
    GameMode = Cast<AGameOfTagUnrealCPPGameMode>(
        UGameplayStatics::GetGameMode(GetWorld()));

    // Get all actors in the world
    TArray<AActor*> FoundActors;
    UGameplayStatics::GetAllActorsOfClass(
        GetWorld(), AATagCharacter::StaticClass(), FoundActors);

    const float MinDistance = 10000000.0f;
    AActor* TagCandidate = nullptr;

    for (AActor* Actor : FoundActors) {
        if (IsPossibleTarget(Actor)) {
            if (Actor->GetDistanceTo(ControlledPawn) <= MinDistance) {
                TagCandidate = Actor;
            }
        }
    }
    // Set blackboard value
    OwnerComp.GetBlackboardComponent()->SetValueAsObject(
        FName("TargetActor"), TagCandidate);
    return EBTNodeResult::Succeeded;
}

/**
* @brief Checks if the found actor is not "It" and not the "PreviousIt". If
* true then they are a possible target
* @param Character: The actor to compare
* @return bool: true if possible target
*/
bool UBTTask_FindTargetToTag::IsPossibleTarget(AActor* Character) const {
    return Character != GameMode->It && Character != GameMode->PreviousIt;
}
```

Listing 2.2: C++ class for the FindTargetToTag implementation

### Version Control

Another negative to the Blueprint system is the fact that Blueprints are binary data. A key part of the version control process is the code review step (see *Chapter 3.3*). While Blueprints can be 'diffed'[20], this process can only be done from within the Unreal Engine editor. To code review, a developer has to open the editor, find the necessary blueprint files, and then open the blueprint difference viewer to be able to understand changes made to a set of Blueprints. This process adds time to the reviewal process of changes. This directly contrasts with implementations in C++. As C++ is text-based, it is simple and easy to see what changes have been made. This process of review is already built into many different version control systems

### Behaviour Trees

When it comes to implementing AI in Unreal Engine, there are theoretically three possible ways. Firstly a developer could create a custom Behaviour Tree implementation. There is no reason that a developer needs to use the tools provided by the engine. The second way is to look through the Unreal Engine source code (that can be found on GitHub), find the internal implementation of behaviour trees (create an instance of `UBehaviorTree`, populate with nodes and a custom blackboard) and then calling `AIController::RunBehaviorTree` to start the Behaviour Tree. This method is poorly documented, as although possible, it is not the desired way to create behaviour trees in the Unreal Engine. This leads to the third option: using the graph implementation given.

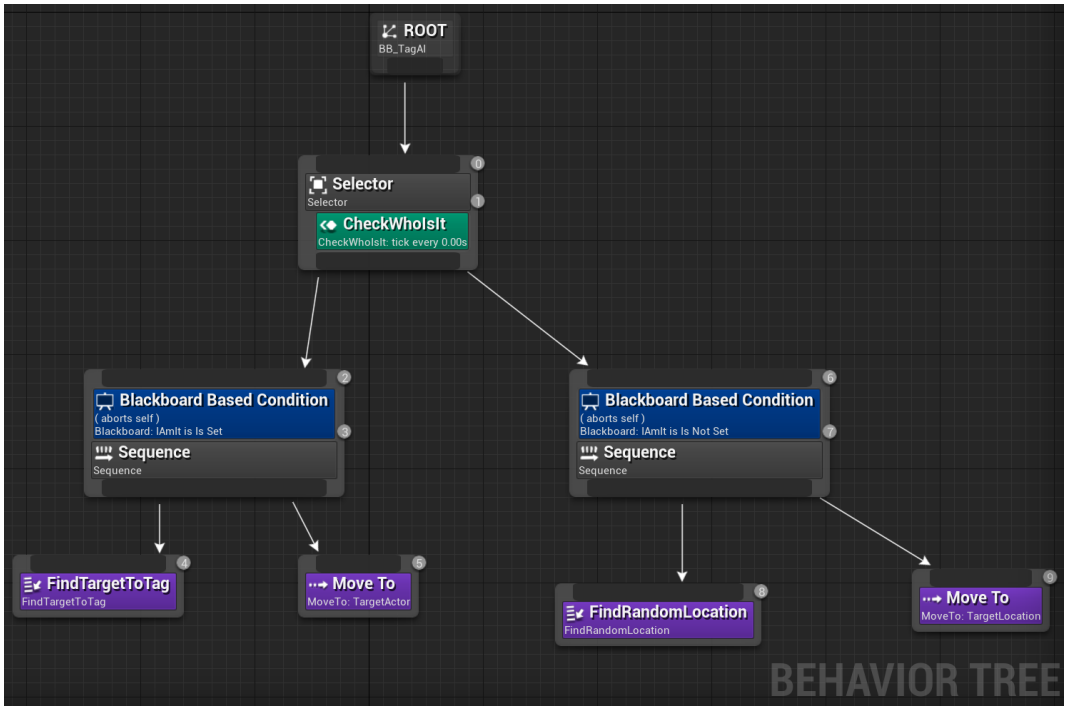



Figure 2.4: Behaviour Tree Graph

The intended way to create Behaviour Trees in the Unreal Engine is using the Behaviour Tree graph *(see Figure 2.4)*. This is a visual behaviour tree window that allows a developer to easily create and customise a Behaviour Tree by dragging and dropping nodes into the graph. These nodes can either be created using Blueprints or can be implemented in C++ and then used in the graph window like a normal Blueprint node.

The advantages of using the Behaviour Tree graph implementation are:

- it is the intended way to create Behaviour Trees; therefore, the documentation and

support surrounding their use are much more plentiful than other alternatives. This makes making them easier to understand and implement

- it is easy to drag and drop nodes around, and a developer can easily view the order in which each node should be considered. It also allows other developers to quickly understand how the tree is built and meant to behave
- during play mode, the editor shows the control flow of the Behaviour Tree, with the current node being processed being highlighted in yellow. This allows for much simpler debugging as a developer can see if the current state of the tree is the same as the desired state of the node

Overall, the Unreal Engine has been designed that both Blueprints and C++ should be used together to maximise performance and speed of development. Although everything within the engine can be done using C++, many parts of a game project can be made easier through the use of Blueprints. There are perks to Blueprints and perks to C++ - the amount of each that is used within a project will highly depend on the requirements and demands of a project.

#### Best Approach to Unreal Game Development

The Converting Blueprints to C++ course[21] from the Unreal Engine Online Learning hub discusses one efficient way of implementing a game using both Blueprints and C++, which was adapted to create the C++ version of the Game of Tag. The course discusses the idea of creating all components, initially, in a game using Blueprints. This allows for easy iterative development to find a fun concept that gives users a good gameplay experience. From here, a developer can use profiling tools and testing to see where the bottlenecks of a system are. This allows for crucial aspects of a project to be moved into C++ code, with less critical components being left in Blueprints.

As Blueprints can call C++ code, the course discusses moving a section at a time into C++. For example, in the Game of Tag, the AI behaviour tree has different nodes, such as the 'CheckWhoIsIt' node. A C++ class that extends from `UBTService` can be created, and then the Blueprint logic can be moved into C++. This exact node can then be put into the place where the original Blueprint node was called. Despite 'FindTargetToTag' and 'FindRandomLocation' still being in Blueprints, a developer can quickly test if the existing functionality works with the new C++ code while using both C++ and Blueprint Behaviour Tree Tasks simultaneously.

For the Game of Tag, this approach was taken one set further. As the entirety of the game was built in C++ (aside from the Behaviour Tree implementation), profiling tools were not used and instead, everything was taken from Blueprints and built into C++.

Blueprints are also relatively simple to convert to C++ as they are designed in such a way that they preserve object-oriented functionality, and the node objects are named as such that they are very similar, if not the same, as the original C++ classes. This makes finding the documentation or C++ source code to work out how to implement the C++ version simple.

### 2.2.3 Unity

Unity is the *world's most popular* development platform for creating 2D and 3D multiplatform games and interactive experiences[22] which was first introduced in 2005. As of 2018, Unity now supports more than 25 platforms, such as PC, Xbox, iOS, and more[23]. At launch, Unity aimed to "democratize" game development by making it easily accessible to more game developers[24], and ever since, it has been a popular engine, especially for indie developers and mobile game development.

Unity utilities C# as its programming language of choice and currently offers no other alternatives to creating games within its engine. Previously Unity offered a JavaScript-like scripting language named UnityScript, but this was deprecated in 2017 due to lack of adoption from developers. The only other option for development is to use other .NET languages that can be compiled into a compatible DLL. The compiled .dll file can be added to a Project, and then attached the classes it contains to a GameObject[25].

The following section will discuss how the use of Unity with C# compares to that of building a game in Unreal using a combination of C++ and Blueprints.

#### Unity C# and Backend

Although the Unity engine itself is built using both C and C++ internally, the engine has a C# wrapper that is used to interact with the game engine. C# is a general-purpose multi-paradigm programming language encompassing static typing, strong typing, lexically scoped, imperative, declarative, functional, generic, object-oriented (class-based), and component-oriented programming disciplines[26]. C# first appeared in 2000 and was developed by Microsoft as a rival to Java as Microsoft did not want to make changes to Java and instead chose to create their own language.

When Unity first released, Unity used Mono as their scripting backend to support cross-platform development, which is a free and open-source .NET Framework that is owned by Xamarin, a subsidiary of Microsoft. The reason for this is because C# source code gets compiled into intermediate code, which relies on the Common Language Runtime (CLR) to run on a target platform. Initially, CLR was only targeted to the Windows platform. This is where Mono comes in. The developers of Mono implemented the .NET CLR in platforms other than Windows, which allowed for Unity to be cross-platform and run on all the target hardware of the engine.

Although Mono is still supported, there is an alternative now to Mono named IL2CPP (Intermediate Language To C++). This is a Unity-developed scripting backend that converts the Intermediate Language code from scripts and assemblies into C++ before creating a native binary file. Compile times with IL2CPP are longer than that of Mono while also having to create platform-specific executables, but the use of IL2CPP helps to increase the performance, security, and platform compatibility of Unity projects.

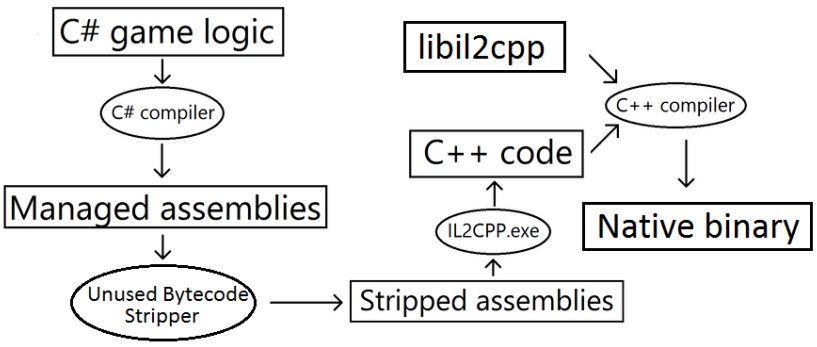



Figure 2.5: A diagram of the automatic steps taken when building a project using IL2CPP[27]

### 2.2.4 Unity vs Unreal

The following section gives an overview of the differences between Unity and the Unreal Engine and the merits and demerits of each.

### Unity C# verses Unreal C++

Although other languages such a C# and Java have been used with much success in the game development space, C++ is one of the most prominent languages used within 3D game development [3] due to its high performance, memory management, and control over computing resources [28], allowing for the ever-increasing gameplay demands of users to be explored.

C# is widely regarded as an easier language to learn than C++. Many features that have to be carefully managed in C++ (such as garbage collection) are managed automatically by C#, making C# code less error-prone while also being quicker to write. C++ is a more performant language than C#, but with the increasing capabilities of modern hardware, the different between extremely fast (C++) and very fast (C#) consequently becomes more of a moot point.

Despite the traditional need for memory management in C++, the Unreal Engine does a lot of the hard work for a developer. The Unreal Engine has built-in garbage collection, which aside from a developer having to ensure that a created object is exposed to the garbage collector using the Unreal Engine's reflection system, there is little effort required to ensure that a created object is removed when no longer needed. By the nature of the engine, if a developer did want full control over how garbage collection works, they can simple disable it from within the engine.

C# developers do not have to consider memory management if desired, as C# employs automatic memory management. This contrasts with C++, where memory management through pointers and references is required, adding an extra layer of complexity when considering how to write code. Although the Unreal Engine's version of C++ is easier than standard C++, it is still harder to use than C#, making it much easier to pick up programming in Unity than the Unreal Engine.

### Ease of Use and Documentation

Unity has a much larger userbase than that of the Unreal Engine and subsequently has a lot more tutorials on how to grasp different parts of the game engine. The Unity Documentation is also rich with information, making the learning experience relatively simple for someone wishing to dip their toes into game development.

This contrasts with that of the Unreal Engine. The documentation for the Unreal Engine is good when it comes to developing games in Blueprints, and parts of the C++ Unreal Engine API that are completed are done well, but many features present in the game engine in C++ are missing. As the entirety of the engine is available for viewing, is it almost expected that a developer needs to look through source code to understand how to use some of the in-built functions. This makes the learning experience for the Unreal Engine a lot more cumbersome than that of Unity. In the past, a lot of the Unreal Engine documentation was also found on the Unreal Engine forum, which has since in large parts been discontinued, again making it more difficult to find relevant discussion on certain topics.

### Source Code

Unity does not offer users the ability to view the full engine source code by default. To do so, a user must buy a Unity Source Code Commercial license. This license is only sold on a per-case and per-title basis via special arrangements made by the Unity business development team. However, the Unity C# reference is available for read-only for learning purposes, but a developer is not allowed to modify this code[29].

This contrasts with the Unreal Engine. Epic Games freely give out source code access via their GitHub to any developer who agrees to their EULA. Developers are allowed to build the engine from its source freely and are encouraged to modify and change the engine to fit their own needs. Where improvements are made to the engine, developers are also actively encouraged to create pull requests to the GitHub repository to add their changes to the Unreal Engine development branch for any other developer to use.

Unity believes that the vast majority of their clients do not need access to the engine source code - which is likely to be true. The only people who would need access to the engine source code would be hobbyists who like to tinker, as well as studios that are trying to push the boundaries of what is possible to do with an engine (such as Riot Games with their game Valorant - see *section 2.1*). This limitation can be a deciding factor for AAA studios who want access to the engine source without having to purchase a commercial license to do so.

**Non-Programmers**

For non-programmers, such as artists and game designers, the Blueprint system of the Unreal Engine makes it simple when it comes to modifying assets from within the engine. The material editor in the Unreal Engine uses the Blueprint system (see *section 2.2.2*) to change the look and feel of a material, making it super simple to represent how the material should behave visually.

The Unreal Engine, while sometimes daunting, also has specialised windows and dedicated workflows for different parts of the engine. An artist can completely isolate themselves from parts of the engine that are not relevant to them and purely focus on windows that are.

**Built-in Tools**

The Unreal Engine has a lot more built-in, out-of-the-box functionality than Unity. Many features that the Unreal Engine has enabled by default (such as Behaviour Trees) are not present by default in Unity. A lot of this can be primarily made up for by the Unity Asset Store. As Unity has a larger userbase than that of the Unreal Engine, the asset store has a wide range of features and tools for a developer to use. Some of these are free, either developed by Unity or themselves or by other developers on the platform. This makes adding missing functionality to the editor easy. However, not all desired functionality can be found for free. For example, the highest-rated behaviour tree implementation that is built into Unity is the 'Behavior Designer - Behavior Trees for Everyone' tool by Opsive that costs *€75.93* to purchase. This feature that was built into the Unreal Engine has a cost in Unity. This also uses an 'Extension Asset' license type, which means that one license is required per individual user. For a small team of one developer, this cost may not matter, nor for the large budgets of AAA studios, but for teams in-between, the cost of missing functionality could add up.

**Visuals**

Visually, out-of-the-box, the Unreal Engine has much higher-fidelity graphics and overall better customisation option for tweaking the visual settings of a game. The overall rendering, postprocessing, shaders, and visual effect pipelines are much easier to use in the Unreal Engine than they are Unity. Unity can still produce impressive visuals, but this usually takes more time and effort than that of the Unreal Engine, which is often associated with its impressive graphics.

The Unreal Engine has capable processing power when it comes to lighting, but these are not typically required when creating a game for a lower-powered device such as a mobile phone. Unity is a powerful tool for 2D and mobile development due to its simplicity and ease of use, especially when the full power of the Unreal Engine is not required.

**Default Character Controller**

The default character controller in the Unreal Engine is much simpler to use than in Unity. In Unity, there is a character controller included in the standard assets package (this must be installed from the Unity package manager) but has poor camera control, and the standard assets package is also now deprecated (despite still being downloadable). The expected way to create a character controller in Unity is to develop it from scratch. This process is relatively easy and gives a developer full flexibility over how the controller works, but simply wishing to create a character and run around the newly created world is not possible. This process adds extra complexity to starting a game development project.

In comparison, the Unreal Engine has a default mannequin that can be initially added to the game world through one of the default game templates. For example, the 'Third Person Template' that was used to create the Game of Tag worked out of the box and can be added with either C++ controls or Blueprint controls from the get-go (see *section 2.2*). This mannequin makes it much easier for human-like players to be added to the Unreal Engine than in Unity.

The Unreal Engine does not include any other controllable pawn template other than the default Unreal Engine mannequin; therefore, if the controller that is added does not represent that of a humanoid, both Unity and the Unreal Engine match in terms of pawn creation due to having to make them from the ground up.

### 2.2.5 Closing Remarks

Both Unity and the Unreal Engine are powerful engines that can create top-selling games for all platforms. Despite this, both engines have their flaws. The Unreal Engine is a game engine that was designed for AAA studios and has since been opened up more to allow for small indie developers to jump aboard and make games. Unity is an engine that was developed to 'democratize' game development for indie developers but has since tried to open up more to become more of a AAA game engine in its own right.

The overall consensus seems to be that despite Unity and the Unreal Engine being able to work for everything, Unity is much better designed for 2D games, as well as mobile games. It is simple enough to use and get into with well developed 2D tools and ports well over to these platforms. Unity is good for making 3D games, but the higher-fidelity graphics and dedicated workflows in the Unreal Engine typically make it a more appealing option to larger teams looking to push visual boundaries with built-in tools designed to do the job. This approach can also be seen by Riot Games with their mobile game Wild Rift being created in Unity, while their 3D First Person Shooter being developed in the Unreal Engine (see *section 2.1*).

