DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2019

Software Reuse in Game Development: Creating Building Blocks for Prototyping

RICKARD BJÖRKLUND

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Software Reuse in Game Development: Creating Building Blocks for Prototyping

RICKARD BJÖRKLUND

Master in Computer Science
Date: June 5, 2019
Supervisor: Cyrille Artho
Examiner: Martin Monperrus
School of Electrical Engineering and Computer Science
Host company: EA DICE AB
Swedish title: Återanvändning i spelutveckling: byggstenar för prototyping

Abstract

As games and the technologies used by them have become more advanced, the cost of producing games have increased. Today, the latest AAA titles are the results of hundreds or as many as thousands of people working full-time for years, and even developing a prototype requires a large investment. This project sets out to reduce that cost by looking at how reusable building blocks can be used in the prototyping process.

During the project, seven interviews with game designers were conducted. The interviews found that building character controllers for the player was the most common activity and one of the more difficult tasks when prototyping a new game. As a result, a tool for creating character controllers was made.

The tool builds the character controllers to work as state machines where actions in a state and transitions between states are editable through a visual programming language. The visual programming language uses nodes. These nodes work as reusable building blocks.

The tool was evaluated by six game designers and four programmers who all thought the tool used a good approach for building and prototyping character controllers. The evaluation also showed that the building blocks, in the form of nodes in the tool, should be functionally small and general, like nodes for applying forces and accessing character data.

Sammanfattning

Allt eftersom spel och de tekniker som används i spelutveckling ökar i komplexitet, desto högre blir kostnaderna för att utveckla spel. Idag krävs hundratals, ibland till och med tusentals heltidsarbetande i mångåriga projekt för att utveckla AAA-spel, och även utvecklandet av en prototyp innebär stora investeringar. Detta projekt ämnar att minska den kostnaden genom att undersöka hur återanvändningsbara byggstenar kan användas i prototyputveckling.

Under projektets gång, utfördes sju intervjuer med speldesigners. Dessa intervjuer visade att utveckling av karaktärkontroller var en av de vanligaste och svåraste aktiviteterna i utveckling av prototyper. Som en följd av detta skapades ett verktyg för att utveckla karaktärkontroller.

De karaktärkontroller som verktyget bygger fungerar som tillståndsmaskiner där händelser i ett tillstånd och övergångar mellan tillstånd kontrolleras med ett visuellt programmeringsspråk. Det visuella programmeringsspråket använder noder och det är dessa noder som utgör de återanvändningsbara byggstenarna.

Verktyget utvärderades av sex speldesigners och fyra spelprogrammerare som alla tyckte att verktyget var ett bra sätt att utveckla och prototypa karaktärkontroller på. Utvärderingen visade också att byggstenarna, i form av noder i verktyget, bör vara funktionellt små och generella, så som noder för att applicera krafter och modifiera karaktärsdata.

Contents

Chapter 1

Introduction

It has been almost 60 years since "Spacewar!", often accepted as the first digital video game, was developed by a team of computer scientists at MIT in 1962 [1, 2]. Since then digital and computational technologies have developed immensely, and so has the video game industry. What used to be a niche entertainment industry for and by the quite few people who owned a computer has grown to be a 138 billion dollar industry reaching over 2.3 billion people worldwide [3]. In 2018 the revenue from the global video game industry even exceeded the revenue of the global movie industry [3, 4].

As the gaming industry and the technologies used by it has developed, so have the actual games. What used to be blocky models stuttering around in low-resolution worlds can now be digital experiences difficult to distinguish from real life. But as the complexity of video games and the industry around them have grown, so has the effort and investment needed to produce a video game.

AAA games of today, like the latest instalment in a series like "Battlefield" or "Assassin's Creed", is the result of several hundred or even thousand people working full time for a duration of 1–3 years [5]. Unsurprisingly, this means that the cost of developing games like these are in the tens or hundreds of millions of dollars. Unfortunately, this also means that the investment needed to produce only a prototype for a game is large.

Today, it can take an investment of several months and millions of dollars to produce only a prototype for a new video game and it is in the interest of gamers and game developers alike to reduce this cost. Ideas for achieving this could, of course, come in many different forms. One could, for example, try to change the prototyping process or improve the usability of the tools most important in the prototyping process. However, this thesis investigates the

possibility of creating reusable building blocks for prototyping, what these blocks should be as well as how they can be used, designed and implemented.

1.1 Problem Definition

The main problem that will be examined in this thesis is:

> *How can a set of reusable building blocks that would allow game designers to quickly prototype gameplay ideas be composed?*

In finding an answer to this problem, the following research questions are to be examined:

1. *What is a set of building blocks that would allow game designers to quickly prototype gameplay ideas?*
2. *What is the closest equivalent to reusable building blocks in Unity?*
3. *Where in a game engine should reusable building blocks be implemented?*
4. *What is a suitable design that makes reusable building blocks easy to work with for game designers?*

1.2 Industry Cooperation

This thesis was conducted in cooperation with the international video game producer and publisher Electronic Arts (from here on referred to as EA). The thesis work was made partly on site and in close cooperation with EA DICE and the Frostbite division of EA in Stockholm, Sweden.

EA DICE is a Swedish game developer studio best known for its "Battlefield" series, but also for the "Mirror's Edge" and "Star Wars Battlefront" games [6]. The Frostbite division of EA is responsible for Frostbite, the game engine used in most of EA's newer titles [7]. Some examples of games made in the Frostbite engine are "Battlefield V", "Star Wars Battlefront II", "Need for Speed Payback" and "FIFA 19" [7].

A result of this cooperation is that interviews are sourced from people currently working at EA and that some implementation details are specific to the Frostbite engine. It should, however, be stated that EA is one of the largest video game producers in the world and that the Frostbite engine is a state-of-the-art game engine. The cooperation also allowed a proof-of-concept to be implemented in Frostbite and to be tested by experienced game designers.

1.3 Scope

This thesis project was limited to identifying a set of building blocks that would allow game designers to more quickly prototype gameplay ideas. The scope was limited to prototypes of games where the player traverses a 3D world since accommodating for simulation or strategy games would require more work than what is doable in a master thesis project.

The thesis also focuses on game development for PC and console. It should, however, be noted that it is games of this type (3D games for PC and console) that is generally the most expensive to develop and therefore poses the most interesting problem. Worth noticing is also that game development of other types of games for other consoles is not very different from those examined in this thesis project.

1.4 Ethical and Societal Aspects

There are many ethical and societal areas of discussion related to games. First and foremost is the alleged connection between violent games and violent behaviour in real life, an issue that has been both discussed and researched extensively [8]. Other areas for discussion is how games might have a negative impact on children and teen's performance in school, how games might have a negative impact on production and societal development and how games might provide the same type of escapism as drugs.

Although interesting topics, these are out of scope for this project. The overall desired effect of this project is to make it possible to prototype games faster, specifically with the help of reusable building blocks. These building blocks can be used for prototyping of both violent and non-violent games, and therefore the first area of discussion is not really relevant for this project. Since this project could make the production of games faster, it could be argued that the other ethical aspects are of relevance for this project. However, there are already that many games made, so if one truly wanted to escape reality into virtual worlds for the duration of a lifetime, this could be done today without having to wait for new titles.

Two areas of discussion that are within the scope of this project are the potential issue of eliminating job opportunities and the effect reusable building blocks can have on creativity. The first issue is that reusable building blocks could be a way to remove jobs by rationalizing. However, it has often been argued that whenever technology takes away jobs in one place, new positions

appear elsewhere [9]. It should also be noted that these building blocks are intended to be used primarily in prototyping, which is one of the less workforce-intensive stages of the development process.

The second area is more problematic. It could be argued that if, for example, a reusable character controller was made 15 years ago, we would not have games where the player have unique and interesting move sets like in "Mirror's Edge" or "Assassin's Creed" today, and that is a valid concern. It should be stated though that the intention with reusable building blocks is not to create a set of pieces from which only a finite set of games can be created. The intention is to have blocks of fundamental gameplay features available so that development time can be spent on unique features of the game instead.

1.5 Thesis Outline

The rest of the report is organized as follows. Chapter 2 presents the theory and knowledge necessary to follow the rest of the report. This includes an overview of game development, development tools and engine architecture. Chapter 3 exhibits previously published works related to the field and presents a state-of-the-art analysis into the Unity game engine. Chapter 4 describes and discusses the methodology used in the study.

Chapter 5 presents the results of the interview study. Chapter 6 describes the proof-of-concept that was created, how it was created and what motivates its creation. Chapter 7 presents the evaluation results of the tool. This is followed by Chapter 8, which discusses changes that can be made to the tool and analyses the results in relation to the problem statement. Chapter 9 summarises the project and concludes the report.

Chapter 2

Background

This chapter covers the fundamentals of the game development process and team as well as game engine architecture and tools. The chapter also presents some of the differences that exist between game development and traditional software development.

2.1 The Game Development Process

The game development process typically consists of five different phases: concept, prototyping, pre-production, production, and alpha-to-release [10]. Although this is a linear model sharing similarities with a waterfall model, it should be noted that game development rarely uses a fully linear process [11, 12, 13]. Instead, it is much more common that a hybrid process is used, meaning that the overarching processes can be seen as linear, but the everyday work is done according to an agile model [12, 13, 14].

2.1.1 The Concept Phase

The first step towards a finished game is the concept phase [10, 11]. During the concept phase, a small development team typically consisting of less than ten people, work for 1 or 2 months with core concepts and visual representations of the game [10]. It is usually also during this phase that initial financial agreements between developer and publisher are made; however, these are almost always subject to change [10].

2.1.2 The Prototyping Phase

After the concept phase comes the prototyping phase [10, 11]. This phase usually lasts between 3 and 6 months and allows for prototype development of many different aspects of the game [10]. These prototypes provide examples of features such as menus, physics and vehicle handling or could be technical demos such as grass rendering or destructible environments [10]. These prototypes are often interactive, but it is not unusual to use non-interactive prototypes either. By the end of the prototyping phase, prototypes and concepts are evaluated. If the project is approved, it moves into the pre-production phase [10, 11].

2.1.3 The Pre-production Phase

During the pre-production phase, fundamental game mechanics are developed and tested [10, 11]. This phase is typically 6 to 12 months long, and its purpose is to develop and try out ideas without having to consider the issue of final presentation quality [10]. During the pre-production phase game developers also try to identify risks and prove important aspects of the game concept, it is not unusual that concepts and ideas are changed during this phase.

2.1.4 The Production Phase

The main production phase is usually the longest and most expensive stage of the development cycle [10]. In this phase, the development team will be scaled up and work on characters, levels, menus and other components of the game [10, 11]. It is also often during this phase that a high-quality demonstration of the game is produced. This demonstration is also known as a “vertical slice” and typically represents 10–30 minutes of sample gameplay [10].

2.1.5 Alpha to Release

In the closing phase of the development cycle, the game progresses through a series of statuses indicating how close the development is to completion [10]. The first status to achieve is Alpha. Alpha status is usually achieved a few months after the main production phase and by this stage all content in the game is represented, but usually not of final quality [10].

The next status to achieve is Beta and the time from Alpha to Beta is typically around half a year [10]. By Beta all content and features should be finished and any work done by developers after this stage is contained to bug

fixing, tweaking and final adjustments [10]. After Beta the game reaches the shippable status [10, 11]. Depending on the publishing format the game is then either released directly on an online platform like Steam or Origin, or it goes through an approval process by a format holder. A format holder is a company like Microsoft, Sony or Nintendo, that has the right to review games before they are released on their platform.

2.2 The Game Development Team

Creating a game that competes on today's market requires a large team with many different skills [15, 16]. In this section, a description of the different roles within a game development team is given.

2.2.1 Game Designers

The responsibilities of game designers are level and world design, gameplay design, goals, objectives, quests and story in the game [1, 5, 17]. It is uncommon to see game development teams without at least one person doing the creative job of a game designer, and in most modern game development projects, there are several people with the role of game designer.

2.2.2 Engineers

Engineers have several different responsibilities, common tasks for engineers are gameplay programming and scripting, but there are also engineers that develop game engine components such as rendering, articial intelligence, audio and development tools [1, 14, 15]. Within larger game development teams, there also exist engineers who focus on theoretical work such as designing physics and collision systems [1].

Since the task of implementing a game designer's idea might be too large or difficult for a game designer it is common that engineers are assigned the work of realising a designer's vision [16]. However, since it might not be clear from the beginning what the designer's vision is and how it performs the context of a full game, this type of cooperation might cause irritation and inefficiency in the development process [14, 15, 16].

2.2.3 Artists

The artists' tasks are producing concept art, 3D models, textures, lighting, animation, voice/motion acting, sound design and music. Most game development projects have artists in the team, but in smaller projects, this might be outsourced or a decision is made to only use freely available assets [1, 18, 19].

2.2.4 Producers

In a game development environment, a producer is the highest ranking project manager. The producer's responsibilities are similar to any project manager's and he or she governs over the designers, engineers and artists, and make sure the project is moving in the right direction [1].

2.2.5 Publishers

The publisher takes care of marketing and any problems that might arise with the format holder [1]. In contrast to the other roles in a game development project, the publisher role is often filled by a company and not persons within the development team.

2.2.6 Others

The last category of roles in a game development project are roles that never actually touches the game during the development, but helps in other ways. Examples in this category are departments like HR and IT [1].

2.3 The Game Engine

Games are complex pieces of software and building them is no easy task [15, 16, 20]. To alleviate this difficulty, game engines are created. The purpose of a game engine is to provide a basis for games to be developed upon. This basis includes a lot of functionalities that are shared between games like rendering, networking and artificial intelligence [1, 20, 21, 22].

There is, however, a difficulty in describing a game engine. First of all, it is not really possible to define a clear line between engine and game. Secondly, adding to the complexity is the fact that different game engines have slightly different architectures [1, 20, 21, 22].

A game engine can generally be divided into two major components, the tool suite and the runtime component [1, 21, 22]. Of the two parts, the runtime

component can be considered more complex than the tool suite. Also, the architecture of the runtime component is usually not that different between different modern day game engines. Like other software systems, a the game engine is built in layers with higher layers being dependent on lower layers, but not vice versa [1, 20, 21]. An outline of the architecture can be seen in figure 2.1.

2.3.1 Target System

The three lowest layers, the hardware, drivers and OS layers, together represent the target system on which the game will run. Hardware can come in many different shapes and sizes like PC, console and handheld device, and in the case of PC, there is a nearly infinite number of combinations of components. The drivers are there to manage the hardware resources and provide a level of abstraction between the OS all the variants of hardware that exist [1, 20, 21]. On PC and modern consoles like the Xbox One and PS4, the OS is always running, managing the execution of multiple programs, meaning that the game will need to work together with the OS to keep running [1, 20].

2.3.2 Third-party SDKs

Third-party software development kits are pieces of software that are not developed by the game engine team. Since the game engine developers want to save time, game engines often facilitate a number of such kits. These kits usually provide interfaces for tools that handle data structures and algorithms, hardware interface for rendering as well as physics and character animation [1, 11, 21]. There is a number of available third-party kits, for physics and character animation Nvidia's PhysX and Havok's Havok are popular choices. For rendering, DirectX, Vulkan and OpenGL are common choices, but depending on the target system other SDKs might be chosen [1].

It is often discussed in the game developer community whether to use third-party SDKs or develop them in house. This is especially true for data structure and algorithm kits where for example STL and Boost are powerful libraries, but also argued to be unoptimized for game development [1].