# Chapter 3: **Software Engineering Methodologies, Tooling and Processes**

Software engineering is the systematic application of engineering approaches to the development of software[30]. Within this discipline, there are fundamental approaches and tools used to improve the process of developing, maintaining, and testing software. There are many of these methods that the games industry adheres to and others that are not.

Traditionally, many games in the past were created using the waterfall software methodology to create a boxed product. The development phase was followed by a bug-fixing phase, where a few months at the end of a project were dedicated to getting close to zero bugs for its release. Once the game was released, the development team then moved onto a new game. Nevertheless, many games are now moving away from this approach since games are more commonly becoming software as a service, where development teams continue to work on a game long after the game is released, continuing to add new features and gameplay to keep players engaged and spending for longer. This means that the software principles needed to produce games have changed over the years.

This chapter describes some of the critical software engineering methodologies, tools and processes that are used to aid these differences and how they can be applied and used within the AAA game development space. It also discusses some of the most advocated practices in the software engineering world that the game industry seldom follows.

## 3.1 Software Testing

Software Testing is one of the most important stages within the Software Development Lifecycle. The Art of Software Testing says[31]: "At the time this book was first published in 1979, it was a well-known rule of thumb that in a typical programming project approximately 50 per cent of the elapsed time and more than 50 per cent of the total cost were expended in testing the program or system being developed. Today, a third of a century and two book updates later, the same holds true."

Software testing is paramount to ensuring that a program is usable. Without it, programs would never work as intended, resulting in all the time and money spent on creating the software going to waste. Ensuring that a program is usable under as many conditions as possible is a mandatory aspect of software and game development.

All games are tested. What differs between different studios is the approaches used to test a game. This section contains information on methodologies and approaches typically used to tackle Software Testing in the games industry, with mention to approaches historically used within the AAA games industry and the reasons why some companies are moving away from these.

### 3.1.1 Manual Testing

Manual testing, also known as Alpha / Beta testing, is the Customer Validation methodologies (Acceptance Testing types) that help in building confidence to launch the product, and thereby results in the success of the product in the market[32]. They rely on real users to test the

product and for these users to then provide feedback of issues that have been found and the steps needed to reproduce said issues.

#### Testing Phases

Typically manual testing goes through five distinct phases[33].


Figure 3.1: The five stages of manual testing [33]

The **Pre-Alpha** phase is where the game being written is still in development. This phase will include typical testing by the developers who create the code to make sure that it works as intended, and they will also write automated tests (see *section 3.1.3*) if required.

The **Alpha Testing** phase involves both white-box and black-box testing and is internally tested for bugs/fixes. In a AAA studio, this process is typically carried out by dedicated in-house teams whose sole purpose is to play video games to check that they work and discover and record problems in the game. This role has been historically common in the games industry due to the nature of the games industry being considered by many a 'passion' industry. The idea of playing games as a career seems fun; therefore, AAA companies could pay cheap wages to those wishing to do so to test their games. As the alpha testing phase is carried out by game testing professionals, a significant amount of bugs will be found during this stage.

The **Beta Testing** phase is the first stage where players get to use the new updates. The software is considered relatively stable and is released to a small player base. It is common for the beta changes to be pushed to a 'test server', which gives players a way to try out new unreleased features while also being used as a way to test for bugs on a higher set of hardware configurations than that of the alpha testing phase. The significant advantage of beta testing is the fact that a developer can quickly get direct feedback from a player base as they have the changes in their own hands and can experience the changes in their own time.

The **Release Candidate** phase is typically where feedback from the beta stage has been implemented, and the software is ready for release. No huge changes in functionality will happen now, but these changes are still available for the public to use to try and catch any last bugs that are still present in the game.

The **Release** phase is where the game releases to the public in a patch. The majority of bugs in the game have been removed, and, likely, any bugs present will only appear in the most niche of cases.

### 3.1.2 Test-Driven Development

Test-Driven Development (TDD) has become one of the most common agile practices that are advocated and accepted by the software engineering industry[34]. TDD advocates for tests driving the fundamental development of a program. This practice is done by:

1. Writing a unit test that describes a small feature of the program
2. Running the test, which will fail as there is no feature implemented

3. Writing enough code so that the test passes, then refactoring the code
4. Repeating this process to create a codebase with 100% code coverage with unit tests for every aspect of the program

TDD is great for the software engineering profession. By creating granular tests and slowly building up a codebase, the developer constantly has to consider how the production code is being written. This helps to develop tidier code as well as giving the developer a better focus when developing new features - they only need to focus on one part of the problem at a time.

The main issue when it comes to TDD is the notion that "tests drive development". In Software, requirements are usually pretty well-defined and understood before coding begins. Components of a program develop throughout its lifecycle, but usually, only small portions of a program's specification are changed. This contrasts with game development. The issue stems from the idea that a game should be 'fun'. A new feature is discussed, designed, and then implemented as planned. Time is spent writing the tests for this feature to ensure that its implementation is correct. This feature is then playtested, and it is likely that it will be decided that the feature needs to be changed to improve it. It could also end up that the feature is not enjoyable to use, and as such, gets removed during the development cycle.

This definition of fun is one of the main arguments as to why TDD is rarely used in the games industry. How does someone define fun? What is enjoyable to one person can be very different from another, and as such, compromises are made to ensure that a game is as enjoyable to as many people as possible in a target audience. That being said, it is definitely paramount to use a form of TDD to create good software. This process of prototyping an idea in the Unreal Engine is typically done using blueprints, where ideas can be changed rapidly without too much hassle. In this prototyping stage, nearly all of the code wrote will be thrown away at some point. It is purely here to develop a specification where the game designers can agree on what they believe to be a fun product. Once the prototyping process is completed, the prototype becomes the 'specification' for the project. At this point, a standard software lifecycle can occur, where software engineers build a product from the prototype using TDD, building up their automated tests to ensure that good software is developed.

TDD helps to create a large set of unit tests that, in theory, give 100% code coverage in a program. These tests can then be automatically run while creating features to help ensure that new code does not break existing functionality. Although the methodology of creating a test first and writing the production code of the test afterwards is not commonplace in the games industry, the idea of having a form of automation tests is starting to appear in the game development space (see *section 3.1.3*).

### 3.1.3 Automated Testing

Automated testing is the use of software separate from the software being tested to control the execution of tests, and the comparison of actual outcomes with predicted outcomes [35]. Automated tests allow for a developer to check their new changes against every pre-existing test for a system quickly. Instead of waiting for a Quality Assurance (QA) team to manually check new feature for any bugs, a developer can know, potentially within minutes, if their new changes have broken any major features in a working branch.

Many different parts of the development process can be automatically tested, but it requires developers to write these tests in the first place and also requires a good software development approach to ensure that the tests are useful and meaningful.

One of the main disadvantages of automated tests is the time spent writing the tests in the first

place. As every new change to the production environment needs to have corresponding tests, there is an increased cost when making changes. Instead of quickly adding new gameplay features, a developer will likely need to spend more time thinking about code architecture (tests can be hard to write otherwise), but this also means that the code being pushed to the production branch is likely to be of a higher quality. The time spent doing this can also be offset by the fact that the bug-hunting process at the end of a games development cycle becomes almost non-existent, and it is easier to track down any bugs that do appear in development.

### Automated Testing at Rare

Rare Ltd, an Xbox Game Studio, known for making the games Banjo Kazooie: Nuts and Bolts, Sea of Thieves, and Battletoads (to name a few), decided to move to a CI/CD (see *3.2*) approach when building their newest game as a software as a service, Sea of Thieves, using the Unreal Engine. To do this, they took the mainstream software engineering approach of TDD and adapted it to how they believed it could better suit the games industry.

Rare's approach to game development was first to develop any ideas that they had in Blueprints on a separate prototype branch (which had no automated testing). A quote from a software engineer at Rare, Robert Masella, says, "Automated testing is more of a hindrance than a help if you're still iterating on your game to find if its fun." Rather than writing tests to cover the entirety of their blueprint prototypes (much of which will be later converted into C++), they used this prototype branch to quickly develop features, iterate over ideas, and therefore made it easy to find out what features were fun and which were not. This feature could then be created in the production branch with corresponding tests to match.

When a change was made on the production branch, instead of following a TDD approach to creating tests, Rare decided that the workflow of a developer was to usually make a change to the code, see if the change worked (from a gameplay perspective), and then to create tests after to pin down the behaviour of the feature afterwards. This allowed developers to iterate and improve upon ideas that were already in the production branch without having to recreate them in the prototype branch.

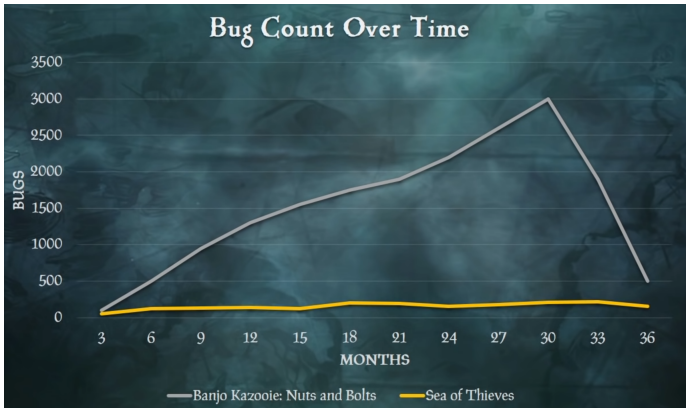



Figure 3.2: Bugs over time in Sea of Thieves and Banjo Kazooie [36]

*Figure 3.3* shows a compelling reason as to why automated tests should be commonly used within the games industry. Rare compared the estimated amount of bugs that were present in Banjo Kazooie to the number of estimated bugs in Sea of Thieves at any given time. Automated tests combined with a CI/CD pipeline meant that at the release of Sea of Thieves, there were fewer bugs present, as well as no longer having the traditional crunch period at the end of a game development lifecycle to ensure that a game is bug-free. Automated tests ensure that any new features added after the release of a game can be tested against the current suite of tests, which can be invaluable, especially if the game continues to receive new features and

functionality.

#### Summary

As a whole, automated testing does not necessarily replace manual testing (see *Section 3.1.1*). Even with 100% code coverage and near-perfect tests, games can be unpredictable and do not always behave the way that a developer predicts. What automated tests do, though, is ensure that large portions of a game are constantly being checked for irregularities. This helps to speed up the development feedback loop when adding new changes, as simple bugs are a lot easier to discover, and it also ensures that simple bugs do not make it past the development stage. This can free up time for a quality assurance team to focus on bugs that can be more game-breaking or harder to understand the reason why they exist in the first place.

An example of how automated testing can be used is shown in *Chapter 5.3*

## 3.2 Continuous Integration, Continuous Delivery

CI/CD is the combined practice of both Continuous Integration and Continuous Delivery. CI/CD bridges the gaps between development and operation activities and teams by enforcing automation in building, testing and deployment of applications[37].

This section contains information on what CI/CD is, how CI/CD can be applied to the games development process, as well as the benefits of using CI/CD in a game development pipeline.

### 3.2.1 Continuous Integration

Continuous integration (CI) is the practice of automating the integration of code changes from multiple contributors into a single software project[38]. It allows for developers to frequently merge code changes into a version control system *see Chapter 3.3* where builds and tests can then be run. Automated tooling is used to assure that the new code being added is 'correct' before it is integrated.

The most important part of the automation process is to check that the new code being added does not break any existing functionality. This is done through a set of automated tests *see Section 3.1*, which checks the code against predefined tests. If any of the tests fail, then the code will be automatically rejected from merging into the central repository, and the developer who pushed the changes will be notified. Despite this, the automation process does not only have to cover tests, and can also be used to, for example, ensure code style or a minimum requirement on code coverage.

CI also helps reduce the code review process. If a pull request is flagged as broken by the CI server (e.g. the code style is incorrect), then the developer who made the change will be notified automatically by the CI server. This means that the other developer doing the code review never has to check the pull request, as the CI server will have automatically closed it.

### 3.2.2 Continuous Delivery

Continuous Delivery (CD) is an extension of CI. After changes have been validated in the CI stage, the code changes are then automatically deployed to the testing and production environment. By having an environment that is constantly in a deployment-ready state, the application can theoretically be deployed at any time.

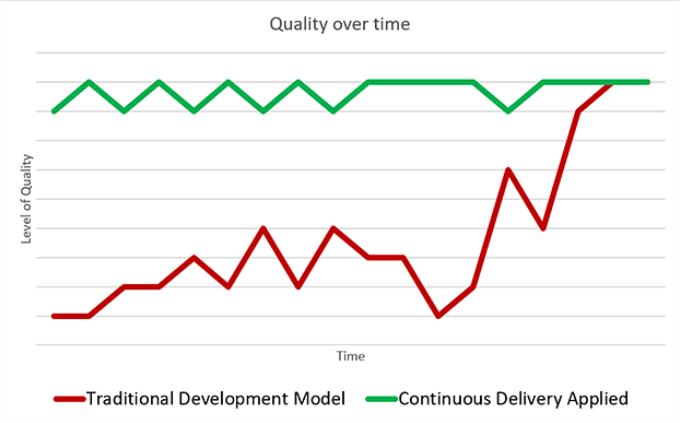



Figure 3.3: Quality of a game over time[39]

CD ensures that new features can be delivered to players more regularly than through traditional means, where a whole game is released in one big go, and then the entire development team moves onto a new project. Developers have to constantly consider the fact that any features that they push to the project could at any time be delivered to players, and therefore it ensures that only features that are considered to be working are pushed to the project. As developers are also encouraged to push often (at least once a day), they will ensure that they will work on only a minimal amount of a feature to get it working, making it easier to modularise a codebase and helps to keep code from being tightly coupled.

#### Continuous Deployment

The CD in CI/CD can also mean Continuous Deployment, with takes Continuous Delivery one step further, and instead of deploying all code changes to the testing and production environment, the changes are directly pushed to live for all customers to use.

Continuous Deployment is not typically used within the AAA game space. Continuous Deployment relies on being able to constantly send updates to users so that any change made by developers is in the hands of users instantly. This approach is most suited to applications such as a website, where every time a user loads a new page, they can get the most up-to-date information. This contrasts with the games industry. Many big games have dedicated “patch days” where lots of new features are released at once, and users can be excited to see the latest changes. Part of the reason for this is due to the size of updates. Every time they logged onto a game, if a user had to download a few hundred megabytes worth of data to play the game, people on slow internet speeds could end up having to wait a decent amount of time. This could deter people from wanting to start the game when they know that there is a patch and they only want to play.

Games are also notoriously hard to test. There are too many different factors that go into making games that no matter how good a testing pipeline is, bugs will still be able to make it through each stage. By not having changes made to the production environment going instantly to live, it gives testers one last safety net to ensure that the patch to be published does not include any substantial game-breaking bugs.

## 3.3 Version Control

Version Control Systems (VCS) are used to record changes to a file or set of files over time so that you can recall a specific version of the file(s) later[40]. This allows for a project to have detailed backups, as well as the possibility for different teams of people to work on independent versions of the same project.

This section intends to compare Perforce and Git and the reasons that Perforce is typically used within the games industry. Other VCS are available, such as Subversion and Mercurial, but this section will be limited to Perforce and Git due to their overall popularity within the game development space.

### 3.3.1 Why Version Control?

In software, codebases are constantly modified. New features are added, redundant aspects of a project are removed, and existing functionality is replaced by new and improved approaches to a problem; therefore, VCS software is needed to help with the maintenance and development of projects. Without it, a company like Google that maintains billions of lines of code[41] would not be able to manage their development workflow.

VCS has become an imperative feature of the software world. It gives projects a back-up strategy and a simple approach to managing projects[42]. The fact that it allows for detailed logs to be stored, multiple people to easily share access to the same project, for people to collaborate effortlessly, as well as providing a source of accountability for any changes made to a project. VCS continually proves to be an efficient development solution.

### 3.3.2 Perforce

Perforce's *Helix Core* is a 'Centralised VCS' (CVCS) that stores everything in one central repository (rather than on an individual users workstation). This means that all users commit changes to a central server, which allows for a single source of truth across an entire enterprise[43]. The single central repository can store code, large binary files, and digital assets all in one place.

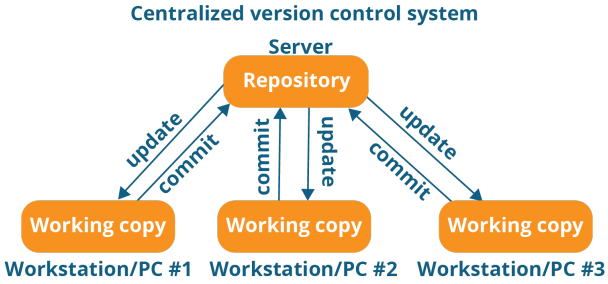



Figure 3.4: Centralisd VCS diagram [44]

Perforce also has substantial fine-grained access control[45]. This allows, for example, an art team to have full access to all art assets, but read-only access, or even no access, to source code within the same mono repository.

#### Advantages

- Can store large binary files in the same repository as the source code as the user can opt-in to which files they clone to their machine
- By storing everything in one place, developers will always have the latest version of what they need
- Do not need to install the entire repository to the local machine, which can save time and space if using a large mono repository

### 3.3.3 Git

Git is a 'Distributed VCS' (DVCS), where developers can download source code, along with the full version control history of the repository to their machine, which allows for changes to be made locally. Although it sounds wasteful to store full copies on every machine, many repositories are small in size and only contain plain text files.