2.3.3 Platform Independence Layer

Since third-party SDKs might behave differently or different SDKs have to be used all together depending on the target system, a platform independence

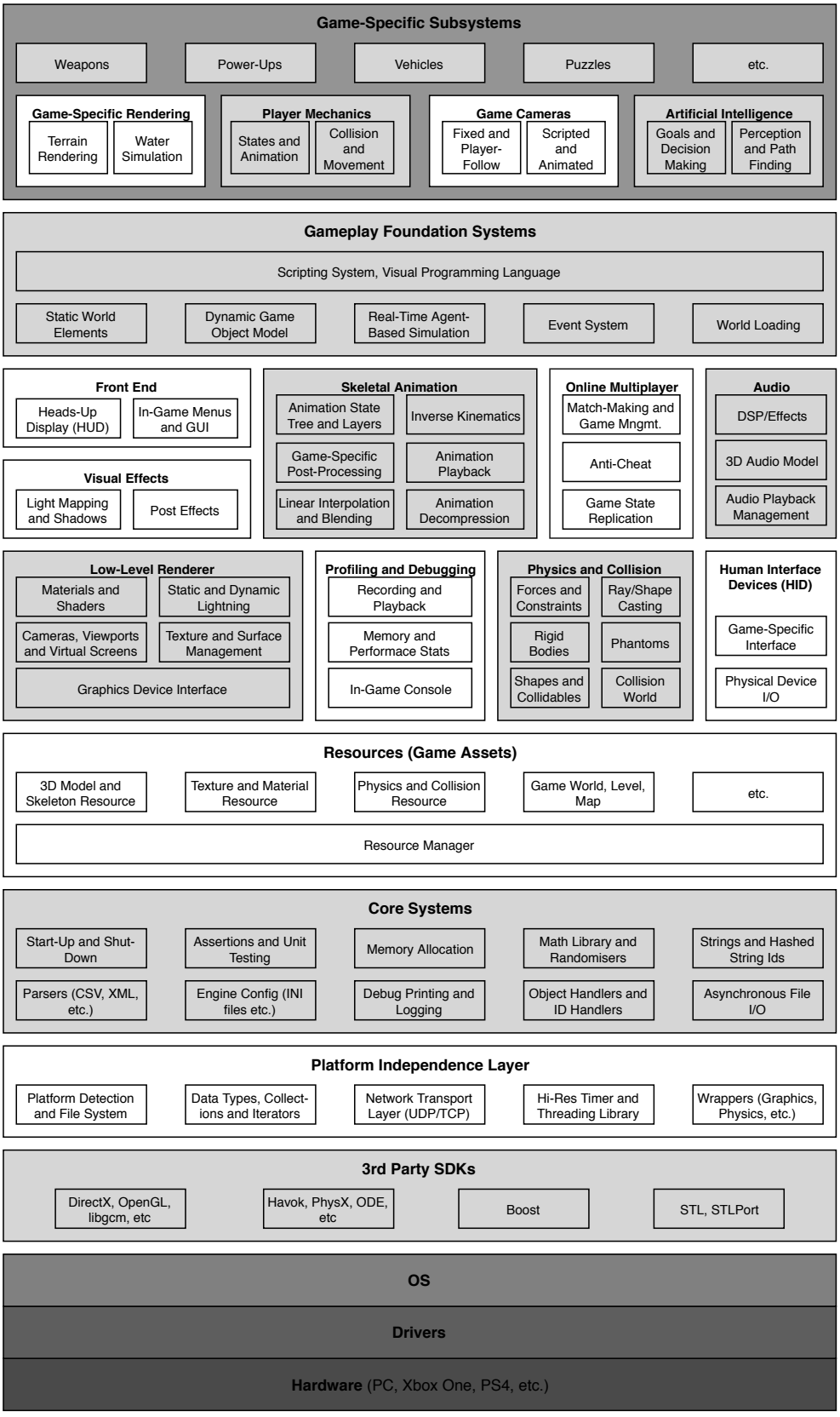

Figure 2.1: Game engine runtime architecture as described by Jason Gregory [1]

layer is implemented in all game engines that are intended to be used for development to different platforms [1, 20, 21]. By wrapping often-used C functions, OS and API calls the platform independence layer makes it easier for developers to produce a game for, for example, PC, Xbox One and PS4 at the same time.

2.3.4 Core Systems and Resource Management

Two components that are central to a game engine are the so-called core systems and resource management. The core systems are basically a set of software utilities like assertions, memory management, data structures and algorithms [1]. Although these utilities might be dependent on third-party SDKs beneath the platform independence layer, the implementations in this layer is most often specific to the game engine.

Since games include a large number of resources, like texture maps, 3D meshes, animations, audio clips, collision and physics data, game world layouts, etc., every engine needs a resource manager [1, 20]. This component is sometimes referred to as an asset or media manager, but regardless of its name the component is responsible for making sure that functions get the correct resource and that resources are only loaded into memory when needed.

All components layered above the core systems and the resource manager use these components in one way or another [1, 20].

2.3.5 Gameplay Foundation Systems

On top of the core systems and resource management lays a large number of very complex components. These components handle everything from rendering, physics, animation, visual effects, online multiplayer, audio and communication with input devices [1, 20, 21]. For all these components third-party solutions exist, but it is not uncommon that they are created in-house [1, 16, 21].

In order to bind the components together and make the engine easier to work with a layer on top of the major part of the engine is often built [1, 22]. Since this layer is used to build the gameplay of a game it is sometimes referred to as the gameplay foundation systems [1]. In this layer, systems for handling game objects like the world, the player, enemies and inanimate objects like buildings and furniture exists. There are also systems for handling events and rules in the game as well as artificial intelligence.

In order to design how game objects, events, gameplay rules and AI play

together the gameplay foundation systems implement a layer on top of the previously mentioned systems. The way game developers and designers use this layer varies between engines and individuals, but two common options are a scripting language or a visual programming language in which the flows and states of the game can be programmed [1, 16].

2.3.6 Game-Specific Subsystems

On top of the gameplay foundation layer and the other low-level engine components, lays the game-specific subsystems. These subsystems are often many and consist of features that make up a specific game. Since these subsystems are usually specific to each game, it is in or below this layer that the line between engine and game should be drawn [1]. However, as mentioned at the beginning of this section, it is not that easy to make a clear distinction since some game-specific data always seeps down through the gameplay foundations layer and might even reach the lowest parts of the engine [1, 21]

2.4 Tools in the Game Development Process

In order to make a game out of a game engine, the engine needs to be fed with data [1, 20, 22]. This data comes in several forms. Scripts and configuration files are given to the gameplay foundation systems to make up game-specific rules. Resources like 3D models, animation data, texture maps and audio files make up the world and characters of a game [1, 22]. In this section a description of the tools used to create that data is given. Although the tools are almost exclusively third-party and rarely used for game development alone, they are an important part of the game development process and the tool suite is often considered a part of the engine [1, 20, 22].

2.4.1 Programming Tools

The tools used when creating scripts and configuration files are generally the same as the tools used to develop the engine and will be familiar to anyone with a programming background. The tools used by programmers in game development projects are in most cases third-party tools, however, there are exceptions often in the case of a visual programming language implemented in the engine. The arguably most central tool in the development are code editors or IDEs, the most commonly used IDE in game development is Microsoft's Visual Studio. [1, 16]. Other important tools for programmers in a

game development project are debugging tools, profiling tools and tools for finding memory leaks and corruptions [1]. Tools like these are almost never implemented in a game engine, but are instead bought from IT giants like Microsoft, Intel and IBM.

2.4.2 Creative Tools

This subsection describes the tools used to create 3D models, animation data and texture maps. Just like in the case of programming tools, these tools will be familiar to anyone with a background in digital creation. The most commonly used tool for creating 3D models and animation data is Autodesk's Maya and the equivalent for creating texture maps is Adobe's Photoshop [1, 20].

When prototyping something called white- or grey-boxing is often practised, meaning that no textures and only simple models and animations are used in the game world. These simple models and animations can in some game engines be created with simplistic tools in the engine, but in most cases, the same tools as in production are used [1].

It should be noted though that there is usually a step between for example creating a model in Maya and using it in a game. The reason being that a file format other than the one provided by the third-party tool is usually preferred [1, 20, 22]. Binding these resources together into a game world is then something that is done in tools or programs specific to the game engine [1, 20, 22].

2.5 Differences between Game Development and Software Development

There exist some significant differences between game and traditional software development. For example, requirements are less functional, testing less common and teams less homogeneous in game development [12, 16, 18].

A lot of research has gone into the topic of finding and highlighting differences between game and ordinary software development and some of the most important discoveries will be presented in this section. Knowing these differences is important in order to understand why some traditional software methods or tools might not work in a game development environment. Understanding the differences will also help explain why some research done in software engineering might not be relevant for research in game development.

2.5.1 Requirements

Requirements might constitute the biggest difference between game and software development. In game development, the requirements are often fun rather than functional. It is not unusual that the only guiding requirement in game development is that the end product is fun and entertaining, something that rarely happens in software development outside the gaming industry [12, 16, 18].

Another big difference is how the requirements change during a project. In a game development project, detailed requirements change much more than in an ordinary software project [12, 16, 18]. This is because game designers can come to new realisations of what best fulfils the guiding requirement during the development.

The demand for reliability and perfection is also lower in the gaming industry than in other software projects [12, 18]. The reason for this is that the consequences of an unpolished feature in a game are often not as serious as in other software products, for example, a bug in a program used for controlling traffic is of course much more serious than a bug in a game.

2.5.2 Testing

As a result of quickly changing requirements and lower demand for perfection, software testing is not as extensively practised in game development as in other software projects [12, 18]. Another reason why testing is less prevalent in game development is that it is harder to test user experience than functional requirements.

2.5.3 Team and Work Process

The team and work process differ significantly between game and ordinary software development projects [11, 12, 16, 18]. In terms of team, the biggest difference is that game development teams are often much more diverse in terms of skills than in a traditional software project making it harder to manage a game development project. It is also the case that game projects use agile or hybrid models more extensively than in ordinary software development projects [11, 12, 13, 16].

2.6 Summary

This chapter presents the context and technical background needed to interpret the rest of the report. Chapter 2 exhibits the game development process, describing how a game goes from a concept, through prototyping and production to a finished product. The chapter also describes the responsibilities and relations between the designers, engineers, artists, producers and publishers working in a game development project.

The background chapter also exhibits the hierarchical architecture of the complex runtime component in a game engine and the tools needed to create content for a game. Finally, the chapter presents some of the differences between game development and traditional software development. The most notable differences presented are that requirements are fun rather than functional in game development, testing is also less commonly practices in game development and the development teams are often more diverse in terms of skills when producing a game compared to traditional software.

Chapter 3

Related Work

This chapter introduces the reader to previous research done at intersection between software engineering and game development, including the fields of prototyping, visual programming languages and software reuse in game development. At the end of the chapter, an analysis of the state-of-the-art Unity game engine is presented.

3.1 Reuse in Game Development

Reuse in software development is the practice of taking something old, in some cases make minor changes, and then use it in a new context. The idea behind this is, of course, to reduce time investment and save on development cost [13, 21, 23].

Reuse is very important in game development too, since the development costs are typically high and reusing assets can save a lot of money [12, 21, 24]. In game development, the part that is most often reused is the game engine, and it is not uncommon that one game engine is used in tens or hundreds of games over the course of several years [1, 21, 22]. It is also common that game assets like music and character models are reused between titles in the same game series [21].

Unlike the game engine, the game-specific subsystems that are described in Section 2.3 are something that are rarely reused between titles [1, 21, 23]. There are several reasons why game-specific subsystems have been less exposed to reuse than other parts of games. One of the more important reasons is that reusing puzzles, power-ups and menus can be ill-received by a game's audience [5]. There is also a risk in trying to reuse game-specific code since a change to the engine could make the code behave unexpectedly, this problem is

not as serious for game assets like 3D models and music [1]. Although challenging, it has been argued that reusing game-specific subsystems would be profitable and research into the topic has been conducted [23, 25, 24]. Some of the more recent research is presented below.

In 2016, van der Vegt et al. used a software architecture called RAGE, see Figure 3.1, to develop game-specific code that could be used in several different game engines [23]. The RAGE architecture can be described as similar to a client-server approach and using the architecture van der Vegt et al. were successful in implementing a simple 2D game in three different game engines [23].

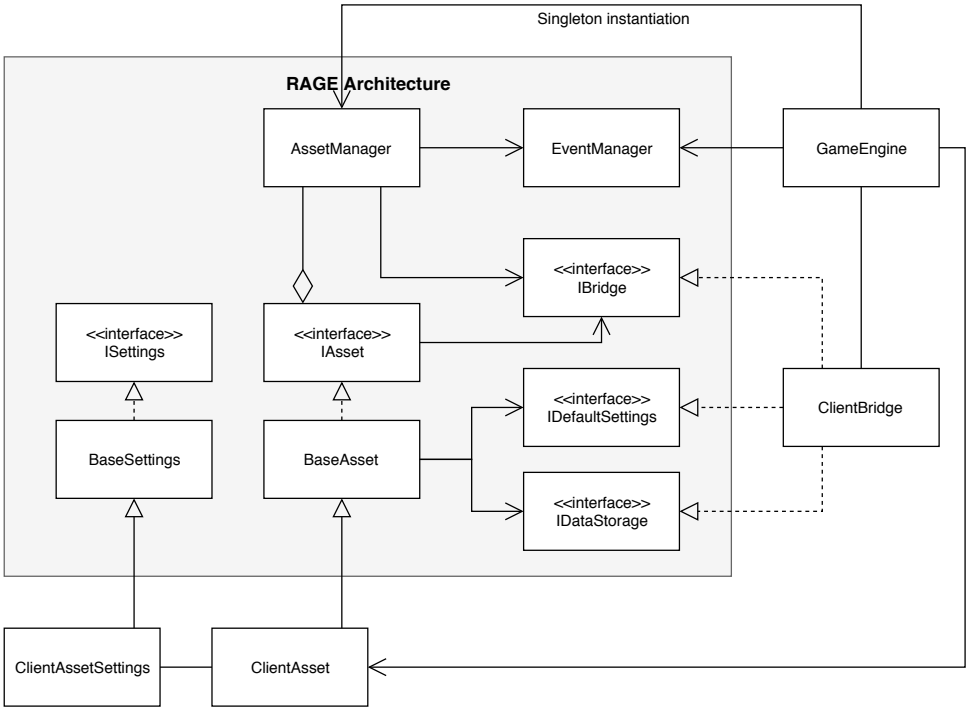

Figure 3.1: The RAGE architecture as described by van der Vegt, Nyamsuren and Westera [23]

In 2012, Madsen et al. developed a tool, see Figure 3.2, to easily create AI in games by combining reusable pieces of behaviour into a finite state machine and manipulating their transition conditions [25]. The approach was deemed successful, although it was not tested in the development of a full game [25].

In 2016, an attempt to automatically create reusable building blocks from game source codes was made by Nemitz [24]. The tool used an approach where it extracted classes with low cohesion and divided them into classes

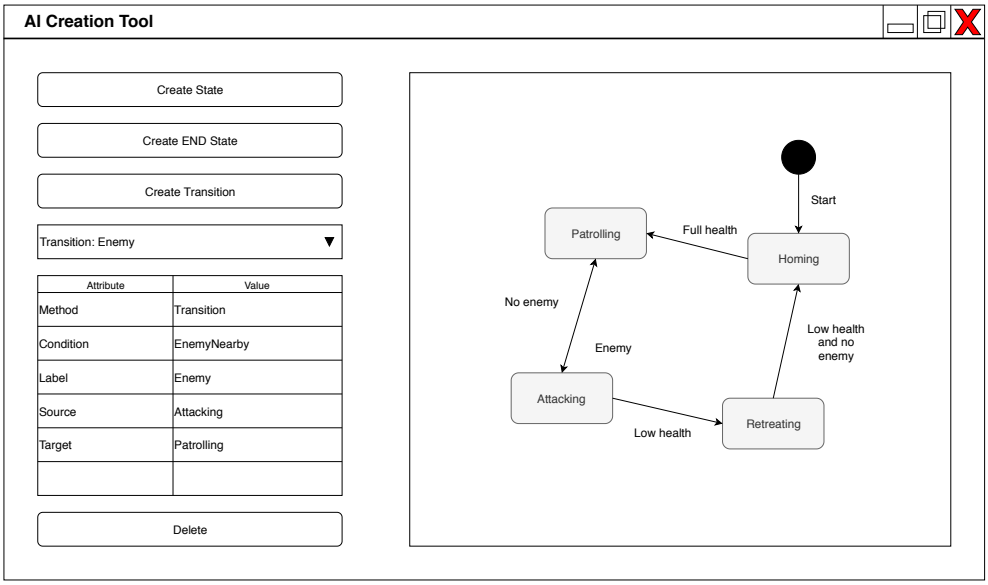

Figure 3.2: Blueprint of the AI creation tool developed by Madsen et al. [25]

with higher cohesion [24]. The tool was tested on submissions from a Python game jam and produced building blocks that were deemed more reusable than the original codebases [24].