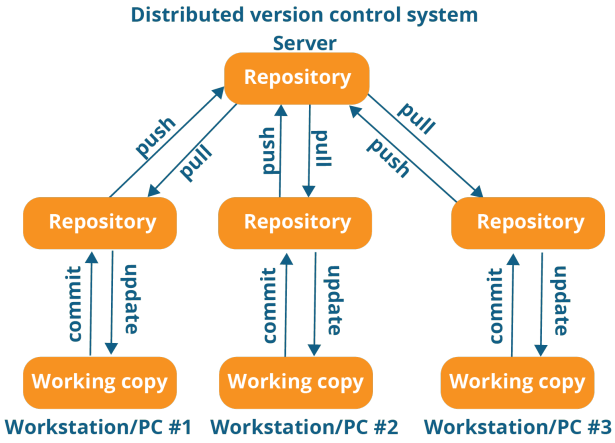



Figure 3.5: Distributed VCS diagram [44]

By ensuring that that each developer has a copy of the entire repo on their machine, local changes can be made fast. These changes can then be pushed to the master repository all at once. Branching and merging in a DVCS is quick because a developer can just start working on the local branch.

Git has become almost the defacto standard for VCS in the software industry, with 88.2% of professional respondents to Stack Overflows Annual Developer Survey[46] in 2018 saying they use Git. This is primarily due to sites such as GitHub and GitLab that offer Git repository's for free to individual users, Git's blazing fast speed for commits and changes, as well as having the ability to make changes offline.

#### Advantages

- Performing any task apart from pull and pull is fast as all actions are made on the local machine
- As a copy of the entire repository is stored on the local machine; git changes can be made without an internet connection
- Typically cheaper than a centralised option as git is open-sourced and a centralised server to store the repository is not required

#### Git LFS

The major problem with Git for game development is the store of binary large objects (blobs). Games are not just text-based programs, and because of this, their repositories can be made up of hundreds of terabytes of assets such as images, music and 3D models. VCS typically works by storing the changes between files, so a program can easily be stored due to only needing to record the text differences between each version. Blobs cannot be stored this way. Git LFS is designed to help overcome this.

Git Large File Storage (LFS) is an open-source Git extension developed by Atlassian, GitHub, and a few other open-source contributors[4] that replaces large files such as audio samples, videos, datasets, and graphics with text pointers inside Git while storing the file contents on a remote server like GitHub.com[47]. When a developer clones a Git repository, Git LFS uses the pointer file as a map to find the large file on the remote server, which ensures only the files relevant to a project are downloaded, rather than all the files and backups of those files. Consequently, less data is downloaded, giving faster cloning and fetching, as well as ensuring that the central repository is at a more manageable size.

The most significant advantage of Git LFS is its seamless integration into Git. In a working copy, a developer will only see actual file content, which means that using Git LFS does not change the existing Git workflow[4]. If a developer is already using Git, this makes it nearly as simple as installing Git LFS, enabling Git LFS in the repository settings (if required - for example, Bitbucket enables Git LFS by default), and then using git as usual.

Git LFS 2.0 also introduced a feature named 'File Locking'. As blobs are hard to merge, it is advantageous to ensure that a blob cannot be modified by more than one user at the same time. If a file is marked as *'lockable'*, then Git LFS makes these files read-only on the file system automatically, preventing anyone from editing the file. If a user wishes to make changes to the file, then the user must first lock the file to edit it (which is a client-server command), preventing anyone else from locking the file for their use. Once they have finished editing the file, then the user can use the unlock command and push their changes as soon as possible to allow another user to edit the file.

#### Partial Clone

Partial Clone is a Git performance optimization that allows Git to function without having a complete copy of the repository[48]. During the clone and fetch operation, Git downloads the complete contents and history of a repository, including commits, trees, and blobs which can be hundreds of gigabytes in size. Partial Clone allows a user to avoid downloading unneeded objects in advance during the clone and fetch operations, reducing disk space required as well as the time needed to download files. This can be used to ensure that a user only downloads the content that they need. If the user is missing objects, these objects can then be 'demand fetched' when needed.

This feature can be combined with Git LFS. Git LFS will reduce the size of the blob files being downloaded, and then Partial Clone will ensure that only the content required is downloaded in the first place.

### 3.3.4 Perforce and Git in the Games Industry

So the question is: why do 19 of the top 20 games companies[3] choose to use Perforce? Git certainly is not unused in the games industry, where a survey by Perforce has 39% of game developers using Git as their VCS of choice. Even so, the top AAA games companies typically

all use Perforce, and this section will look into the reasons why its typically used and how Perforce compares to that of Git.

Git and Perforce both fundamentally do the same job. They are both version control software that is designed to allow for collaborative editing between developers, store back-ups of files, and enhance the development workflow. Their key differences, where game development is concerned, lie in the way that they deal with large binary files. Perforce is designed with scalability and keeping with a single source of truth, where an entire repository can be stored in one place. Many of the necessary features for dealing with binary files are also key components of the Perforce workflow. Git, on the other hand, natively does not support binary data well due to having to download entire copies of binary files as backups, bloating the individual working environment. This can be somewhat alleviated through the use of additional features to Git.

Git LFS (see 3.3.3) fixes a lot of the shortcomings of Git when it comes to versioning binary files. By storing pointers to the files that are needed and offering a locking mechanism on specific file types, developers can ensure that the codebase is kept small and that binary files can still be easily accessed. The *problem* with Git LFS is the fact that it requires a client-server command to lock files for editing. One of the main advantages of Git is the fact that it is meant to be distributed, with the ability to edit files offline. By having a single repository where binary files are stored, the distributed advantages of git are no longer present. Git LFS is also an additional feature that can be added to Git instead of being a native feature that is always present. This compares to Perforce that manages binary files efficiently by design.

One of the biggest advantages of Git is cost. Git is open-source and, as such, is free to use. Many websites (such as GitHub and GitLab) offer to host for free or provide additional storage for a low cost. Perforce does offer a free plan (for up to five users), but this would not provide enough for a AAA game studio. Instead, a studio must opt-in to a commercial license, which can cost upwards of $160 per user[49]. This cost can be a big factor in the decision between which VCS to use. For a large AAA studio, this cost is likely to be a small factor in the decision between different VCS due to their larger budgets. For a small game studio with less than five members, Perforce offers a free license making the cost of Perforce that much less. The hardest decision surrounds small studios who wish to grow and those who already have over five developers but who are still on a tight budget. The costly nature of Perforce could be a big deterrent towards its use. Instead, many of these companies may opt-in to other VCS like Git due to the initial and ongoing costs being significantly less.

Despite Git LFS and the Partial Clone feature giving better management of blob data, one of the main reasons Perforce is used in the games industry is due to legacy reasons. Perforce was initially released in 1996 and has been a popular VCS in the games industry ever since. This compares to Git: although Git has been around since 2005, Git LFS version 1.0 was not released until 2015, with support for 'locking' (an integral feature in Perforce, not an additional feature) not appearing in Git LFS until version 2.0 (released in 2017). Partial Clone is even newer, being initially released in late 2018, with real adoption not being seen until early 2020. This is one of the key reasons that Perforce is more popular than Git - the tooling needed to manage Git at scale is new. Many games companies would be reluctant to move over to Git, as all their developers already know how to use Perforce, as well as existing pipelines already being present with Perforce. The Unreal Engine also has the use of Git currently marked as *'in beta'*, which shows that even in one of the most popular game engines, Git is still being tried and tested as an alternative to Perforce.

Although many games companies may already have existing pipelines within Perforce, due to the overall popularity of Git and its open-source nature, new development pipelines, especially CI/CD (see *section 3.2*), are much simpler to setup. Hosting services, especially GitLab, have many development integrations built-in, such as CI/CD, issue tracking, code review,

and more[50]. This means that game developers can use modern software engineering tooling without having to develop specialist pipelines for their projects.

For studios that cannot decide between Perforce and Git, Perforce offers a solution. Helix4Git is a high-performance Git server that resides inside of a Perforce server[51]. This allows for Git developers to continue to work on code using Git, and access source code, as usual, using Git commands while also leveraging the single source of truth that Perforce offers. This allows for a 'best of both worlds' approach to version control, as developers can use the ease of blob versioning within Perforce while continuing to use Git that is quick and efficient for code changes.

Overall, Git and Perforce both have their advantages and disadvantages as VCS. Perforce is a robust, scalable tool that has been designed to manage large amounts of data and has been the defacto standard in the games industry for years, but it has costs that incur due to its required commercial license and costs of self-hosting servers. It is also not open-source, which makes integrating custom development pipelines and workflows within Perforce harder. Despite this, Perforce is still a tried and tested advanced solution that many games companies have used for years. Many game developers already have existing knowledge of Perforce, which makes the integration process for new employees easier. In the past, Perforce was a clear winner and an easy choice in the game industry. With the introduction of features such as Git LFS, the decision is now a much closer one and will very much depend on the specific requirements and needs of an individual game studio.

# Chapter 4: **Game Design Patterns**

> *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice".* — Christopher Alexander[52]

This chapter will discuss the merits of design patterns, how they are beneficial, especially within the context of the gaming industry, as well as discuss a few patterns that are specifically applicable to games and how the architecture of the Unreal Engine was influenced by these patterns. This will be done by exploring patterns that are injected into games via the use of the Unreal engine, as well as patterns that can be implemented by the user. Implementations of some of these game patterns will then be explored within *Chapter 5*.

## 4.1 Why Design Patterns?

Designing good object-oriented software is hard, and designing reusable object-oriented software is even harder[5]. Programs need to be specific enough to tackle the problem that they are designed for but must also be general enough that they can be used to handle any similar problems that may or may not appear in the future. Re-design should be minimised as it takes time to do, but it will always be inevitable when trying to make two similar, but different problems work within the same codebase.

This is where design patterns come into practice. They are a software engineering solution that helps to prevent programmers from 'reinventing the wheel'. By using predefined approaches to similar problems, programmers can spend less time designing and iterating over ideas to find the best approach to implementing new architecture and instead spend valuable development cycle time working on optimising architecture that cannot fit into a predefined mould.

## 4.2 Patterns Injected By Engine

This section contains information on game design patterns that are injected into a game by the engine. This is where developers do not need to consider the implementation of a specific pattern, as, by design, the engine uses these patterns to ensure game optimisation.

### 4.2.1 Flyweight Pattern

This section intends to cover what the Flyweight Pattern is, its importance in minimising memory usage within applications, as well as how the Unreal Engine architecture has been designed to use Flyweight internally.

The Flyweight Pattern is a software design pattern that allows for more objects to be stored in the available amount of RAM on a system by sharing common properties between multiple objects instead of keeping all the data in each object[53]. This allows for objects with similar traits to be stored as one shared entity, greatly reducing the amount of memory required.

This pattern makes use of two key parts - the intrinsic and extrinsic state. The former contains

any data that is repeated across all the objects, and the latter makes up all the contextual data that is unique to each object. An example of this is the use of projectiles. A projectile's intrinsic state can contain many similar traits: such as damage dealt, the speed of the projectile and the appearance. The context class (the extrinsic state) stores the different data between each instance, namely: the location of the projectile, rotation of the projectile, and more.

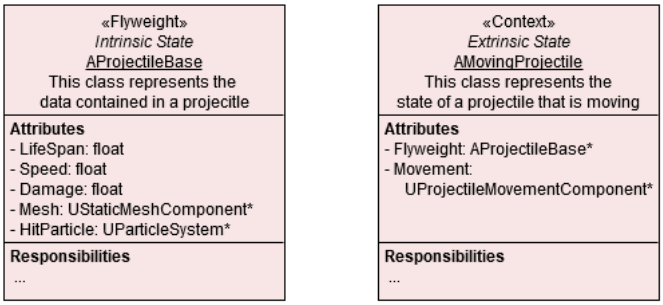



Figure 4.1: Example of information stored in intrinsic and extrinsic state for a Projectile

A factory is typically used within the implementation of the Flyweight Pattern. As objects are shared, clients should not be allowed to instantiate them directly. The factory is used to create a flyweight and then manages the pool of flyweight objects. This allows for clients to locate a particular object through the Factory through an *associative store* that lets clients look up flyweights of interest.[5].

### Advantages

- Reduces the amount of RAM required by sharing an intrinsic state between each flyweight, so that common properties do not need to be copied into each object individually. The Gang of Four[5] says, "The cost savings increase as more flyweights are shared, with the greatest savings occurring when the objects use substantial quantities of both intrinsic and extrinsic state, and the extrinsic state can be computed rather than stored".

### Disadvantages

- May introduce run-time costs that are associated with transferring, finding, and computing the extrinsic state[5].

## Flyweight in Unreal

Objects in The Unreal Engine are tagged using a `UCLASS` macro. Internally, each UClass maintains an object called the 'Class Default Object' (CDO), which is an immutable default template object that is generated by the class constructor[54]. This is perfect for the creation of a Flyweight.

As the CDO is immutable, it can be used to represent an intrinsic state class. All non-changing elements can be stored, and then a pointer to the CDO (Flyweight) can be referenced within the Context. The Client can then *draw* the context to the game world using the `GetWorld()->SpawnActor<ActorType>` method, which adds the actor to the game world for the engine to process.

Factories are practical when all Flyweights are not wanted to be created upfront, as well as when a developer cannot predict what objects will be needed. This approach does not fit into the flow of a game designer modifying variables exposed by a games programmer. Typically within

Unreal, the parameters of the CDO are tagged using a `UPROPETY` macro, which exposes any properties tagged to the *Details Panel*. This panel contains information, utilities and functions specific to a current selection, allowing game designers to easily change the properties of an object visually within the editor without needing to understanding how to program.

In the example of projectiles, it is likely all the properties are known upfront. In this case, it is better to create a Blueprint instance and then parenting it to the C++ class, which allows for the individual projectiles to be manually created by a games designer, where they can visually modify all the properties of the projectile.

An implementation of the Flyweight Pattern in Unreal can be found in *Chapter 5.1*.

### 4.2.2 State Pattern

This section intends to cover what the State Pattern is, different ways that state can be used, and how the Unreal Engine uses the state pattern substantially within its Animation System.

The State Pattern is a software design pattern that allows for an object to alter its behaviour when its internal state changes. The object will appear to have changed class[5]. The state pattern is a cleaner way for an object to change its behaviour at runtime without having to use a long list of conditional statements to define its behaviour.

#### Advantages

- Reduces conditional complexity, removing the need for if and switch statements that have different behavioural requirements that are unique to the state the object is in
- Much easier to add additional state to an object due to the reduced conditional complexity
- State transitions are explicit - it is easy to understand how one state can move to another

#### Disadvantages

- There are many times when a state is not needed due to the simplicity of the problem and a simple if statement would suit, introducing a state can add added complexity that is not required

## 4.3 Implementable Patterns

This section contains information on game design patterns that can be implemented into the Unreal Engine. This is where the developer would need to consider the implementation of a specific pattern, either by using a feature exposed explicitly by the Unreal Engine or by coding the pattern by conventional means.

### 4.3.1 Observer Pattern

This section discusses the Observer pattern and its importance in ensuring that classes do not need to be tightly coupled when one class influences many other classes.

The Observer pattern defines a one-to-many dependency between objects so that when one object, called the subject, all of its dependents, called observers, are notified and update automatically[5]. The observers can "subscribe" to the subject, and then the subject stores a collection of observers. When an event occurs within the subject that the observers should be notified of, the subject calls an `notify()` method on all of the observers, updating their state.

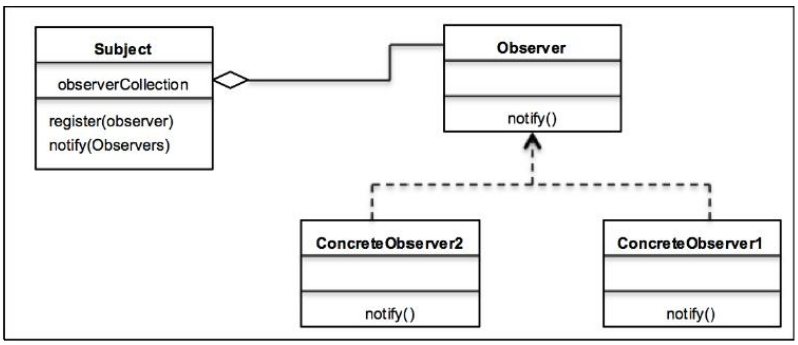



Figure 4.2: Observer Design Pattern[55]

### Advantages

- Helps to decouple classes, can easily introduce new observers without needing to change code within the subject as all it knows is the list of observers
- Allows for the creation of relationships between objects at runtime (by adding the observer to the subject's list of observers)

### Disadvantages

- As observers do not know of the presence of each other; they can be blind to the cost of changing the subject, causing a cascade of updates to observers and their dependent objects when making changes on the subject

## Observer in Unreal

The Unreal Engine makes use of Event Dispatchers in Blueprints, or delegates in C++, which allows for a class to report on its state, and then fire events to other classes that are interested in the event.

The 'Call Event' node in Blueprints is used to fire an event from the subject or `execute` method in C++. The classes that are interested in the event can then use a 'Bind Event' node in Blueprints, or `bind` function in C++, which creates an event listener at runtime from the observer to the subject. From here, a *custom event* can be created, which is the action that occurs when the event is fired.

An implementation of the Observer Pattern was used within the *Game of Tag* implementation in *Chapter 2.2*.

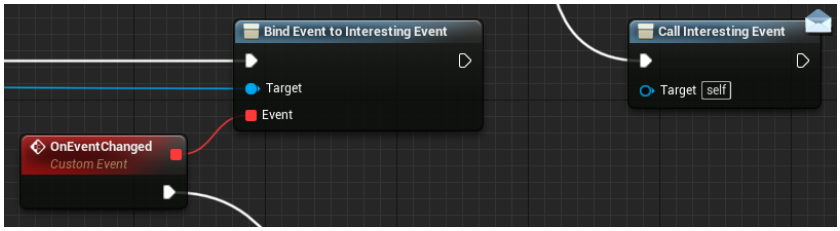



Figure 4.3: Blueprint Observer Events

# Chapter 5: **Proof-of-concept Development**

This section provides proof of concepts for the key programs of the project to show how they can be implemented in the Unreal Engine using game design patterns and key algorithms.

## 5.1 Projectiles using Flyweight

The Unreal Engine natively supports Flyweight (see Section 4.2.1) in its implementation. Although it does not seem obvious that it uses intrinsic and extrinsic classes, each spawned actor stores a pointer to the Blueprint class where, for example, the mesh and particles can be stored, rather than holding a copy per spawned actor.

This implementation creates a projectile in the form of a fireball. The character spawns a small fireball particle where their hand is, which follows their right hand, then when the characters hands are put forward, a projectile gets spawned using the `SpawnActor BP_Fireball` node.


Figure 5.1: Blueprint Observer Events

The projectile is setup using a Blueprint that inherits from the AActor class. From here, the object structure can be seen in *Figure 5.1*. The class has a `ProjectileMovement`, which is the extrinsic state of a moving projectile and is an Unreal class that holds all the movement responsibilities for an actor. The class also holds a CollisionObject, which is a sphere used for simple collision. The 'Flame' portion is an Unreal Particle System Component that has the fireball particle animation.

This projectile will be used as the main attack that a player has for the final game. By implementing this proof of concept, it shows how Flyweight is used within the Unreal Engine, as well as provides a basis for implementing animations when using state machines.

For the full Blueprint implementation with comments, see *Appendix F*.

## 5.2 Projectiles Reflecting off a 'Convex Mirror'

Originally this proof of concept intended to implement the mathematics behind a projectile bouncing off of a convex mirror, which would eventually represent the player having the ability to hold up a 'shield' to block other enemies attacks. However, this proved to be a very simple task, with the ProjectileMovement class already including a `Should Bounce` variable.

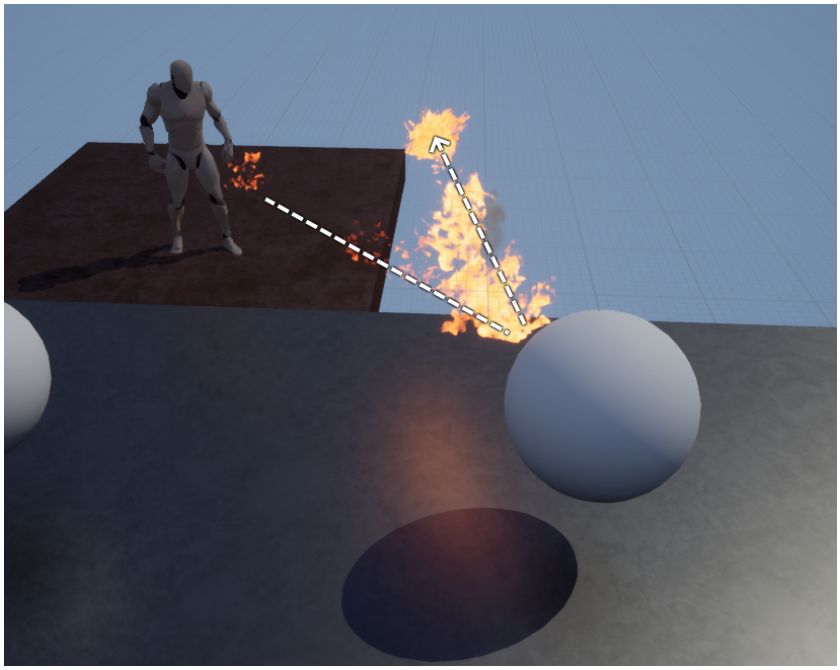


Figure 5.2: Line of Projectile Bounce

For the proof of concept, a sphere has been used as a simple default shape pulled from the `Place Actor` window. The projectile and the sphere also needed to be added to an `object channel`, which is functionality built into Unreal that allows you to define the collision presents of an object. The objects can subscribe to these channels to define the interaction between objects of the same type. The user interfaces for setting up collisions can be seen in *Appendix G*.

This proof of concept helps to reiterate the ideas of *Chapter 2*, which talks about the reasons people use game engines - ideas that can seem complex and challenging at first can be easy for a game designer to implement due to native features already supporting the desired functionality.

## 5.3 Testing in the Unreal Engine - Automated Spec Framework

The automated spec framework is one of the built-in automation tests within the Unreal Engine. This implementation uses a 'Spec', which is a term for a test built following the 'behaviour-driven development' methodology[56].

The proof of concept was implemented by converting a simple element of the `AICharacter` class, the *health function*, that was initially created in Blueprints, and moved this blueprint implementation was recreated into C++.

Part of the health function was to remove the character when its health reached zero. This was done by first spawning a character into a world, damage the character enough so that the character would be killed, and then check that the actor in the world was removed. For the code: see *Listing 5.1*.

```
void FAutomationSpec::Define() {
  Describe("A Character", [this]() {
    BeforeEach([this]() {
      // Create the world and spawn actor for tests on character
      World = FAutomationEditorCommonUtils::CreateNewMap();
      AICharacter = World->SpawnActor<AAICharacter>();
    });

    ..

    It("should be removed from the world when its health reaches 0",
      [this] {
        TArray<AActor*> FoundActors;
        AICharacter->SetMaxHealth(50.0f);
        AICharacter->TakeDamage(100.0f, DamageEvent, EventInstigator,
                                DamageCauser); // Reduce health below 0
        UGameplayStatics::GetAllActorsOfClass(
          World->GetWorld(), AAICharacter::StaticClass(),
          FoundActors);
        TestEqual("amount of actors in world",
                  FoundActors.Num(),
                  0);
    });

    ..

    AfterEach([this]() {
      // Remove the character to ensure previous tests don't conflict
      AICharacter->Destroy();
    });
}
```

Listing 5.1: Checking that an AI opponent when killed is removed from the world.

## 5.4 Testing in the Unreal Engine - Automated Functional Testing

Aside from the automated spec framework, there is another key method for creating tests. This is through Blueprint Functional Tests. These are tests that can be defined in Blueprints (see *Figure 5.3*) to carry out a test much like an automated test in C++ (see *Section 5.3*). The advantages of using Blueprint tests are much the same as the advantages of using Blueprints (see *Section 2.2.2*). The tests are easier to write as well as being more efficient when using blueprint actors due to not needing to hard-code a reference to its location.

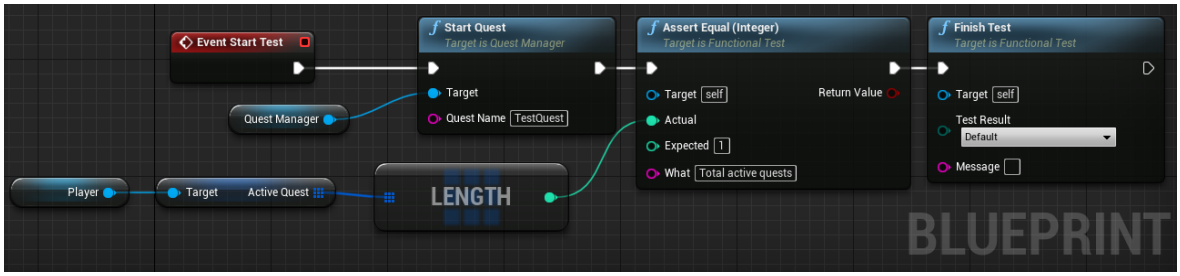



Figure 5.3: Blueprint Functional Test

*Figure 5.3* tests whether or not a quest gets added to the player's list of active quests. This is done by starting a new quest in the quest manager and then checking the length of the active quest array on the player to check that the length of the array is now 1.

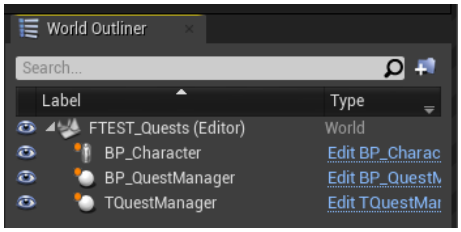



Figure 5.4: Level Components needed for test

This `TQuestManager` is added to a custom 'FTEST_Quests' level, with the corresponding quest manager and Blueprint character. During the testing process, this level is loaded and then the corresponding test is carried out (see *Figure 5.5* for the components in the world). Once the test is completed, then the level is closed down, and the next test is started. By using levels to carry out the test, it allows for detailed tests to be specified. For example, a character could be spawned into a world, and then running could be simulated, with the distance of the character being record after a set interval to ensure that the character is moving at the correct speed.

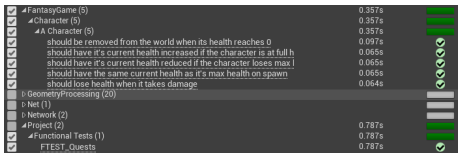



Figure 5.5: Speed of C++ versus Blueprint Tests

The main disadvantage of Blueprints Functional Testing is its execution speed. As seen in *Figure 5.5*, the 5 C++ Automated Spec Framework tests from *Section 5.3*) use the `FAutomationEditorCommonUtils::CreateNewMap()` function to create a world, and then `World->SpawnActor<AAICharacter>()` to spawn an actor. This process creates a new world that runs the test and then breaks down the world in *0.357 seconds*. This contrasts to the Blueprint implementation which runs *one* test in *0.787 seconds*. These numbers seem insignificant, but games can have tens of hundreds of tests to cover most of the possible edge cases. Five C++ tests in 0.357 seconds versus one blueprint test in 0.787 seconds for *ten thousand* tests is the same as 12 minutes versus two hours and 11 minutes.

Blueprint tests have their use. They are much simpler to set up and a better solution for testing blueprint actors. However, they do have a performance hit over that of C++ tests, and therefore should only be used where most applicable.

# Chapter 6: **Game Development**

This chapter sets out to present how the final game in the Unreal Engine has been implemented and developed using ideas set out in the report, along with how the proof-of-concept programs have been realised into a game. Any notable features that have been added that were not discussed in the proofs-of-concept development section (see *chapter 5*) will also be discussed.

## 6.1 Game Information

Pursual is a small quest-based fantasy-style role-playing game. The user is set on a series of small quests, which allow them to save a town from destruction. The user starts by discovering the small village of Twerth and then is set on a task to free the villages from thieves who have raided the village.

Complete information on how to play the game can be found in *Appendix D*, including how to find and download it.

## 6.2 Gameplay Systems

This section contains information on some of the vital gameplay systems implemented into the game that are not proofs-of-concept.

### 6.2.1 Dialogue System

The dialogue system allows a user to get information about the world from different actors. There are currently two types of actors: the `BP_SignPost` and the `BP_NPC`. *Figure 6.1* shows an overview of how the classes in the dialogue system interact. There is an interface that defines the Interact class to ensure that any classes that wish to implement the dialogue system have an interact function. Both classes also use composition to access the `BP_InteractComponent`, which implements common behaviour functions between the two classes.

The interface also allows for the 'Get All Actors With Interface' blueprint node to be called so that the player can easily check when they press the interact button if they are overlapping another actor that implements the interface.

### 6.2.2 Quest Management System

The quest management system's design is a modular approach that is expandable to accommodate any number of quests.

#### Inital Design

Initially, the quest management system inherited from the AActor class and was placed in the world. The idea was that every object that needed access to the quest manager could easily

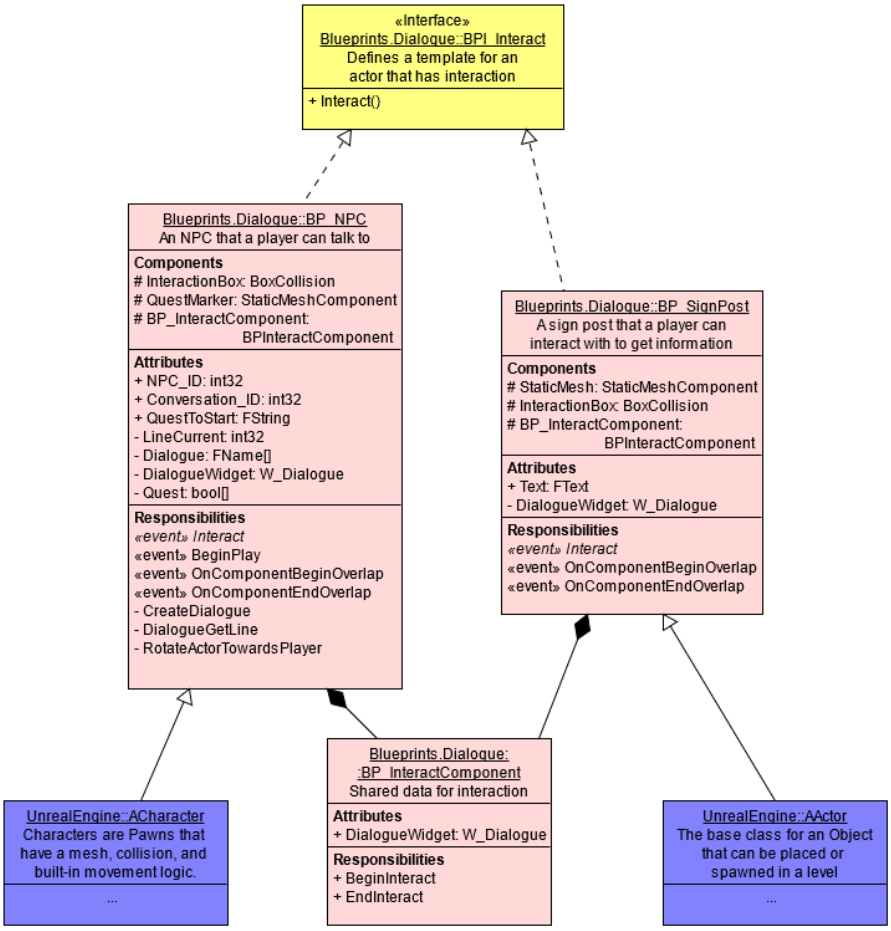



Figure 6.1: UML of the Dialogue System

access it. The problem with this approach was that the quest manager is needed by many objects, which created a long list of dependencies, as every object that needed access to the quest manager needed to store a reference to the quest manager location. The alternative option would have been to have used a `GetWorld()` method to find the quest manager in the world, but this is an inelegant solution at best and an inefficient solution at worst.

### Final Design

The shortcomings of the quest manager discovered in the initial design were improved upon by changing the class to a `UActorComponent`, from that of a `AActor`. The `UActorComponent` is an Unreal Engine class that is used to define reusable behaviour that can be added to different types of Actors. By using this component, the quest manager could be attached to the player character. As quests were only executed when the player made an action, it made sense to keep the quest manager here. This also meant that an object that needed to access the quest manager could instead access the player, and then once the player was accessed, they could get the quest manager from the players component list. Using a component also allows for additional characters to be added later that can use the quest manager. All that is required is for the quest manager to be added to the character, and then they would have full access to the quest manager's functionality.

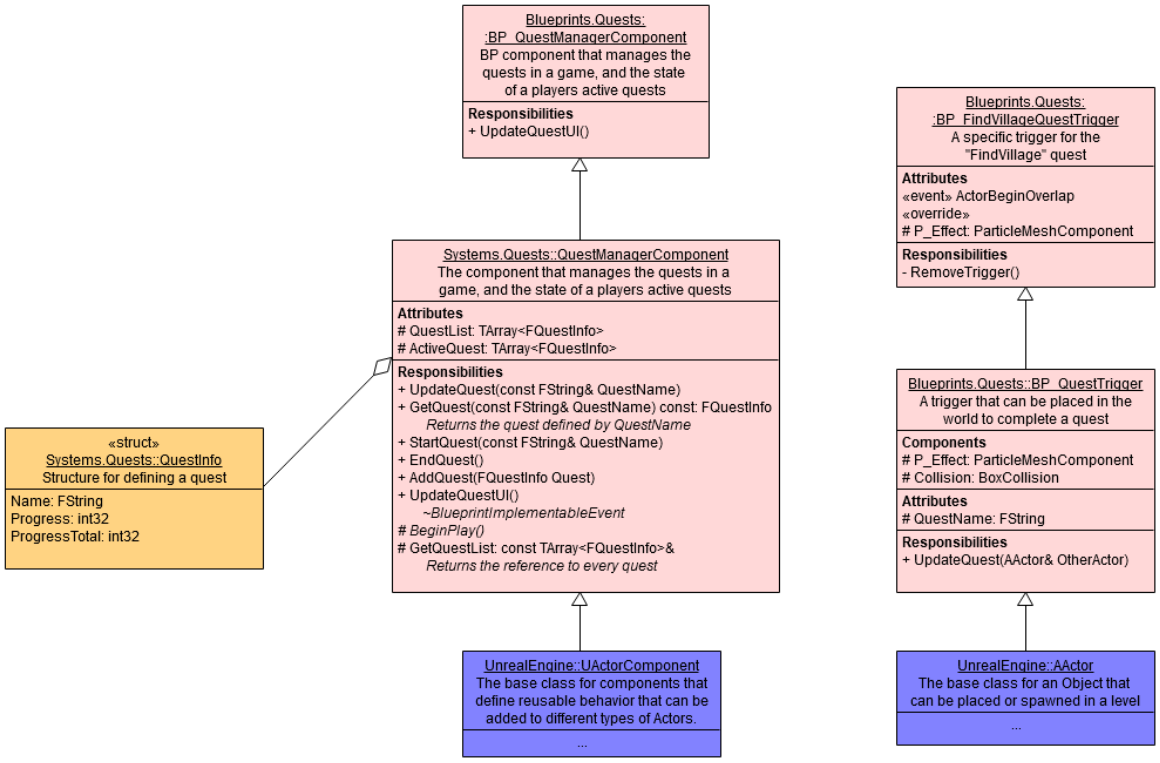



Figure 6.2: UML of the Quest Management system

The 'Current Quest UI' that appears in the top left of a players screen is managed by the level blueprint. Using a quest manager that is attached to the player character also ensures that only the quest manager needs a dependency on the level blueprint to update the players UI (through the use of an observer), rather than each object, each needing a dependency.

The `QuestTrigger` is a class that inherits from the AActor class and is used to update the progress of a quest when the player walks into its collision box. When an overlap occurs, an event is started that talks to the quest manager to deal with the update of the quest, as well as removing the trigger from the world, ready for garbage collection.

## 6.3 Software Engineering

This section shows the software engineering principles that have been put into action while developing the game Pursual. It explores how version control has been managed., the tooling that has been used, as well as an overview of the proof of concepts testing that has been applied.

## 6.4 Version Control Branching

Pursual was initially developed using the standard git workflow. Each new feature that was worked on (e.g. the quest system) was put into a branch. The benefits of this process are the fact that it helps ensure that unstable code is not merged into the main codebase and also ensures that incomplete features are not present in the main codebase. The issue that stems from this process is the way that the Unreal Engine stores engine files. A lot of these files are generated by the engine when it is launched so that they do not need to be stored within version control and are created on a per-version basis. The problem stems from the fact that these files are different for each version of the game; therefore, when changing to a different branch, first, it is required that the game editor is closed. Afterwards, it is possible to change the current branch, but then the game must be recompiled before reopening the editor so that changes can be made. This process adds significant development time, especially if a developer forgets to follow this exact process.