3.2 Visual Programming

Visual programming offers a different approach compared to text-based programming for creating computer instructions. This is done by using visual representations like blocks, puzzle pieces and graphs in the programming process [26].

Visual programming languages have existed since the 1980s, but have recently gained a lot of traction as teaching tools for introductory programming and computer science [26]. Visual programming languages like Scratch [27] and Alice [28] are among the most popular for usage in schools were they teach programming to young students. Studies have shown that students both learn more and have a higher level of engagement when using visual programming languages compared to text-based programming languages [29, 30].

Visual programming languages are not only popular in schools and universities, but are also found outside the world of education. Robotics, interactive graphics and visualization are some areas where visual programming is used and has been shown to be a practical tool [31, 32, 33]. Another field where

visual programming is used extensively is in game development [1].

Since game development is such a complex task, several layers of abstractions are needed, and it is common that some type of high-level visual programming language is implemented for scripting gameplay and programming game-specific systems [1]. Examples of visual programming languages in game development are Unreal's Blueprints, CryEngine's Flow Graph and Frostbite's Schematics [34, 35, 36].

3.3 Prototyping in Game Development

Just like in other human-computer interaction fields, prototyping in game development is an important and commonly practised activity [37, 38, 39]. However, unlike prototyping in the general HCI field, prototyping in game development has only recently been subject to thorough research [37, 38, 39]. In this section, a presentation of some academic work about prototyping in game development is given.

In 2011, Manker and Arvola conducted a study in which they interviewed game designers from six different European game development companies asking if and why they prototyped in their development process [37]. The study showed that all companies used prototyping and although the second question yielded many different answers the two most common reasons for prototyping were communication and testing [37]. Game designers often used prototypes to communicate ideas that otherwise were hard to explain and to test concepts that were difficult to imagine [37].

There have also been studies investigating how the communication and testing of ideas could be made easier for game designers [5, 39]. The research has resulted in the creation of different tools that showed some success in theory and academic environments; however, they have never been tested in industry [5, 39].

In 2016, Gijsen examined how the prototyping process could be made more efficient through feedback loops [38]. Gijsen looked into three different cases of games developed at his university and the study showed that it was in general better with shorter feedback loops [38]. The study also showed that developers often spent too much time developing unnecessary details in their prototypes [38].

In 2010, a study was made looking into prototyping and game jams [19]. The study was conducted by Musil et al. who argued that both the game and general software development industry had much to learn from game jams when it comes to prototyping [19]. The main reasons for this were that pro-

totypes in game jams focus on key-features, are user-focused and developed very quickly with already available pieces [19].

3.4 State-of-the-Art Analysis

Whenever one tries to solve a problem that requires some sort of innovation, it is good to take a look at what has already been done and what is available today. In this case, this will be done by looking into the state-of-the-art Unity game engine.

There are of course many other game engines available today, some of the more popular choices are Unreal, CryEngine, GameMaker, Godot and Amazon Lumberyard [40]. However, due to time constraints, there is only time to look more carefully into one engine. Since Unity is one of the most popular game engines available, and it can be used to create any type of game [40], the choice for this state-of-the-art analysis is Unity. In addition to being one of the most popular game engines, Unity is also considered to be one of the easiest game engines to work with and it is by far the most used engine in game jam environments where development speed is of great essence [40, 41].

Unity's strongest asset, in terms of prototyping, use- and reusability, might be its asset store. On the asset store, there exist several thousand assets, available both for free or for a price of a few dollars up to a couple of hundred. The assets available on the asset store ranges from models, textures and sound files through tools for creating environments and quests to playable characters, enemies with AI and even complete games.

It is easy to download and import assets into a project through the project editor. Using the assets in the project is usually easy, but since there is only a minor reviewing process before an asset is published there is no guarantee that an asset works like expected or has thorough documentation. However, it should be stated that since Unity has a large user base and it is possible to review assets on the asset store, there is a degree of self-sanitation and many poor assets can be avoided by reading reviews.

All game objects in a Unity game project, regardless of them coming from the asset store or not, follow the same design pattern in the sense that they are always built by components. The components are smaller parts that make up certain parts of an object. Examples of components are textures, physical bodies and scripts, for example, character controller or enemy scripts.

Some components like textures, sounds and materials are easy to reuse, both within a project and in other projects. Other components, in most cases scripts, are not as easy to reuse since they can be dependent on other compo-

nents within the same object. Reusing these components without modification can cause unexpected behaviours and errors. For example, if you use a character controller made to work with an input handler for digital controllers together with an input handler for analogue controllers it is likely that you will get type errors where the character controller expects the input to be represented by Boolean values, but get floating point values instead.

Unity also makes it possible to save objects with configured components into prefabs which are then easily reusable within the project. Prefabs can also be exported for use in other projects. Unless one has caused some kind of dependency problem, for example making a weapon dependent on a character, this can be done with the click of a button. Importing the prefab into a new project is then done as easily as importing from the asset store.

By default, Unity does not come with a visual programming language. Instead, Unity relies on application scripting in C# through a hidden implementation of the development platform Mono [42]. Since Unity exposes a lot of functions and offers thorough documentation it is quite easy for a programmer to recode script components. If one prefers visual programming, it is possible to download tools that use visual programming from the asset store. Popular choices are NodeCanvas for building AI behaviour, FlowCanvas for scripting events and Bolt which works as a visual layer on top of C#.

3.5 Summary

This chapter presents research that has been done in areas related to game and software development. The chapter presents methods for reuse in game development, for example, an AI creation tool that allows users to create state machine based AI with reusable behavior blocks and a method for creating building blocks from game source codes by extracting classes with low cohesion and dividing them into classes with higher cohesion.

The chapter also includes information about visual programming and presents research showing that using visual programming increases understanding and engagement among young students when used in schools. There are also areas outside education that use visual programming and it is commonly found in robotics, interactive graphics and visualization. Visual programming is also used in game development and Unreal, CryEngine and Frostbite comes with visual programming languages for scripting gameplay and program game-specific systems.

Chapter 3 also looks into research showing that prototyping is common in game development companies and that the purpose of prototyping often is

communication and testing of ideas. The section also presented research arguing that prototyping is more efficient with short feedback loops and that game jams are good role models for prototyping since they focus on key-features and use as many readymade pieces as possible.

Finally, the chapter includes a state-of-the-art analysis into Unity, which is an engine considered easy to work with and often used in game jam environments. The state-of-the-art analysis showed that it is easy to use and reuse game assets in Unity and that its asset store, where users can upload assets, is a powerful tool for prototyping and usability.

Chapter 4

Method

This chapter presents the methodology used to examine how a set of building blocks that would allow for quicker prototyping can be composed. A general methodology for the study is given as well as a detailed description of how the interviews and evaluation were conducted. Motivations for the chosen methods are also presented.

4.1 General Methodology

The general methodology that was used in this study consisted of three concrete steps:

1. The first step consisted of finding information on what building blocks would help in the prototyping process and how these could be used and implemented. This information coincided with research questions 1, 3 and 4, and was gained by conducting semi-structured interviews.

2. The second step was the implementation of a proof-of-concept (POC). What was created was a tool that builds character controller to work as a state machine where actions in and transitions between states are editable through a visual programming language. The visual programming language uses nodes and it is these nodes that work as reusable building blocks. The implementation of the POC provided information for research question 3, where in a game engine, the building blocks should be implemented.

3. The third and last step consisted of evaluating the POC. Two environments were set up, one using the POC in Frostbite and one using an

example character controller in Unity. Game designers got to use both environments with the task of implementing a double jump ability and then provide feedback on the POC in terms of use- and reusability. This evaluation provided information that helped answer research question 1 and 4.

4.2 Interviews

The method for answering research question 1 and partly answering research question 4 consists of semi-structured interviews. This format was chosen based on several factors, the first being the number of available interviewees. Since it was only possible to interview 7 subjects, it was deemed insufficient to conduct structured interviews, since structured interviews would not provide enough quantitative data nor any qualitative data.

In preparation for the interviews, a systematic literature study into interviews and interview techniques was made. Adams provided more motivation for conducting semi-structured interviews in the circumstances of this study [43]. Adams gave several examples of situations were semi-structured interviews are well suited, two of which fit the situation of this study well: "*If you need to ask probing, open-ended questions and want to know the independent thoughts of each individual in a group.*" and "*If you are examining uncharted territory with unknown but potential momentous issues and your interviewers need maximum latitude to spot useful leads and pursue them.*" [43].

A total of 7 semi-structured interviews lasting between 50 and 90 minutes were held. Of the 7 interviewees, 5 held the role of senior game designer and 2 interviewees had managing roles related to technical design. All interviewees had at least 10 years of experience working as game designers. All interviews followed the same overall structure as described below, and the complete interview guide can be found in Appendix A:

1. Brief introduction to the master thesis project.
2. Description of the goals of the interview session.
3. Questions about the interviewee's prototyping experiences.
4. Probing questions into common prototyping activities and challenges.
5. Questions about tools used when prototyping.
6. Questions about things that, in the interviewee's experience have been working well when prototyping.

4.3 Creation of Proof-of-Concept

After the interviews, a proof-of-concept (POC) was implemented in Frostbite. The creation was a tool that builds character controllers that work as state machines. Actions in a state and transitions between states are editable through a visual programming language. The visual programming language uses reusable building blocks in the form of nodes. A detailed description of the tool, the implementation of, and motivation behind it can be found in Chapter 6.

4.4 Evaluation

The final step in the method was to evaluate the proof-of-concept. Since it was not possible to test the POC's use- or reusability in a real-life project, a controlled experiment was made instead. In this evaluation, the POC in Frostbite was tested against a script-based character controller in Unity in terms of prototyping capabilities, use- and reusability.

In preparation for the evaluation, two environments were set up, one in Unity and one in Frostbite. Both environments came with a started project, with empty game worlds, and a player character. In the Unity environment, Unity's 3D example character, Ellen was used, in the Frostbite environment, a character created with the POC was used. Ellen, the character in the Unity environment, came with a humanoid model and animation while the character in the Frostbite environment was a snowman without animation, however, both characters could move around in the world and perform a single jump.

The test that was made was to get game designers and game programmers to use both environments with the task of implementing a double jump ability in the character controller. In the Unity environment, the character controller could be modified using the C# script it was built with, in the Frostbite environment, the controller could be modified using the proof-of-concept. The way the test was set up meant that only scripting and the POC needed to be used and the evaluation subjects' experience of Frostbite and Unity had no impact on the results. All test subjects used both environments and were free to provide feedback on the environments while they were using them. The implementation attempts were timed, and the maximum allowed time for an attempt in one environment was 45 minutes.

After an implementation attempt had been made, the test subject was asked questions about the experience of the test and POC. They were also asked if

they believed the proof-of-concept would be easy to use in prototyping and production and how they thought it compared to the Unity equivalent. In total 10 tests, lasting between one and two hours, were conducted.

Chapter 5

Interviews

Results from the semi-structured interviews described in Chapter 4 are presented in this chapter. To improve readability the results are sorted by interview question topic.

5.1 Interview Subjects

S1 – Senior Game Designer with over 10 year experience;

S2 – Senior Game Designer with over 10 year experience;

S3 – Technical Director with over 10 year experience in game design and development;

S4 – Senior Game and Virtual Vehicle Designer with over 10 year experience;

S5 – Senior Game Designer with over 10 year experience from 4 different game studios;

S6 – Senior Game Designer with over 10 year experience;

S7 – Director of Technical Design with over 10 year experience in game design

5.2 Good Prototyping Experiences

For the first question in the interview, a question about good prototyping experiences, the first interview subject, S1, chose to describe when he was design-

ing a single player aircraft carrier mission for Battlefield 3. The interviewee described this experience as especially good, since the Frostbite team had at the time just improved the event system in the engine. They had also added the visual programming language Schematics, which made it possible for him, as a designer with at the time limited programming experience, to develop the mission largely by himself.

The first interviewee added that he loved the visual programming language from the get-go, and he thought the reason for that was his background as an electrical engineer. He also said that programmers today spend more time on developing nodes for Schematics than they do develop code directly for a specific game.

Another interesting prototyping experience was the one described by the second interview subject. S2 described a project he was currently working on together with S6. The game was in a very early prototyping phase and together with S6, they developed the prototype using a library called Allegro. S2 described Allegro as a lightweight library with which most development was done using the C programming language. He said that this was a good experience as it was easy to use already available assets like sprites and animations and that he could focus on the most important part of prototype development which according to him was gameplay. S6 said that this was the best prototyping experience he has had and it was thanks to being able to develop quickly and have short prototyping cycles.

S2 explained that he did not have a background as a programmer and that even though he quite often had to spend time solving memory-related bugs in the current project, he preferred the lightweight library over heavy engines for prototyping. S2 said that he liked it better having to spend time solving an issue instead of figuring out how to solve a problem which was often the case with heavyweight engines. S2 gave the example of fixing a function with poor performance. He said that he preferred spending a day rewriting the function over spending a day getting an optimization tool to fix the performance issue.

Both S2 and S6 concluded their descriptions of the current project by describing it as a pure prototyping experience. What they developed was not pretty or graphically impressive, but since they were in full control of the gameplay development, and playtests had so far been positive, they knew the game could only improve from here on.

5.3 Common Activities when Prototyping

The second question regarded activities that are common when developing gameplay prototypes. The first, third, fourth, fifth and seventh interview subjects responded that input management and character controller for the player were always done from scratch in new projects. They said that even though this was doable in Schematics, it was still quite complex to develop since games often have many different input commands and different platforms use different controllers.

S2, S3, S6 and S7 described User Interface (UI) and Head-up Display (HUD) as activities that always had to be done when prototyping, however, S4 and S5 said that UI and HUD often were ignored when prototyping because it was too much effort compared to their importance. The fourth, fifth, sixth and seventh interview subjects said that one always has to have some type of level and terrain in a prototype and that these often were cloned from old projects.

Weapons, vehicles, enemies and metadata files for information intended for the player were also described as somewhat common activities when prototyping, although not as common as the previously mentioned activities.

5.4 Challenges in Creating Prototypes

Question number three asked what some of the biggest challenges are in the development of gameplay prototypes. The most frequent answer was that it was difficult to create new components, both small, like nodes in Schematics, and large components, like the player or enemies. The given description was that when nodes need to be created, designers must get help from engineers because there was no easy way of creating nodes by themselves. The difficulty in producing large components was their complexity and dependence on many different systems.

All interviewees except S1 and S4 said that setting up a project was difficult. They said that most designers probably do not know how to set up a new project in Frostbite. S5, S6 and S7 also explained that there was no easy way of setting up essential game objects like the world, player and enemies.

S2, S3, S6 and S7 said that UI and Head-up Display are challenges when creating prototypes. S2 added to this by describing how UI and HUD were always dependant on game-specific code which makes them hard to reuse. S2, S6 and S7 also said that poor UI and HUD could cause prototypes to be

unfairly dismissed by playtesters who, because of the poorly made HUD, do not understand the game prototype.

The first and fourth interviewee responded that the most difficult implementation challenge was audio. According to S1 and S4 making the correct sound play on different surfaces and in different environments were some of the most challenging aspects of development and that bad audio can ruin any gaming experience.

The third interviewee said that any type of multiplayer prototyping was the most difficult challenge, something S2 also mentioned. Other challenges mentioned during the interviews were time constraints, understanding of game architecture, managing game objects, and animation.

5.5 How Tools Hinder Development of Prototypes

Question four discussed tools and if and how they have caused hindrance in the development of gameplay prototypes. The first interviewee described that the Frostbite engine has both native tools and a programming language called FB-Script that allows game development teams to create their own tools. S1 explained that most tools created in FB-Script were made for a specific game, but if they were usable outside a specific project they can be shared. Examples of native tools were tools for creating UI, animation, and the game world, examples of tools that can be created with FB-Script were tools that create forests or places objects randomly in the world.

The key takeaway from this question was otherwise that all interviewees complained about the usability of the tools. There was a general consensus that tools often hindered development by being too technical, too difficult to learn and too complex for creating something simple. All interviewees except S1 and S6 explicitly said that they wished there existed simple, uniformly designed tools for modelling, animation and creation of UI and Head-up Display in Frostbite. They explained that expertise was required for creating even simple models using the current tools which made it impossible for a single designer to create a prototype from scratch in Frostbite.