This reason is where 'Trunk-Based development' comes in; a source-control branching model, where developers collaborate on code in a single branch and therefore avoid the need to merge changes. This approach was used to finish the game as, at times, many different branches were being worked on at once. Too often, early on in development, the process of needing to close the engine first before changing to a different branch would be forgotten, which would end up creating conflicts within the codebase that needed to be resolved. Another issue is the fact that binary files cannot be merged easily. If changes were made to two different blueprint files, the process of merging them through Git would be impossible, and the conflict would have to be resolved in other ways.

Typically a version control like Perforce or use Git LFS would be used to manage changes as a developer can access the binary file where needed, but for this project, it was required that either a departmental Subversion or GitHub repository is used. GitHub offers a maximum of 2GB of storage for binary files, which for the use case of this project, would not have been enough. Therefore it was chosen not to use Git LFS, and instead, the files were just backed up as usual in Git.

## 6.5 Testing

Two of the proofs-of-concept were different approaches to testing in the Unreal Engine. These were namely a Behaviour-Driven Development approach using the Automated Spec Framework (see *section 5.3*), as well as using the automated functional testing blueprint approach (see *section 5.4*).

These give good examples of how an entire game can be developed using a testing framework to allow for systems such as CI/CD (see *section 3.2*) to help ensure working releases. It would

have been advantageous to have used testing for all the systems put into place, but writing good tests can be difficult, especially when someone is in the process of learning how everything works. Much time was also spent redesigning components to work more efficiently due to the report progressing and giving an improved overall understanding of the Unreal Engine.

## 6.6 Tooling

The game was developed using a few essential tools (further explanations of these tools can be found in *Appendix H*). Trello has been used as a management software to ensure that each section of the game is prioritised correctly, as well as easily being able to view what parts of the game had been completed and that still needed doing. Trello is a simple tool, but its flexibility still helps significantly when tackling a problem.

Rider was used as the IDE of choice due to its excellent integration with the Unreal Engine. Additionally, the game was backed-up to GitHub using the GitKraken Git client.

## 6.7 Conclusion

Overall, many different aspects of the report came together to built this game. Not all parts have been visited in the full capacity that would be essential to making a commercial game (such as a full testing suite), but many different elements have been implemented, such as the observer pattern, the flyweight pattern, testing, as well as good use of version control.

# Chapter 7: **Professional issues**

Computing professionals' actions change the world. To act responsibly, they should reflect upon the wider impacts of their work, consistently supporting the public good.[57]

This section will discuss some of the key issues that can arise when developing games in Unreal Engine, and in general, and what considerations can be made to overcome these problems.

## Licensing

When publishing a game using the Unreal Engine, those who publish the game must consider the Unreal Engine End User License Agreement (EULA) For Publishing [58]. The Unreal Engine is free to download and use for personal use, but once a game is published and can be bought by other users, fees can occur. Currently, a 5% fee is incurred when a video game is monetised, and the gross revenue of the game exceeds $1,000,000, which must be taken into account when distributing a game. This revenue includes aspects such as in-app purchases, in-app advertising, subscription fees and virtual currencies. Because of this, a publisher must ensure that the revenue any specific game is making is being recorded and stored correctly. Although a small game company may not earn enough revenue to require paying the Unreal Engine license, they are still obligated to ensure that their books are in order. If a game suddenly broke the $1,000,000 revenue mark, and up till then, they had not been keeping track of sales (due to no expectation to meet the required revenue), then the publisher would have broken the EULA and could subsequently be sued.

## Usability and Accessibility

*"Be fair and take action not to discriminate."*

— ACM Code of Ethics and Professional Conduct[57]

Computer Scientists are ethically required to ensure that the quality of life of all people using our product is not affected. We must ensure that no matter who is using the game we develop, that they can enjoy the game to the fullest, and as such, assessability is of great importance when considering video game development. Many people who play video games have a disability and rely on accessibility features to ensure that they can play the game as it is designed. These can range from simple features such as adding colour changing options to help those who are colour blind to more complex features such as ensuring that precise timing is not essential to gameplay.

The Game Accessibility Guidelines[59] is a website for developers that gives a straightforward reference for inclusive game design. This is a straightforward and developer-friendly guideline that serves as an easy reference for ways in which to avoid unnecessarily excluding players from enjoying a game. Although not all of the guidelines will apply to every game, many of the considerations can be used to benefit many players.

If accessibility guidelines are not followed, many users will be simply unable to play a game. Ethics aside, 15% of the population are disabled[60], making it good business sense to incorporate accessibility features and allow more people to play the game as intended. By incorporating accessibility features, a game will be opened to a wider audience and has the potential to be more popular due to the number of games that support a full range of accessibility is limited.

## Management

Many game studios work with publishers to develop and release games. The job of a publisher is to give developers support when making a game. This can include giving money to make a game (such as hiring developers, artists and paying for maintenance costs), as well as provide marketing, quality assurance, brand management, and the complexities of selling games in other countries[61].

This relationship between a game studio and publisher requires appropriate management from both sides. A publisher is investing in a games company and providing expertise to help the games studio succeed, which means that they expect a level of transparency and dedication from the studio. The studio is obligated to create a game to an agreed brief that both the studio and publisher have agreed to prior. If either party for any reason feel the need to defer from the original plan, then they must first discuss the reasons for doing so with the other.

## Trust

Games (usually) cost money. A user finds a game that they like, for example, by looking through a curated game store and then chooses to purchase the game. As the user is paying for a service, the user is trusting that the game they have purchased meets certain technical conditions. The user expects the game to run on their machine (as long as they have above the advertised minimum hardware requirements). The user also expects the game to run smoothly and without major disruption. If the game is full of bugs and problems, then the level of competence expected is not met.

The ACM Code of Ethics and Professional Conduct states that a computing professional must be *'honest and trustworthy'* as well as having the professional responsibility to *'Strive to achieve high quality in both the processes and products of professional work.'*[57]. When a user buys a game, they become an end-user, and as such, the developers of the game become committed to ensuring that the game works as expected and to the best standard possible. Bugs will likely always be present in games due to their complexity, but that does not mean that developers should ignore them and sell a product that they know to have many 'major' bugs. Publishers should not lie to their customers about the quality of a game and instead should work to use appropriate methodologies, such as automated testing (see Section 3.1.3), to guarantee the quality of a game.

# Chapter 8: **Assessment**

## Summary of Completed Work

This chapter compares the original plan to the work that has been completed so far, the components that have changed since the original plan, as well as highlighting aspects of the plan that have were removed from the scope of the project.

### Project Reports

Table 8.1: Project Reports

| Deliverables | Progress | Additional Comments |
|---|---|---|
| Why is Unreal typically favoured over other game engines, and why has the industry moved away from in-house engines? | Completed | Moved to be a comparison between PyGame, Unity and the Unreal Engine, but PyGame was left out due to time issues, along with the move away from in-house. |
| What are the advantages and disadvantages of using Perforce as a Version Control System over other VCS such as Git or Subversion, and why is Perforce typically used in large-scale game development? | Completed | Also discussed the merits of how Git could be used in large-scale game development, even if not widely done. |
| What are common game design patterns, object-oriented or data-oriented programming approaches, and software methodologies used to approach game development? | Completed | Game design patterns discussed. OOP has been discussed over the project but never as a single section. Unity uses data-driven programming but did not get mention due to Unreal not using it. |
| Why does the industry typically not adhere to mainstream software engineering testing principles such as test-driven development, and what methods do they use instead to ensure bug-free releases? | Completed | Explanation of TDD introduced, as well as behaviour-driven development and how automated testing can be used to improve software development. |
| How can compelling and interactive user interfaces be created for games? | Redundant | Removed from the project scope. |

## Proof of Concepts

Table 8.2: Proof of Conceptss

| Deliverables | Progress | Additional Comments |
|---|---|---|
| Create Projectiles in 3D Space using the Flyweight Pattern | Completed. | - |
| Projectiles Reflecting off a "Convex Mirror" | Completed | Initially planned to use vector mathematics to implement bouncing, but found this to be built into Unreal's projectile system already. |
| Testing within the Unreal Engine Framework. | Completed | Discovered and found some of the 'best' ways to automatically test a video game using the Unreal Engine, using both the BDD method as well as Blueprint testing. |
| Fluid Combat While Covering Edge Cases using the State Pattern | In Progress | Will be a work in progress until all elements of the character are complete as edge cases are added. |
| Reactions Based on Player Event | Not Started | Other AI proof of concepts need to be implemented first. |
| Pathfinding Algorithms Implemented Using A* | In progress | Other AI proof of concepts need to be implemented first. |
| Interesting Opponents Through Artificial Intelligence | Redundant | This topic was too broad for a proof of concept. |
| Compelling Interactive User Interfaces | Redundant | Removed from the project scope. |

# Critical Analysis

This section is a self-assessment of the project's achievements, with an additional reflection on the difficulties faced during the project, as well as where the project could improve further given more time.

## What Went Well

Overall, I believe that the project has been a success. I was able to put into practice and further explore ideas that have were taught during my time at University. An example of this is the use of Design Patterns. Despite using design patterns during CS2800 Software Engineering in our Test-Driven Development project, this project has allowed me to understand further the nature of these patterns and how they are applied to develop better software. This project also allowed me to put this knowledge into the context of an industry that I wish to create a career in, making the learning experience all the more enjoyable and relatable.

Despite never having used the Unreal Engine before starting this project, I believe that I now have a good grasp of the engine and the reasons why certain parts of the engine were designed how they are. I also have a strong understanding of how to approach game development using the Unreal Engine. Although I still have much to learn, I am in an excellent place to find the knowledge that I need to enhance my experience further. I can use the documentation that Epic Games provides to find new features, and I have a better understanding of C++ due to

looking through the Unreal Engine source code to understand how certain parts of the engine were implemented.

The overall report is in a good place. I was able to get an overview of most of the specification that I set out to achieve, even if the said specification has evolved and adapted. There is good cohesion in how the report has been laid out and how each chapter's sections flow from one to another. There is also sufficient detail for each section, and I have included a significant amount of relevant references in the bibliography. There is also a glossary for relevant terms to make sure that it is clear at all times what has been discussed. A user manual has also been included on how to run the game of tag, along with the final game.

## What I have Learnt

In addition to putting into practice and further exploring ideas taught at University, I was also able to learn about new software engineering tools, concepts, and methodologies that I had little prior knowledge of and had not used before. An example of this is the process of Continuous Integration and Continuous Delivery. Despite having heard of this DevOps practice, it was not something that I had personally any experience with, while also not having a good understanding of what it was before this project. After talking to representatives from both Mediatonic and Red Kite Games, I was able to understand how CI/CD is progressively being used more and more within the games industry, especially coupled with automated testing. This project and the surrounding research that I did gives me a much better understanding of how a career in the games industry will progress.

Another significant concept that I have understood is the use of Perforce in the games industry. Many of the companies that I would likely go on to work with already use Perforce, and therefore having experience with Perforce is a huge bonus. I believe that Git could be a contender in the upcoming years against Perforce, especially for smaller teams on a tighter budget. However, I can also understand the reasons that, historically, Perforce is used and is continuing to be used. Perforce definitely does the job it has been designed to do well.

Another interesting topic was that of Test-Driven Development. The CS2800 Software Engineering module at university advocated for its use highly. I believe it to be a potent tool at the disposal of software engineers, especially considering the use of continuous integration pipelines. When I came into this project and decided to tackle the use of TDD in the games industry, I first started to wonder why TDD was not used in the games industry and if it was just because people did not understand it correctly. The more I researched, the more I realised the reasons why traditionally has not been used in a game development pipeline. While it can do iterative development, the tests written with TDD need to change often throughout due to games requirements changing much more often than traditional software projects. That being said, I believe TDD should still be used when taking a working prototype and rebuilding this prototype using good software engineering approaches.

The final central aspect that I learnt is the use of Unreal's C++. Before touching the Unreal Engine, I had no prior knowledge of C++ (only some knowledge of C). Even though Unreal Engine's version of C++ is quite different from that of standard C++ because most of the standard libraries are reimplemented, the overall syntax of C++ and structure is the same. This project has allowed me to learn a new language that is widely used within the game development space and will be a valuable tool for a career in games development.

## What Went Wrong

One of the sections that I would have liked to have explored in more detail is my comparison between the Unreal Engine and Unity. I have spent much time researching and using the Unreal Engine in preparation for the final game, creating an understanding of the engine architecture so that I could write about design patterns and overall have good knowledge of how the engine works. This process has allowed me to have a definite opinion about how the engine works and its merits and demerits. The issue stems from the fact that I do not have the same understanding of how Unity works. This lack of knowledge has led to my section on Unity vs the Unreal Engine being somewhat biased. Rather than learning the 'Unity way' of tackling a problem, I came to Unity with only my understanding of how the Unreal Engine works and therefore wanted to do things the Unreal Engine way rather than the way in which they designed.

An example of this is AI behaviour trees. Yes, behaviour trees could be implemented in Unity, but the 'Introduction to AI' course in Unity was sixteen hours long (plus the time needed to learn and implement that spoken about in the course), and as such, I could only compare it to the knowledge I already had. It already took me a significant amount of time to learn parts of C#, alongside learning an overview of how Unity works, sadly making it so that I could not have the most accurate impression to compare both engines. I tried to take an objective approach, but many of the comparisons that I found online comparing the two engines that I used to support my findings are biased by nature, and therefore made it hard to be objective to the matter.

The main issue with creating a game in Unity was the time that it took to learn C#, a new game engine, and a new approach to making games. This time is taken leads on into the comparison between PyGame and the Unreal Engine. Initially, I wished to compare the Unreal Engine, Unity, and PyGame compare two different game engines, along with an approach that did not use an engine. This comparison would help me support a narrative on how game engines are being used to make games easier to make. The first hurdle to making this comparison was that I was unable to get a response to a questionnaire that I had given to a member of the Frostbite game engine team. As EA is a publicly-traded company, it made it hard to get the required permissions to answer the questions I wished to ask, and therefore meant that I was unable to respond in a timely manner to complete my report. The second issue was that it took me significantly more time to learn Unity than I had planned. This delay meant that I did not have the time to, again, learn a new approach to making games (especially without an engine) which would have taken too long and could have been started, but either would not be finished or would have caused something else to be uncompleted. Due to both these reasons, I decided to remove the use of PyGame from this project. It would have been an exciting topic to approach, and given more time, it would have been beneficial towards my report and overall understanding of game development.

The last criticism would be the breadth of some of the parts of the report. I went into quite a lot of detail with certain parts of the report. For example, I discussed manual testing and automated testing in reasonable detail with testing, which gave me a good understanding of how these work. The problem is that there are many other types of testing used, for example, feature testing, smoke testing, localisation testing, or screenshot testing, to name a few. There was no mention of any of these features; therefore, if I were to tackle this project again, I would likely try to cover more aspects of a topic rather than specific parts in great detail.

## Future Enhancements

One of the main aspects of the project that I would have liked to have improved on, given more time, was the overall gameplay experience. Although this is not a crucial part of the project, the feel of a game can influence someone's decision on how good a project is. Despite the animations used coming from Mixamo, my knowledge of animations and skill of implementation was poor, and it made some of the attacks feel 'clunky'. I would have liked to have spent more time polishing the gameplay so that there was a better, overall, cohesive experience, but as a software engineer in the AAA industry, these jobs would be done to someone with specific animation experience, and therefore overall, it was not paramount to the success of the project.

An excellent future enhancement would be implementing some of the Game Accessibility Guidelines discussed in the Professional Issues section. If I choose to continue to work on this game past the project deadline, the guidelines are something I will definitely implement, as most of the suggestions on this document are features that are trivial to add if considered early enough but can become increasingly more complex the further into a game that they are considered.

Another future enhancement would be the use of the AI behaviour trees in the Unreal Engine. The AI work that I have done is relatively simple compared to what is possible, and I would have liked to have spent more time exploring the intricacies of what could have been done. Currently, the AI finds a random spot in the world to run to, and if they find the player through the AI perception system, they will run towards the player and then attack. In real life, people would not just randomly run around and would have a much better thought process, and as such, looking more into detail of the AI of my game would be an interesting topic to discover.

# Bibliography


[1] Scott Steinberg. *Videogame Marketing and PR: [The Definitive Guide]. 1, 1,* New York, NY: iUniverse, 2007. ISBN: 978-0-595-43371-1.

[2] *Video Game Market Value Worldwide 2023*. Statista. URL: https://www.statista.com/statistics/292056/video-game-market-value-worldwide/ (visited on 2nd Dec. 2020).

[3] Perforce. *The State of Game Development Report - 2020 & Beyond*. 2020.

[4] Atlassian. *Git LFS - Large File Storage | Atlassian Git Tutorial*. Atlassian. URL: https://www.atlassian.com/git/tutorials/git-lfs (visited on 25th Feb. 2021).

[5] Erich Gamma, ed. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Reading, Mass: Addison-Wesley, 1994. 395 pp. ISBN: 978-0-201-63361-0.

[6] Robert Nystrom. *Game Programming Patterns*. s.l.: genever benning, 2014. 345 pp. ISBN: 978-0-9905829-0-8.

[7] 'Building a Game of Tag Using Behavior Trees in Unreal Engine 4'. In: (9th Oct. 2018). Ed. by Paul Gestwicki. URL: https://www.youtube.com/watch?v=vXtfOM4Vy8w&list=UUa0T9seDefPcvC6yb9DFnjg&index=28 (visited on 16th Nov. 2020).

[8] *What Is a Game Engine?- GameCareerGuide.Com*. URL: https://www.gamecareerguide.com/features/529/what_is_a_game_.php (visited on 16th Nov. 2020).

[9] *The Two Engines Driving the $120B Gaming Industry Forward*. CB Insights Research. 20th Sept. 2018. URL: https://www.cbinsights.com/research/game-engines-growth-expert-intelligence/ (visited on 21st Sept. 2020).