5.6 What Works Well in Development of Prototypes

The fifth question asked the opposite of question three and four and discussed what aspects of gameplay development that were easy and worked well. The first interviewee here described several aspects that were made easy thanks to the event system and Schematics. S1, together with all other interviewees, explained that game logic, quests and events were easy to develop today.

Another area that according to most interviewees is working well was the creation and editing of the game world. All interviewees except S5 said that they really liked Frostbite's world editor. S5 thought it was okay, but he would have liked the possibility to edit objects directly in the world editor and some type of visualization for how objects can interact with each other.

The second interviewee explained how the possibility to easily reuse assets like sprites and animations in Allegro was working well and aided him in his prototyping work. S1 and S3 both praised Frostbite's ability to allow changes to be made to a game while it was running and explained that it helped them get the correct feel in the game. S5 said that the live edit function was a good concept, but he complained that it became unstable and unpredictable in larger projects.

5.7 Code and Content Reuse

The sixth question regarded to what extent code and content is reused between projects. All interviewees explained that when creating a sequel in a series like Battlefield, prototyping rarely starts from scratch. Instead, they just modify the previous title into what they want the new title to be so in a sense everything is reused.

However, when it comes to the creation of prototypes for a completely new game, much less can be reused. The challenge, according to S1, in reusing larger parts of code and content was that they will inevitably be dependent on parts of gameplay foundation systems and game-specific subsystems that often are unique for individual studios and game projects. S2, S4, S5 and S7 described it similarly, S2 also exemplified the issue with the problem of reusing UI which was dependent on game-specific code.

S3 said that reusing larger components like weapons and vehicles could be done within the same studio and if the same engine version is used, however, this would still require recoding. Due to the dependency complexities, only

a few could manage this task. The fifth interviewee said that medium-sized components like explosions and fires could be reused if they were wrapped into something akin to a module. The first interviewee said that he thought more code and content could be reused if studios made things into more complete packages, like DICE's water system that was used in both Battlefield maps and outside sports arenas in EA Sports' titles.

All interviewees explained that content which was not dependent on any game code could be reused, for example, sound files, 3D models, and textures. However, one would need the original file for this type of reuse since it is difficult to extract such assets from a game. The fifth interview subject explained that models and physics become interconnected in Frostbite. He said that if he, for example, would like to place the physics and behaviour of a car on a barrel, he could not do that.

The first, third, fifth and seventh interviewee said that game studios build their own libraries of components, but that they rarely share these with other studios. The main problem, they explained, was that different studios use different versions Frostbite, there could also be issues with some components being dependant on other components in the same library.

5.8 Other Interesting Information from the Interviews

The last question allowed the interviewees to add information to the general subject of the interview. During the first interview, a lot of different and interesting discussions arose. The first interviewee had experience in working with both Frostbite and Unity and wanted to describe some of the ups and downs with Unity. He said that Unity was good, but that it missed the same support for large detailed world as Frostbite or Unreal.

The first interviewee also said that Unity's asset store was a good concept and it allows for easy code reuse, something that S3 also mentioned. According to S1, it was also quite easy to get different asset packs to work together in Unity, but it was an issue that developers of asset packs are not required to maintain them for at least a couple of engine versions and updates to the engine can render assets unusable.

On another topic, the first and third interviewee explained that in their experiences, as designers and developers, it was hard to know when you enter the pre-production phase from the prototyping phase. Often these phases flow together and most designers would not want it any other way. It would there-

fore not be good if something that worked in the prototype could not be moved into pre-production and later release. S1 and S3 said that if one managed to construct reusable building blocks they would have to be modifiable and modular so that they could reach release. However, S7 described it differently and said that it was better with a clear cut between prototyping and pre-production because that will make the code cleaner and easier to work with.

The second, third, fifth, sixth and seventh interview subjects all said that they would like some sort of tool or easier procedure to set up fundamental, but complex components when prototyping. S5, S6 and S7 suggested that one should implement a wizard for setting up the game world, player and enemies. According to S5, this could also work as validation for the Frostbite team when they make changes to the engine.

5.9 Key Insights from Interviews

With 5 of the 7 interviewees answering that the player's character, including input management and character controller were always done when prototyping a game from scratch, this was the most common prototyping activity according to the interviews. Another common prototyping activity was, according to the interviews, creation of UI and Head-up Display. 4 interviewees said that UI and HUD were always done in new prototyping projects, however, 2 other interviewees said that UI and HUD were often ignored when prototyping.

With 5 interviewees saying that that it was too difficult to develop new components, both large like the player or enemies and small like Schematics nodes, this was the largest prototyping challenge according to the interviews. The same 4 interviewees who said UI and HUD were always done in new prototyping projects said that UI and HUD were some of the biggest challenges in creating prototypes, making this the second largest prototyping challenge according to the interviews.

According to the interviews, the tools often hinder prototyping when they were too complex. 5 interviewees said that it was too difficult to set up a new project in Frostbite and that there is no easy way of setting up essential game objects like the world, player and enemies. 5 interviewees also said that Frostbite lacks easy-to-use tools for creating simple models and UI when prototyping. 3 interview subjects suggested that one should implement wizards for setting up the project, world and player as well as simple tools for modelling and UI creation.

All interviewees they liked the visual programming language Schematics and said that many aspects related to game logic have been made easy to de-

velop thanks to it. Relating to the fifth interview question, this makes Schematics the thing that works best in prototyping according to the interviews.

From the interviews, it was given that code and content reuse in game development is a complex issue, with all interviewees saying that it was difficult to reuse larger components like a soldier and put it in another game. The difficulty laid in differences in Frostbite versions and complicated dependencies to game-specific subsystems. 4 interview subjects explained that sound files, 3D models, and textures can be reused if one has access to the original file. Finally, it was explained by all the interview subjects that development of a sequel in a series rarely starts from scratch, instead the previous title was modified into the new title.

Chapter 6

Creation of a Proof-of-Concept

This chapter presents the proof-of-concept, the implementation of it and the motivations behind it.

6.1 Requirements and Motivation

The interviews showed that creating a player character, including input management and character controller, are the most common activities when prototyping a game from scratch. According to the interviewees, it is also too difficult to develop new components, both large like the player or enemies and small like nodes in Schematics. The interviews also showed that tools should not be too technical and that all interview subjects liked the visual programming language Schematics. This meant that the following requirements existed when the proof-of-concept (POC) was made.

- The POC should address one or more of the most common prototyping activities.
- The POC should not force designers to develop new components.
- The POC should not require expertise to be used.
- The POC should, if suitable, use visual programming.

The POC that was created was a tool that builds character controllers to work as state machines where the actions in states and the transitions between states are editable through a visual programming language. The visual programming language uses reusable building blocks in the form of nodes. The

motivation behind the choice of proof-of-concept comes from the requirements, but also from the fact that there already existed reusable solutions for input management so there was less motivation to work with that aspect. To overcome the issue of having to create new nodes, the nodes used in the POC were meant to be of a general design so that they could be used in many different contexts.

6.2 The Proof-of-Concept

The created proof-of-concept was a tool that builds character controllers to work as state machines where actions in and transitions between states are editable through a visual programming language. The tool used existing solutions for input mapping and attached to a physical player object to be able to interact with the game world.

Looking at a character controller created with the POC from a high-level one would see a character controller behaving as a state machine with endpoints for animation. Looking into the character controller one sees the state machine with actions and transition conditions editable using the POC tool.

Figure 6.1 displays how the character controller can be edited using tool in Frostbite. The state that is shown is Jumping and the action ApplyJumpMovement is selected. To the left, all states in the character controller are shown and to the right, the property editor for the action ApplyJumpMovement.

In the middle are the entry conditions, actions and transition conditions editors for the state. Rounded rectangles are conditions that either return true or false. Rectangles are actions; parallelograms represent input and can either output the input-data into an action or cause a transition to another state. Finally, the hexagons are transition nodes that represent states that can be transitioned into.

The entry conditions help the state machine decide if it should use the state as entry in case it currently is not in a state. This can, for example, happen when the game starts or in server synchronizations, the state machine will then look through the states from Standing to Falling and enter the first one where the conditions are fulfilled.