[10] *League of Legends MAU 2016*. Statista. URL: https://www.statista.com/statistics/317099/number-lol-registered-users-worldwide/ (visited on 2nd Dec. 2020).

[11] *Curious: What Programming Language? - League of Legends Community*. URL: https://web.archive.org/web/20151114143904/http://forums.na.leagueoflegends.com/board/showthread.php?t=16318 (visited on 17th Nov. 2020).

[12] Unity Technologies. *Riot Games Chooses to Build Next Games in League of Legends Franchise on Unity | Unity*. URL: https://unity.com/our-company/newsroom/riot-games-chooses-build-next-games-league-legends-franchise-unity (visited on 2nd Dec. 2020).

[13] *The Future of League's Engine*. URL: https://technology.riotgames.com/news/future-leagues-engine (visited on 2nd Dec. 2020).

[14] *Unreal Engine 4 on Github*. Unreal Engine. URL: https://www.unrealengine.com/en-US/ue4-on-github (visited on 2nd Dec. 2020).

[15] In: *Unreal Engine*. 30th Nov. 2020. URL: https://en.wikipedia.org/w/index.php?title=Unreal_Engine&oldid=991528528 (visited on 7th Dec. 2020).

[16] *Introduction to Blueprints*. URL: https://docs.unrealengine.com/en-US/Engine/Blueprints/GettingStarted/index.html (visited on 2nd Nov. 2020).

[17] *Nativizing Blueprints*. URL: https://docs.unrealengine.com/en-US/ProgrammingAndScripting/Blueprints/TechnicalGuide/NativizingBlueprints/index.html (visited on 7th Dec. 2020).

[18] *Programming with C++*. URL: https://docs.unrealengine.com/en-US/ProgrammingAndScripting/ProgrammingWithCPP/index.html (visited on 7th Dec. 2020).

[19] *VALORANT's Foundation Is Unreal Engine*. Unreal Engine. URL: https://www.unrealengine.com/en-US/tech-blog/valorant-s-foundation-is-unreal-engine (visited on 7th Dec. 2020).

[20] *Diffing Blueprints*. URL: https://www.unrealengine.com/en-US/blog/diffing-blueprints?sessionInvalidated=true (visited on 8th Mar. 2021).

[21] *Converting Blueprints to C++*. Unreal Engine. URL: https://www.unrealengine.com/onlinelearning-courses/converting-blueprints-to-c (visited on 10th Mar. 2021).

[22] *Download Unity!* Unity. URL: https://unity3d.com/get-unity/download (visited on 27th Feb. 2021).

[23] Unity Technologies. *Multiplatform | Unity*. URL: https://unity.com/features/multiplatform (visited on 27th Feb. 2021).

[24] Ars Staff. *Unity at 10: For Better—or Worse—Game Development Has Never Been Easier*. Ars Technica. 27th Sept. 2016. URL: https://arstechnica.com/gaming/2016/09/unity-at-10-for-better-or-worse-game-development-has-never-been-easier/ (visited on 27th Feb. 2021).

[25] Unity Technologies. *Unity - Manual: Managed Plug-Ins*. URL: https://docs.unity3d.com/Manual/UsingDLL.html (visited on 10th Mar. 2021).

[26] *ECMA-334*. Ecma International. URL: https://www.ecma-international.org/publications-and-standards/standards/ecma-334/ (visited on 10th Mar. 2021).

[27] Unity Technologies. *Unity - Manual: How IL2CPP Works*. URL: https://docs.unity3d.com/Manual/IL2CPP-HowItWorks.html (visited on 11th Mar. 2021).

[28] 'The Essence of C++'. In: (6th May 2014). Ed. by Bjarne Stroustrup. URL: https://www.youtube.com/watch?v=86xWVb4XIyE.

[29] *Releasing the Unity C# Source Code - Unity Technologies Blog*. 26th Mar. 2018. URL: https://blogs.unity3d.com/2018/03/26/releasing-the-unity-c-source-code/ (visited on 11th Mar. 2021).

[30] Phillip A. Laplante. *What Every Engineer Should Know about Software Engineering*. What Every Engineer Should Know 40. Boca Raton: Taylor & Francis, 2007. 311 pp. ISBN: 978-0-8493-7228-5.

[31] Glenford J. Myers, Corey Sandler and Tom Badgett. *The Art of Software Testing*. 3rd ed. Hoboken, N.J: John Wiley & Sons, 2012. 240 pp. ISBN: 978-1-118-03196-4 978-1-118-13313-2 978-1-118-13314-9.

[32] *What Is Alpha Testing and Beta Testing: A Complete Guide*. URL: https://www.softwaretestinghelp.com/what-is-alpha-testing-beta-testing/ (visited on 18th Mar. 2021).

[33] *Alpha Testing Vs Beta Testing: What's the Difference?* URL: https://www.guru99.com/alpha-beta-testing-demystified.html (visited on 18th Mar. 2021).

[34] Julio Cesar Sanchez, Laurie Williams and E Michael Maximilien. 'A Longitudinal Study of the Use of a Test-Driven Development Practice in Industry'. In: (2007), p. 10.

[35] Dorota Huizinga and Adam Kolawa. *Automated Defect Prevention: Best Practices in Software Management*. Hoboken, N.J: Wiley-Interscience ; IEEE Computer Society, 2007. 426 pp. ISBN: 978-0-470-04212-0.

[36] Unreal Engine, director. *Automated Testing at Scale in Sea of Thieves | Unreal Fest Europe 2019 | Unreal Engine*. 20th May 2019. URL: https://www.youtube.com/watch?v=KmaGxprTUfI (visited on 2nd Feb. 2021).

[37] *CI/CD*. In: *Wikipedia*. 19th Jan. 2021. URL: https://en.wikipedia.org/w/index.php?title=CI/CD&oldid=1001461006 (visited on 1st Feb. 2021).

[38] Atlassian. *What Is Continuous Integration*. Atlassian. URL: https://www.atlassian.com/continuous-delivery/continuous-integration (visited on 1st Feb. 2021).

[39] *Rare - Tech Blog: Adopting Continuous Delivery (Part 1)*. URL: https://www.rare.co.uk/news/tech-blog-continuous-delivery-part1 (visited on 2nd Feb. 2021).

[40] *Git - About Version Control*. URL: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control (visited on 30th Nov. 2020).

[41] @Scale. *Why Google Stores Billions of Lines of Code in a Single Repository*. 14th Sept. 2015. URL: https://www.youtube.com/watch?v=W71BTkUbdqE (visited on 30th Nov. 2020).

[42] Jon Loeliger. *Version Control with Git*. 1st ed. Beijing ; Sebastopol, CA: O'Reilly, 2009. 310 pp. ISBN: 978-0-596-52012-0.

[43] *Git vs. Perforce: How to Choose (and When to Use Both)*. Perforce Software. URL: https://www.perforce.com/blog/vcs/git-vs-perforce-how-choose-and-when-use-both (visited on 30th Nov. 2020).

[44] *What Is Git | Explore A Distributed Version Control Tool*. Edureka. 16th Nov. 2016. URL: https://www.edureka.co/blog/what-is-git/ (visited on 1st Dec. 2020).

[45] *Using Perforce in a Complex Jenkins Pipeline*. URL: https://technology.riotgames.com/news/using-perforce-complex-jenkins-pipeline (visited on 1st Dec. 2020).

[46] *Stack Overflow Developer Survey 2018*. Stack Overflow. URL: https://insights.stackoverflow.com/survey/2018/?utm_source=so-owned&utm_medium=social&utm_campaign=dev-survey-2018&utm_content=social-share (visited on 30th Nov. 2020).

[47] *Git Large File Storage*. Git Large File Storage. URL: https://git-lfs.github.com/ (visited on 25th Feb. 2021).

[48] *Git/Git*. GitHub. URL: https://github.com/git/git/ (visited on 25th Feb. 2021).

[49] *Perforce Software - Why Rated 6.6/10? (Sep 2020)*. URL: https://www.itqlick.com/perforce-software/pricing (visited on 26th Feb. 2021).

[50] *Migrating from Perforce Helix | GitLab*. URL: https://docs.gitlab.com/ee/user/project/import/perforce.html (visited on 26th Feb. 2021).

[51] *Perforce Git Tools | Helix4Git + Helix TeamHub | Perforce*. URL: https://www.perforce.com/products/helix4git (visited on 26th Feb. 2021).

[52] Christopher Alexander, Sara Ishikawa and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press, 1977. 1171 pp. ISBN: 978-0-19-501919-3.

[53] *Flyweight*. URL: https://refactoring.guru/design-patterns/flyweight (visited on 8th Oct. 2020).

[54] *Unreal Engine Docs*. URL: https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Objects/index.html (visited on 12th Nov. 2020).

[55] Dusty Phillips, Chetan Giridhar and Sakis Kasampalis. *Python: Master the Art of Design Patterns*. 1st ed. Packt Publishing, 2016. ISBN: 978-1-78712-518-6.

[56] *Automation Spec*. URL: https://docs.unrealengine.com/en-US/TestingAndOptimization/Automation/AutomationSpec/index.html (visited on 20th Feb. 2021).

[57] *The Code Affirms an Obligation of Computing Professionals to Use Their Skills for the Benefit of Society*. URL: https://www.acm.org/code-of-ethics (visited on 19th Feb. 2021).

[58] *Unreal Engine | Publishing EULA*. Unreal Engine. URL: https://www.unrealengine.com/en-US/eula/publishing (visited on 20th Feb. 2021).

[59] *Game Accessibility Guidelines | A Straightforward Reference for Inclusive Game Design*. URL: http://gameaccessibilityguidelines.com/ (visited on 20th Feb. 2021).

[60] *PopCap Games Research*. GamesIndustry.biz. URL: https://www.gamesindustry.biz/articles/popcap-games-research-publisher-s-latest-survey-says-that-casual-games-are-big-with-disabled-people (visited on 20th Feb. 2021).

[61] ScreenSkills. *Games Publisher*. ScreenSkills. URL: https://www.screenskills.com/careers/job-profiles/games/production/games-publisher/ (visited on 20th Feb. 2021).

[62] *LaTeX - A Document Preparation System*. URL: https://www.latex-project.org/ (visited on 26th Feb. 2021).

[63] *Hello World · GitHub Guides*. URL: https://guides.github.com/activities/hello-world/ (visited on 26th Feb. 2021).

[64] *Trello Limits Teams on Free Tier to 10 Boards, Rolls out Enterprise Automations and Admin Controls*. VentureBeat. 19th Mar. 2019. URL: https://venturebeat.com/2019/03/19/trello-limits-free-users-to-10-boards-rolls-out-enterprise-automations-and-admin-controls/ (visited on 27th Feb. 2021).

# Glossary

**AActor** the base class for an Object that can be placed or spawned in a level. 37, 41, 43

**Behaviour Tree** a mathematical model of plan execution used in computer science, robotics, control systems and video games. They describe switchings between a finite set of tasks in a modular fashion. 15

**behaviour-driven development** an Agile software development process that encourages collaboration among developers, QA and non-technical or business participants in a software project. It encourages teams to use conversation and concrete examples to formalize a shared understanding of how the application should behave. 38

**cooking** the process of converting content from the internal format to a platform-specific format. 10

**DLL** a library that contains code and data that can be used by more than one program at the same time. 17

**engine-heavy** has a lot of game complexity stored within highly performant languages such as C++, where programmers can manage the complexity of objects and their interactions, and game designers get a playground of robust toys to configure to their whims. 9

**engine-light** stores only the small central core within performant languages, and aspects such as gameplay systems are implemented in scripting languages such as Lua. The game's complexity can be tamed better with higher-level programming constructs, while not sacrificing too much performance in the process. 9

**forward renderer** renders each object in one or more passes, depending on lights that affect the object. 11

**iterative development** is the process by which software is repeatedly proposed, prototyped, playtested and reevaluated before working product release. 16

**mono repository** a single repository that stores all of your code and assets for every project. 27, 28

**profiling** a form of dynamic program analysis that measures, for example, space or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. 16

**software as a service** a software licensing and delivery model in which software is licensed on a subscription basis and is centrally hosted. 21, 24

**unit test** the smallest piece of code that can be logically isolated in a system. 22

# Appendix A: **Project Diary**

This section includes a comprehensive diary of the progress of the project.

### 23rd June 2020

I had a meeting with my Project Supervisor to kick start the project, as well as to get a good understanding of what is expected within the project. This included expectations of the project, as well as formed the basis for the research to be completed over the Summer.

### 30th September 2020

I had a meeting with my Project Supervisor to discuss the project plan and what best to tackle during the first sprint; the plan was then modified to have a clearer core set of goals to complete. The plan for the next two weeks is to start the report on Flyweight, as well as start implementing the Flyweight pattern using the Unreal Engine.

### 14th October 2020

During the meeting with my Project Supervisor, it was decided that I would create a small game of tag based on Paul Gestwicki's Unreal Engine 4 for Computer Scientists' series. This would form the basis of a chapter detailing the difference between different game engines. More context on academic writing was also discussed to help improve my writing and how to layout the report.

### 20th October 2020

I created a Unity student account in preparation for creating a small game in Unity in the future. The Blueprint's and C++ Unreal projects were created, as well as setting up git integration to store the project on the main repository

### 2nd November 2020

The meeting with my Project Supervisor helped to reevaluate the plan for the following sprint. By completing the Blueprint Game of Tag in Blueprints, I will be able to send over a playable executable to ensure that this executable works on the supervisor's computer. I will also be writing reports on the implementation of Blueprints in the Unreal Engine.

### 11th November 2020

I completed the Game of Tag in Blueprints game and passed it onto my supervisor. The Flyweight report is also mostly completed; I just need the last proofreading before the report is to be sent to my supervisor.

### 15th November 2020

The report on Flyweight is completed, as well as the executable being packed, so both have been sent to the supervisor for advice and feedback. I am now preparing to implement Projectiles in Unreal, as well as the C++ implementation of the game of tag.

### 25th November 2020

I discussed the report's nature with my supervisor, especially sections that lacked clarity in the project specification (namely the introductory section and the project specification). It was also discussed to add small code snippets to the Game Design Patterns section, but I believe this will be out of scope for the Interim Report - I am planning to add this before the final report is due.

### 2nd December 2020

I am behind compared to my plan but will use the time between now and the deadline to determine the most important points that need to be completed to complete as much work as possible before the deadline — I am planning to polish up sections that have not been completed, adding a plan for next term, and a summary of completed work to the interim report.

### 7th December 2020

Executables are packed, and the interim report is ready to be submitted. Did not include proof-of-concept relating to animations, as I need to discuss the difference between state machines and state pattern with my project supervisor. I also did not manage to get the game of tag in C++ completed due to issues with getting delegates (Unreal's event system) working using C++.

### 16th January 2021

I have added AI proof of concept to the project, which currently is a simple AI that patrols an area, and then it moves towards a player if it detects it using the Unreal AI Perception module. There are no AI attacks as these shall be added in later. I have also moved the game world to a different level so that proof of concept programs, c++ proof of concepts, and the actual game are different parts of the project. Additionally, the Unreal project has been updated to 4.26.0 with no conflicting issues.

### 19h January 2021

Unity base character added. I have little knowledge of Unity, so I need to go follow a few tutorial guides on how to get set up with the engine and with C# as I also have no experience of this. I have also spent time going through a conversion course on Blueprints to C++ that I have found due to difficulties converting the BP Game of Tag to C++. The base game is completed; just the AI needs implementing in C++, though I am unsure how practical this may be.

### 22nd January 2021

Finally finished the C++ game of tag. The AI components have been implemented in C++, but I decided to leave the behaviour tree logic in the Blueprint BT class due to time constraints and overall lack of understanding on how to set up a BT in C++ (C++ BTs are not the intended way to do it, even if possible). I also packaged the game and then realised that some of the code that worked in the editor did not work when the game was packaged, so I needed to debug the game and find out the error before realising it was due to accessing classes in C++ (again, usually would be done in Blueprints). This is now fixed, and as such, I have a packaged Game of Tag for C++ – Menu system included! I now plan to start working on Unity courses (as I have no knowledge of Unity at this point and need to go back to basics), as well as working on the write up of C++ vs Blueprints implementations. Once these are both done, I will work on the Game of Tag in Unity and then start implementing more on my game — extra note: Currently unsure if I will implement a game in PyGame. PyGame itself does not support 3D. It may be best to work on other parts of the project and come back at the end if I have time, as I believe this will be a bigger undertaking than the previous Unreal and upcoming Unity implementation.

### 25th January 2021

I decided to stop putting off implementing automated testing and spent the weekend working on researching frameworks for testing methods. There are a few to choose from, but none of these methods had reasonable documentation. Because of this, I ended up going through blog posts to try and understand the best way to implement tests, and these were also all over the place. Some used built-in Unreal frameworks but never went into depth with what they could do. Some people decided to make their own approaches, such as integrating Google Test (C++ testing framework) into Unreal, but this required building the Engine from source and playing around with config files (which I was reluctant to do). I decided to implement the standard Automation System in Unreal for simplicities sake but realised I fell into a few pitfalls. There was no BeforeEach method etc. Lots of seemingly basic functionality was missing. It was only, later on, I found the Automatic spec, which used a Behavioural-Driven development approach, and had reasonable documentation, that I decided to go with this method. At least I now know all the possibilities when it comes to different ways to test Unreal!

### 27th January 2021

I am currently working my way through a Udemy course on Unity (by the same people who did the Unreal course that I used originally), which is about 32 hours worth of videos. I will work through this to get a good grasp of Unity before I do the Unity Game of Tag so that I also have a better understanding of how Unity works and so that I can discuss Unreal vs Unity in my report. I have noticed the deadline for the final draft report is on the 19th of February, so I will likely start to work more towards this rather than implementing more on the final game for now.

### 16th February 2021

Professional issues section draft complete, ready for the final report. This section needs looking over by my supervisor, as I likely need clarification on what it means by being *reflective and thoughtful*. Due to unforeseen circumstances, my project is now behind schedule, and I will need to work on catching back up to where I should be for the project. Much of the groundwork for the final report is in place and just needs to be completed. When this is completed, the only work that needs to be completed is the Unity game, the discussion on it, as well as the

implementation in Unreal.

### 11th February 2021

The Game of Tag in Unity has now been completed. I am currently working on my Survey of Related Literature and bringing the report to a close. There are a few parts that need overviewing, but overall, the report is in a good place. Then from here, I will complete the game ready to hand the final project in by the end of the month.

### 11th March 2021

The Game of Tag in Unity has now been completed. I am currently working on my Survey of Related Literature and bringing the report to a close. There are a few parts that need overviewing, but overall, the report is in a good place. Then from here, I will complete the game ready to hand the final project in by the end of the month.

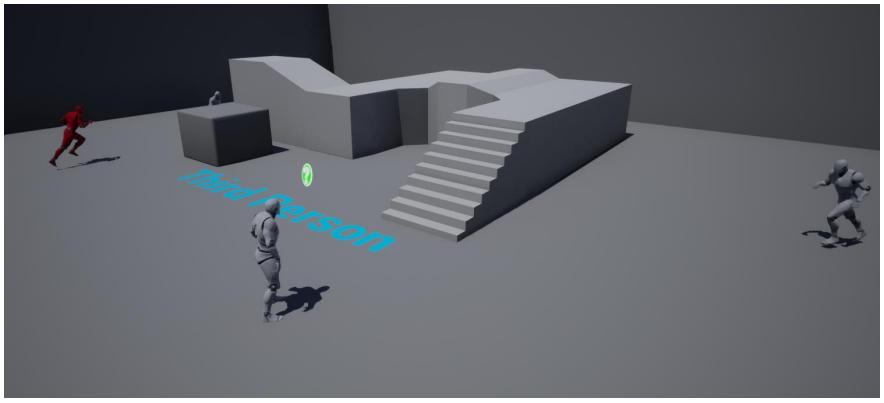
### 18th March 2021

I have now completed my final draft of the report, ready to get an overview from my supervisor before the final deadline. Some aspects from the initial plan are not present, such as the report discussing development using PyGame. This has been removed due to, in part, aiming to complete too much; therefore, the scope of the project has had to be changed to fit within the remaining time that I have left. The final game in the Unreal Engine is now my priority; therefore, I will ask my supervisor if any truly key elements are missing from the report; else, I feel like the report is in a good place. The main section missing from the report is the section on game development, which discusses the overall sections of the game being developed, rather than just the proof of concepts that I have developed. This will be added within the next week as I bring the final game to a close. I also need to package the Game of Tag for Unity and the Game of Tag Blueprints version. Overall I am pretty confident with how the overall project has gone, even if I was not able to complete all the aspects that I initially planned to do.

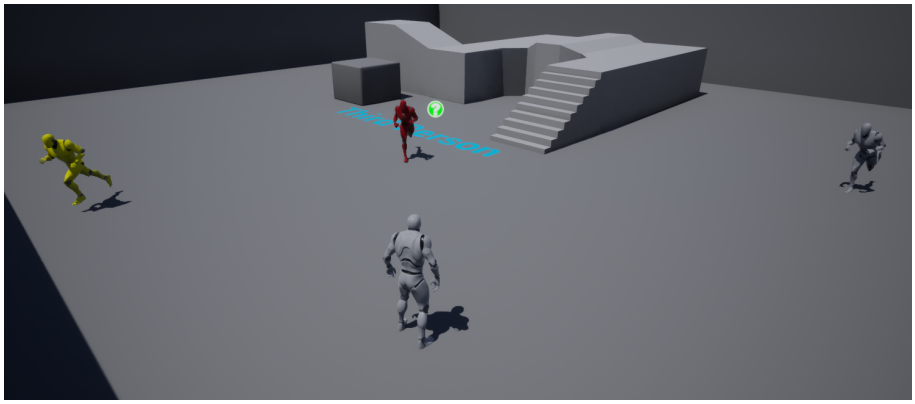
### 25th March 2021

The project has finally, for the most part, come together. There are aspects of the project that I would have liked to have explored in more detail, but overall I am happy with my project and the knowledge that I have gained while doing this project. The final game has come to fruition, and aside from a few bugs in the game that I plan to remove with the remaining time that I have, it is in a good place.

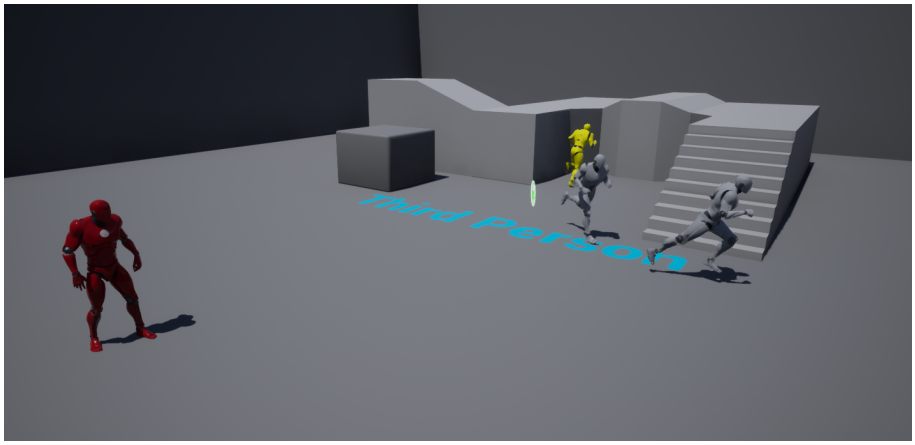
# Appendix B: **Game of Tag Overview**



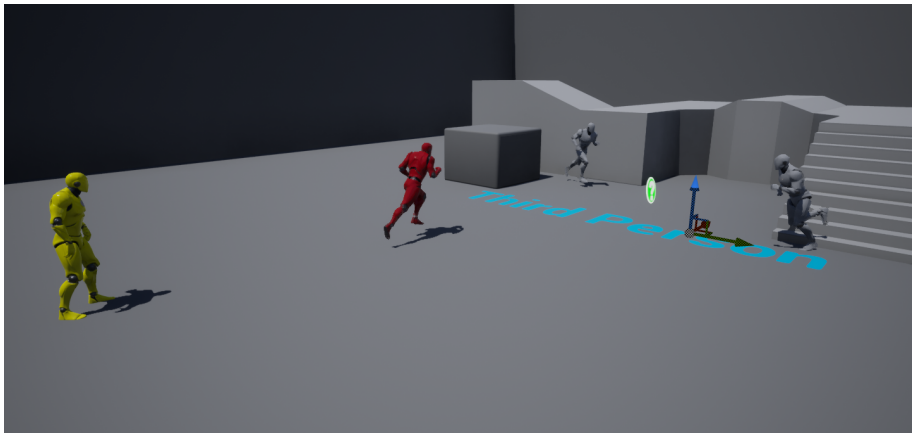
The character on the left is set to a 'tagged' state, and then their colour is set to red to indicate that they are tagged. They then start to move towards the closest character to make them the tagged character.



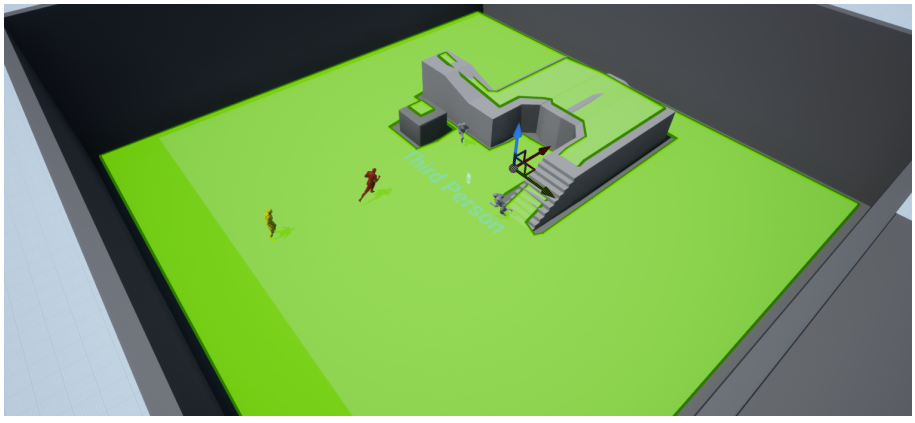
The tagged character is now set to yellow to represent that they were the previous tagged player and that they can no longer be tagged (while they are the previous tagged player). The tagged character now comes towards the next closest actor (in this case, the player).




Now the user is the tagged player, and the AI move away from the tagged user.




Once the player has tagged a character, they are no longer allowed to be tagged, and until the AI tags another AI character, the user can move away from the tagged characters.




The green section indicates the navigation mesh (navmesh) that are built around the static meshes of the game world. This navmesh indicates the possible areas where the AI is allowed to move.

# Appendix C: **Running a Game of Tag**

The following section is a user manual on how to run the game of tag. All versions of the game should run the same, using similar menus and identical keybindings.

## C.1 Download

To download the game of tag version required, check out the GitHub repository. From here, all the executables can be found under the Releases section, with corresponding names for their versions. For example, `Unreal Engine Game of Tag` made in C++ (packed for Linux) is named *Game of Tag | Unreal Engine | C++ | Linux*.

From here, the zip file should be downloaded and extracted. In windows, run the executable. In Linux, run the shell script.

## C.2 Playing the Game

The game of tag is a proof-of-concept for building games in different engines, and as such, the gameplay mechanics are simple and easy to follow.

There is a simple menu system at the start of the game, which has three buttons:

- **Play**: starts the game of tag.
- **How To Play**: shows a menu with all the instructions for the game.
- **Quit**: used to close the executable game file.

Press ESC anytime in the game to return to the main menu. To play: run around the world using the WASD keys on your keyboard, and control your camera using your mouse. If your character is RED, then you are 'tagged'; you must run into another play to tag them (as long as that player is not YELLOW). If you are yellow, you are the previous 'tagged' player and cannot be tagged; you can rest easy! If you're a GREY character, run away from the RED character as they will try to tag you.

# Appendix D: **Running Pursual**

This section will include an overview of how to download the game, as well as a user manual on how to play the game. This same "how to play" can be found in the main menu of the game.

## D.1 Download

To download Pursual, check out the GitHub repository. From here, all the executables can be found under the Releases section, with the corresponding names for their versions. For example, the `Final Game (Pursual)` (packed for Windows) is named *Final Game (Pursual) | Windows*.

From here, the zip file should be downloaded and extracted. In windows, run the executable. In Linux, run the shell script.

## D.2 Playing the Game

Pursual is the final game deliverable.

There is a simple menu system at the start of the game, which has three buttons:

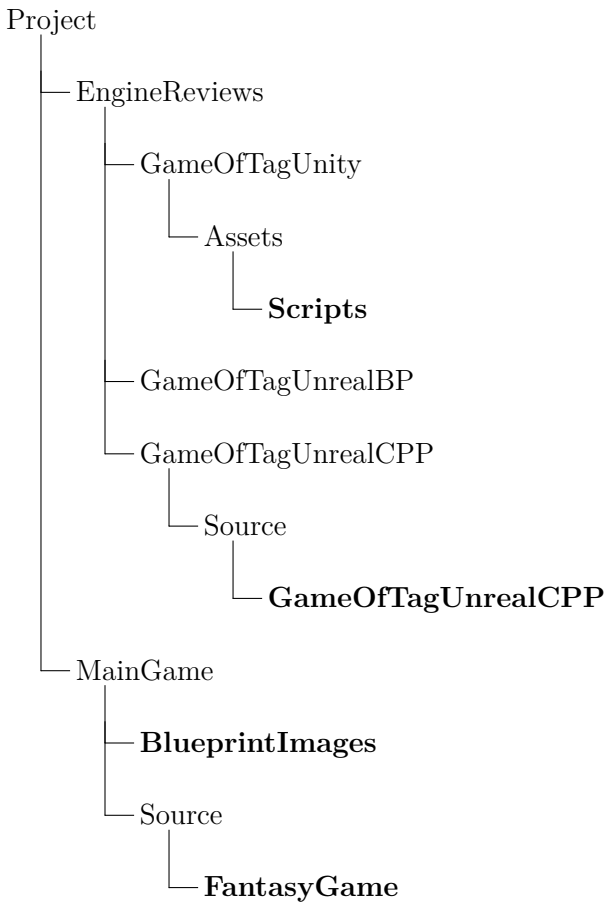
- **Play**: starts the game of tag.
- **How To Play**: shows a menu with all the instructions for the game.
- **Quit**: used to close the executable game file.

The main keys that the user is required to know are as follows:

- **WASD**: these keys are used to control the movement of the player. W to move forward. S to move backwards. A to move to the left. D to move to the right.
- **Left Shift**: holding down will allow the player to run.
- **E**: used for interaction. An 'interact' UI is shown when the user can interact with a game object.
- **F**: the main 'fireball' attack of the player. This will shoot a projectile in the camera direction of the player.
- **Q**: deploys a shield for the player. While the shield is active, the player cannot move or attack, but enemy projectiles will bounce off the shield.
- **ESC**: press anytime in the game to return to the main menu. This will reset the progress of the game.

# Appendix E: **File Structure**

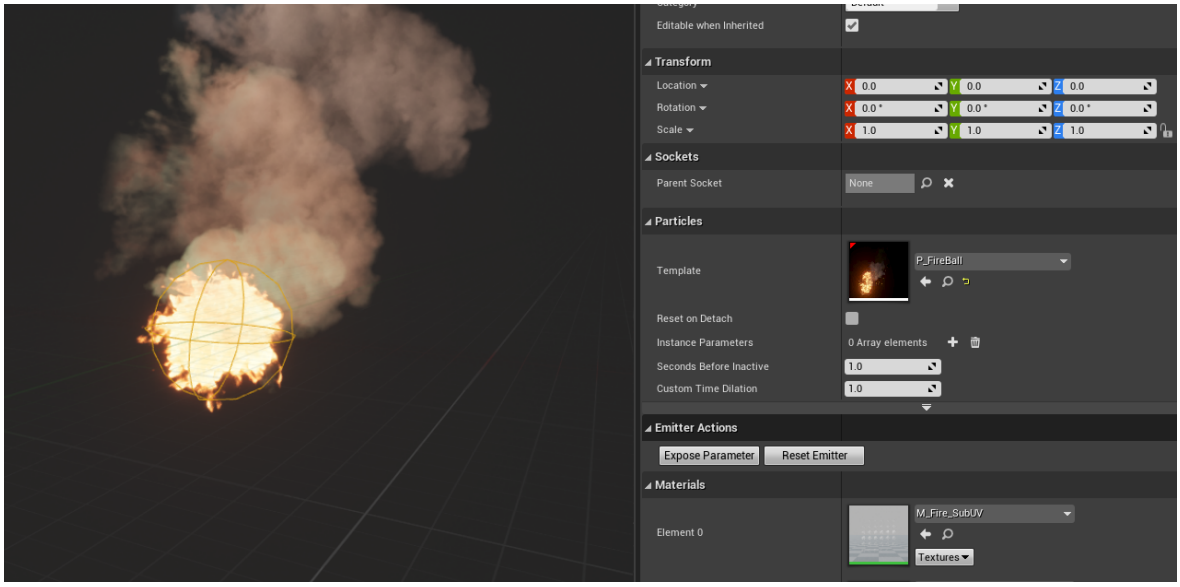
This section gives a clear overview of the file structure of the project. This section also provides the location of folders of notable mention in bold.



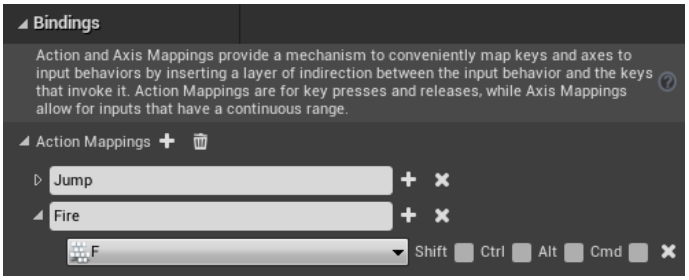
# Appendix F: **Proof of Concept - Fireball Projectile**

The following section is the Blueprint implementation of the Fireball Projectile. This includes ensuring that the player cannot use the projectile if they are already in the animation, as well as spawning a particle at the hand of the player to make it seem like the fireball is *charging up*, rather than just appearing.

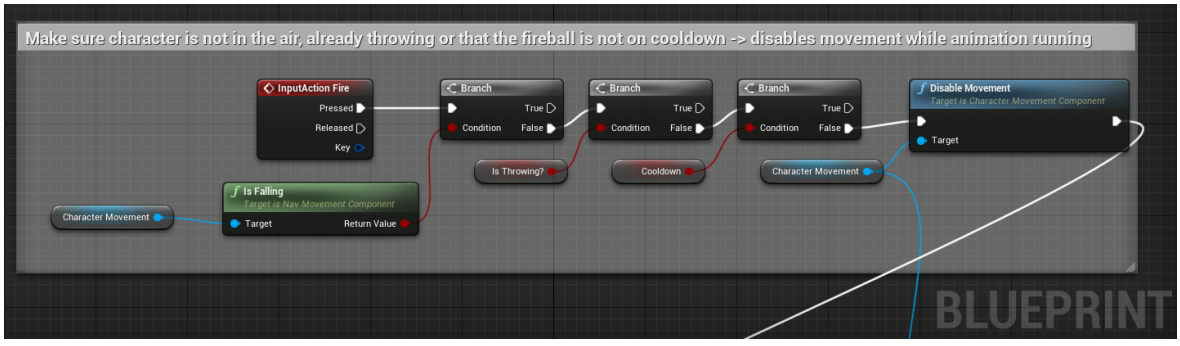
All Blueprints are annotated using the Blueprint `Create Comment from Selection` method.

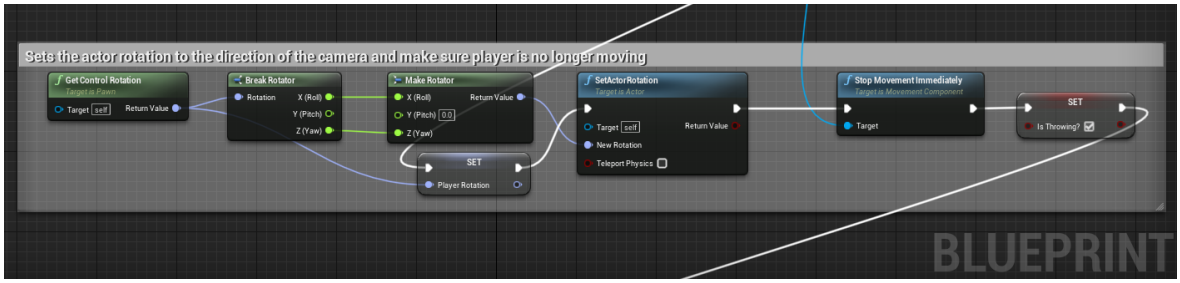


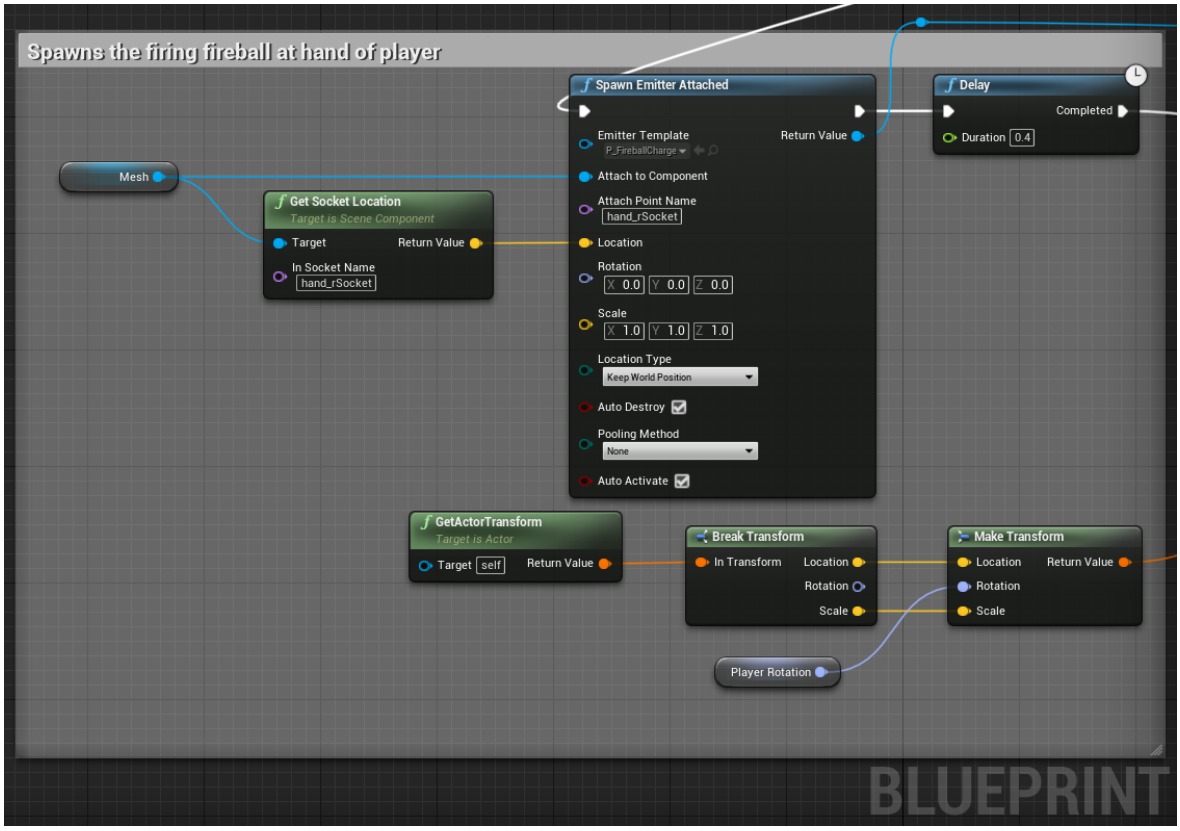

It is used to set the particle animation of the fireball in the Unreal Engine Editor.

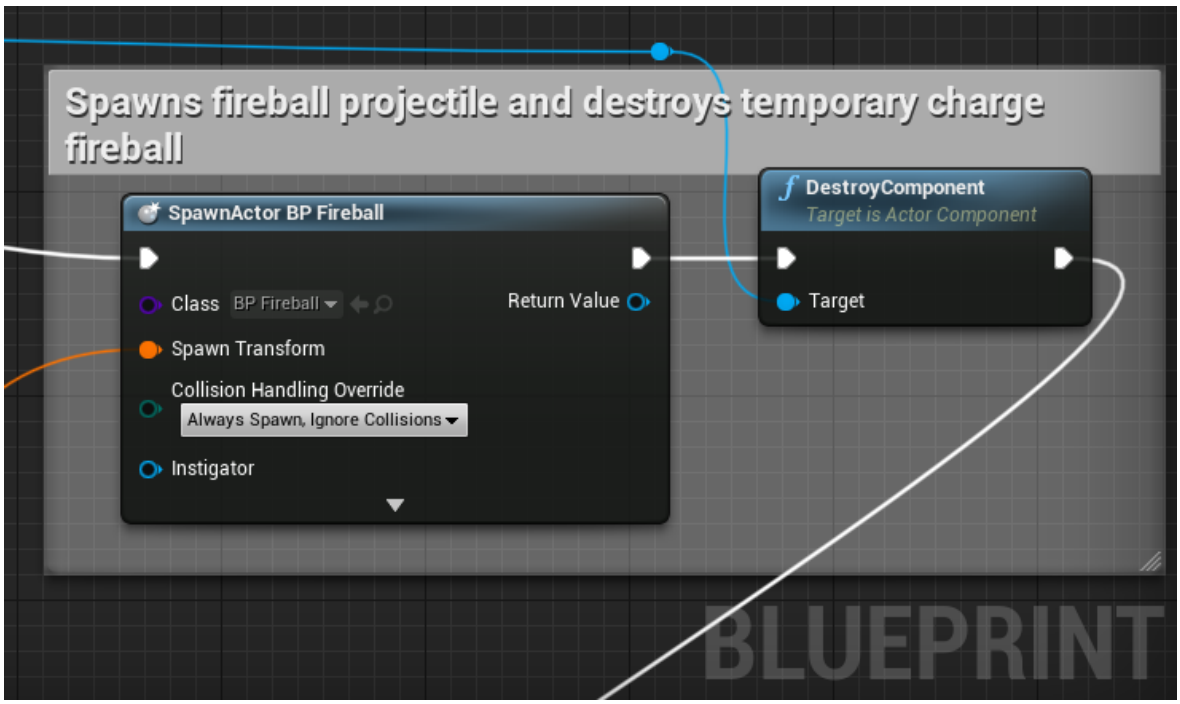


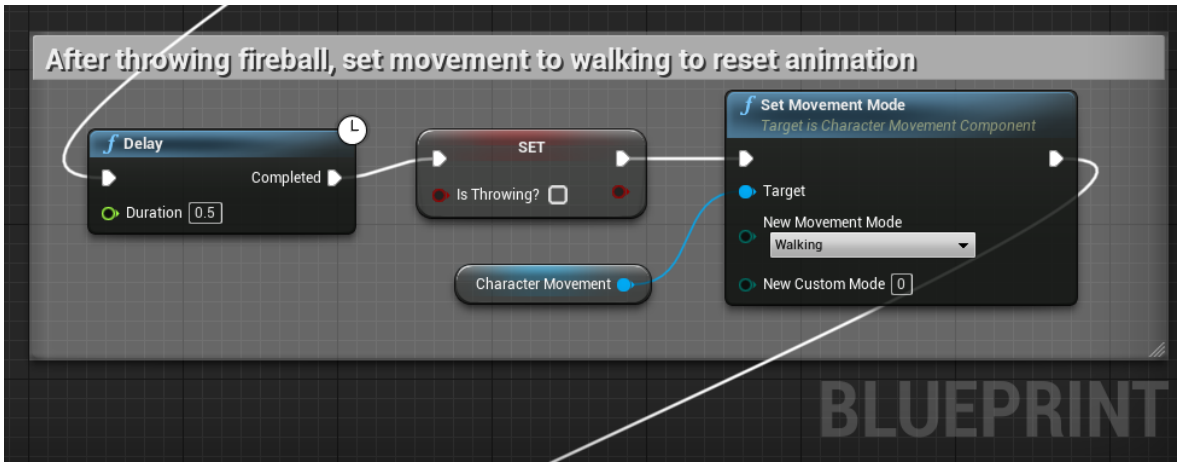

The 'Fire' action mapping (in the Unreal Project Settings) is used to bind the 'F' key to start the Fire Projectile sequence.

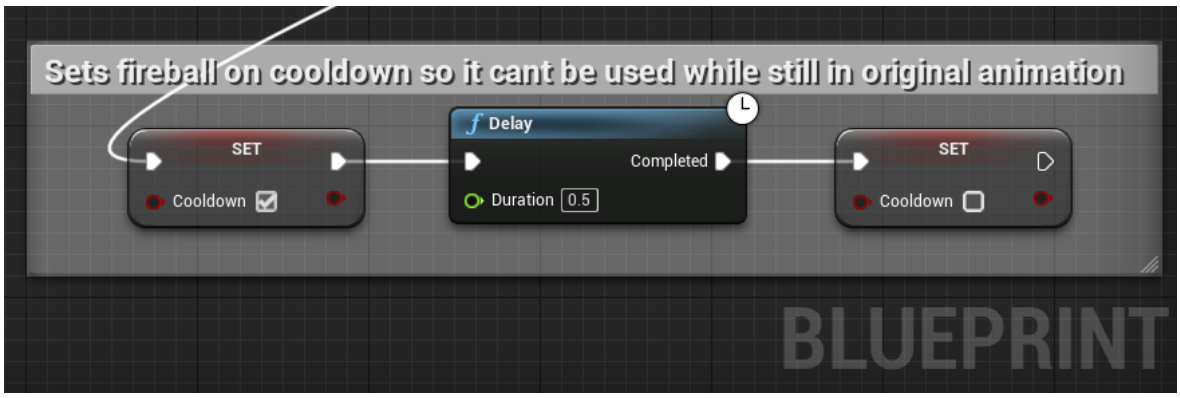



Sets the actor rotation to the direction of the camera and make sure player is no longer moving
Get Control Rotation
Break Rotator
Make Rotator
SetActorRotation
Stop Movement Immediately
SET
BLUEPRINT

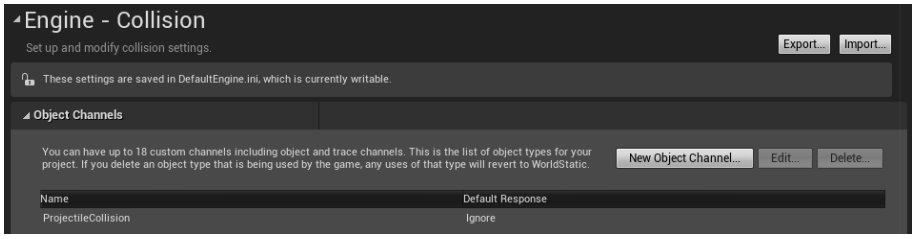

Spawns the firing fireball at hand of player
Spawn Emitter Attached
Delay
Mesh
Get Socket Location
Target is Scene Component
Emitter Template
Return Value
Attach to Component
Attach Point Name
hand_rSocket
Location
Rotation
Scale
Location Type
Keep World Position
Auto Destroy
Pooling Method
None
Auto Activate
Completed
Duration
GetActorTransform
Target is Actor
Break Transform
In Transform
Make Transform
Player Rotation
BLUEPRINT


Spawns fireball projectile and destroys temporary charge fireball
SpawnActor BP Fireball
Class
BP Fireball
Return Value
Spawn Transform
Collision Handling Override
Always Spawn, Ignore Collisions
Instigator
DestroyComponent
Target is Actor Component
Target
BLUEPRINT

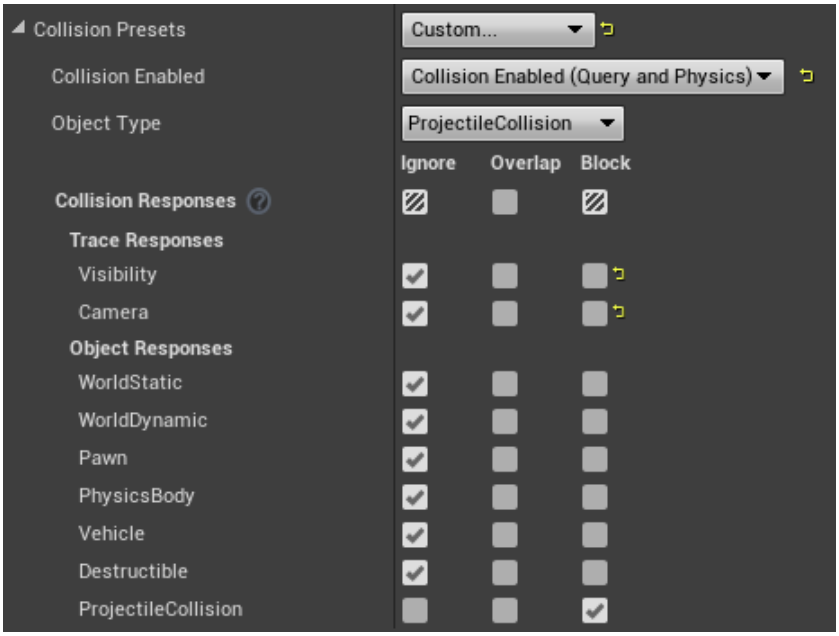

After throwing fireball, set movement to walking to reset animation
Delay
Completed
Duration 0.5
SET
Is Throwing?
Character Movement
Set Movement Mode
Target is Character Movement Component
Target
New Movement Mode
Walking
New Custom Mode 0
BLUEPRINT


Sets fireball on cooldown so it cant be used while still in original animation
SET
Cooldown
Delay
Completed
Duration 0.5
SET
Cooldown
BLUEPRINT

# Appendix G: **Proof of Concept - Reflecting Projectiles**




The collision objects can be set in the `Project Settings → Collision` window.




The object has an Object Type, which is the type of collision that the object is. This allows for other objects to interact correctly.

# Appendix H: **Tooling**

This section gives an overview of all the tools used to prepare this report.

## H.1 Report Writing

**LaTeX** is a high-quality typesetting system; it includes features designed for the production of technical and scientific documentation. LaTeX is the de facto standard for the communication and publication of scientific documents[62]. LaTeX has been used to typeset this report, as:

- it ensures that the entire document has been kept to the same formatting standard
- it provides consistency in the report
- it generates a pdf copy of the document (as it compiles to pdf by default) for the final hand-in

## H.2 Version Control

**Git** with **GitHub** is being used as the version control system for this project. A description of Git can be seen in *section 3.3.3*. GitHub is a code hosting platform for version control and collaboration. It lets developers work together on projects from anywhere[63]. Git is being used due to project requirements, as it is compulsory that either a Subversion or GitHub departmental repository is used. GitHub was chosen due to prior existing knowledge of the platform, as well as the potential to integrate Git LFS (see *section 3.3.3*).

The Git repository is managed through a combination of **GitKraken** along with using the **Git CLI** (command-line interface) from the terminal. GitKraken is a GUI client that allows a user to manage a Git repository through a UI tool rather than a terminal. One of the main advantages of using a Git GUI is its simplicity - a user does not need to remember git commands and instead can use buttons to do the same job. Another advantage is the ability to visualise branches and the commit history and fix merge conflicts all from within the same application.

Additionally, the Git CLI is used as GitKraken needed access to the organisation that the repository was present in order to do specific commands - most notably the 'push' and 'pull' commands, whereas the CLI did not. The CLI was used to issues these commands.

## H.3 Game Engines

**The Unreal Engine** is a game engine developed by Epic Games. It is one of the world's most powerful game engines and is becoming increasingly popular for architecture, Film and Television, and Training and Simulations. It is also a state-of-the-art real-time engine and editor that features photorealistic rendering, dynamic physics and effects, lifelike animation, robust data translation, and much more — on an open, extensible platform that will not tie a developer down. For more information, see *Section 2.2.2*.

The Unreal Engine was chosen due to its prominence in recent and upcoming AAA games, as well as to learn about a AAA game development engine that would likely be used if choosing to pursue a career in game development, especially with the highly anticipated release of the Unreal Engine 5 coming late 2021.

For a comprehensive guide on how to download and install the Unreal Engine, please see here.

**Unity** is a game engine developed by Unity Technologies (see *section 2.2.3*). Unity was chosen as the main comparison to Unreal Engine due to it being the most popular game engine for indie developers and mobile games.

For a comprehensive guide on downloading and installing Unity, please see here.

## H.4 Integrated Development Environment

'**Rider** for Unreal Engine' is an IDE in Public Preview developed by JetBrains for the Unreal Engine. Rider was chosen due to familiarity with their other IDEs (namely IntelliJ and PyCharm) and because of positive reviews found on the web. It also has native C++ support using ReSharper and built-in code completion, making development in the Unreal Engine simpler than in other editors.

Visual Studio Code (**VSCode**) has been used due to its lightweight nature, namely as a LaTeX editor. Using LaTeX in a docker container, alongside the 'LaTeX Workshop' VSCode plugin (developed by James Yu), the report could be written without any hassle. VSCode also has a 'Grammarly (unofficial)' plugin developed by Rahul Kadyan that allowed for spelling and grammar checking while the report was written.

## H.5 Project Management

**Trello** is a web-based, Kanban-style, list-making application[64] that helped to organise the project. By sorting each part of the project into 'To Do', 'In Progress', and 'Done' sections, it became easy to see what needed to improve upon and what features still were required. Labels such as 'Report' and 'Programming' were also able to be added, making it easy to see the different parts of the project at a glance.

# Appendix I: **Acknowledgements**

"Building your own Basic Behavior tree in Unity [Tutorial]" (https://hub.packtpub.com/building-your-own-basic-behavior-tree-tutorial/) by Natasha Mathur (Packt Publishing)

"Exclamation Of Mission" (https://skfb.ly/CzoO) by Huargenn is licensed under Creative Commons Attribution (http://creativecommons.org/licenses/by/4.0/).

"Medieval Kingdom" (https://www.unrealengine.com/marketplace/en-US/product/medieval-kingdom) by beffio

"Wood Road Sign" (https://www.turbosquid.com/3d-models/3d-model-wood-road-sign/538834) by Mantifang