Actions define what happens in a state. An action can either happen once on state entry or exit or with set intervals while in the state; it is also possible to use conditions in the action editor. The order actions take place are determined by their placement in the editor, the first action is placed in the top-left corner and the last action is placed in the bottom-right corner, just like one would read this paragraph.

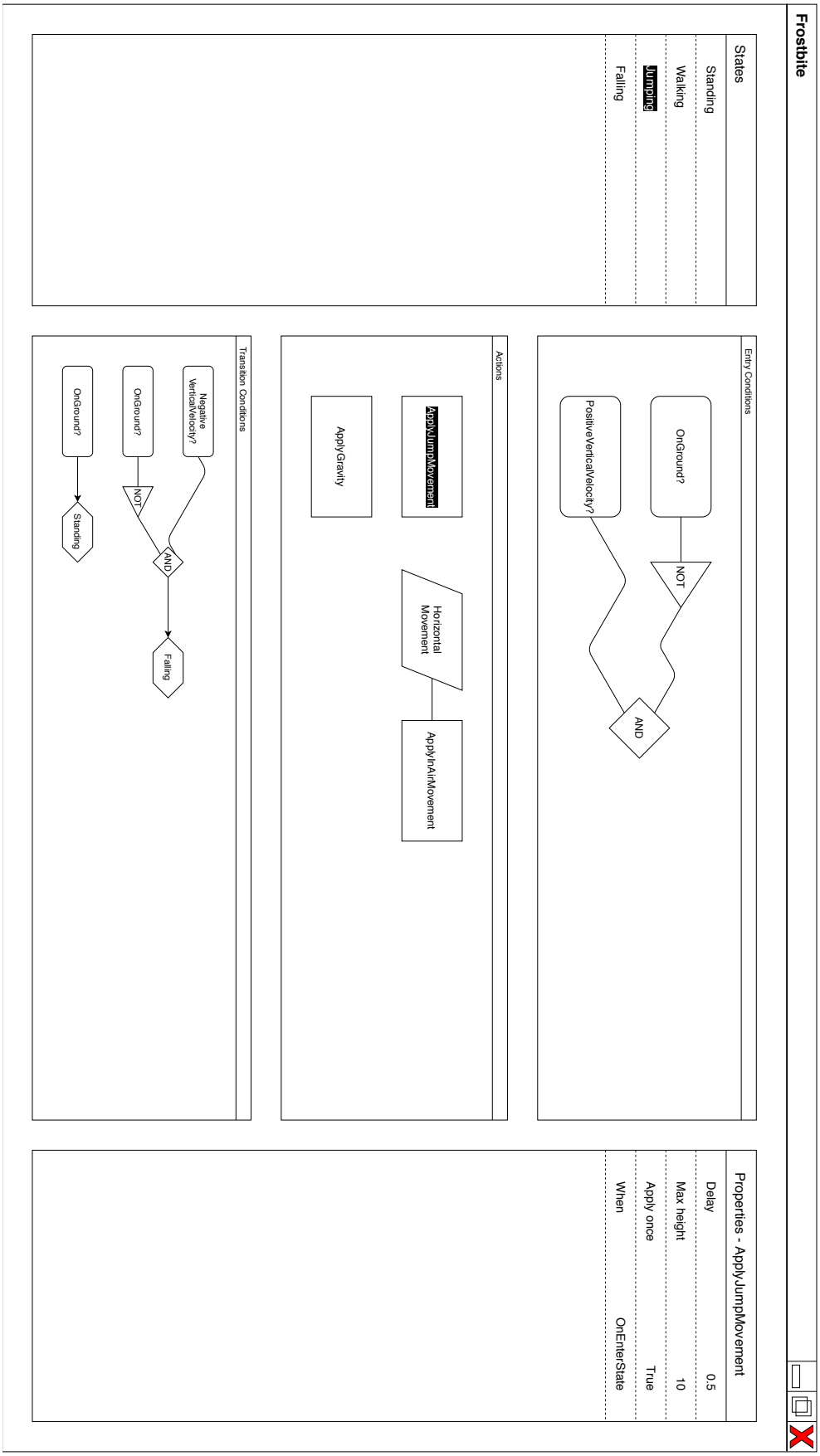

Figure 6.1: Blueprint of the character controller state machine editor in Frostbite.

Transition conditions are the conditions that must be met for a state transition to take place. The condition checks are read using the same placement based logic as actions.

No nodes are dependent on game-specific code and will be easy to reuse. States can also be reused, but since one state might transition to another state there can be dependency issues, however, this should be easily fixed using the tool. The inspiration for this proof-of-concept is the AI creation tool developed by Madsen et al. [25] and a 2014 Frostbite based game, which used a similar tool.

6.3 The POC as an EFSM

The semantics of the state machine in the POC can be described using Cheng's and Krishnakumar's Extended Finite State Machine Model [44]. Cheng and Krishnakumar describe their model the following way:

> An Extended Finite State Machine E is defined as the 7-tuple $\{S, I, O, D, F, U, T\}$, where:
>
> S is a set of symbolic states,
> I is a set of input symbols,
> O is a set of output symbols,
> D is an n-dimensional space $D_1 \times \ldots \times D_n$,
> F is a set of enabling functions f_i such that $f_i : D \to \{0, 1\}$,
> U is a set of update transformations u_i such that $u_i : D \to D$, and
> T is a transition relation such that $T : S \times F \times I \to S \times U \times O$.
>
> We will use x to denote an n-dimensional vector with components $Xi \in Di$. We denote a transition $T((s_1, f, i), (s_2, u, o))$ as $(s_1, f, i) \to (s_2, u, o)$, where $s_1, s_2 \in S, f \in F, u \in U, i \in I$, and $o \in O$. Further, $(s_1, f, i) \to (s_2, u, o)$ means that if E is in symbolic state s_1, with a vector of variables x such that $f(x) = 1$ and the input i is received, then E moves to the symbolic state s_2 while generating output o and performing the update $x \leftarrow u(x)$. [44]

In our POC, we do not use outputs; furthermore, in our models, there is always only one update function per state pair, so our EFSM is deterministic. Also, since the enabling functions $f_1 - f_n$ can be dependent on input, we can

denote the transitions in the POC as $(s_1, f') \rightarrow (s_2, u)$ where f' is a higher-order function that processes data and input, $f' : D \times i \rightarrow \{0, 1\}$.

For example, one of the transitions out of the state shown in Figure 6.1 can be expressed as $(Jumping, f_1') \rightarrow (Falling, u_1)$. f_1' is a the function $(NegativeVerticalVelocity \,\&\, \neg OnGround)$ and u_1 are the actions in the Falling state $(ApplyInAirMovement, ApplyGravity)$. One of the possible transition to the state shown in Figure 6.1 can be expressed as $(Walking, f_2') \rightarrow (Jumping, u_2)$. $f_2' : (Jump)$; $u_2 : (ApplyJumpMovement, ApplyGravity, ApplyInAirMovement)$.

The entry conditions can also be modelled using Cheng's and Krishnakumar's model. The way this is done is by adding an extra set of transitions from a synthetic initial state, which chooses the desired initial state from the model.

6.4 Implementation

The proof-of-concept that was created, built upon on a tool created for a game from the year 2014. The 2014 tool provided a basis for the state machine editor; however, the old tool combined the Action editor with the Transition editor, unlike the POC which separated the editors. Also, the old tool made no visual difference on the type of nodes, unlike the POC where nodes of different categories had different appearances. The 2014 tool also provided the Boolean operators used in the proof-of-concept as well as the Transition node. All other nodes in the POC needed to be created since the old tool only used game-specific nodes that were unusable outside the context of the 2014 game it was made for.

The building blocks, or nodes as they are referred to in the context of the tool, created for the POC are meant to be usable in more games than one. In total five action nodes, three condition nodes and two input nodes were created for the proof-of-concept.

The 2014 tool used an architecture where the tool itself was coded into the Frostbite tool suite using C#. The nodes the tool used were constructed using C++ at a high level in the Frostbite runtime. The character controller the tool builds will also be constructed at a high level in the game engine runtime. In discussions with members of the Frostbite team it was decided that the POC should use the same architecture since the team thought it was a suitable architecture for the tool and the building blocks it used.

Chapter 7

Evaluation Results

The results from the evaluation described in Chapter 4 are presented in this chapter.

7.1 Test Subjects

The test subjects in the evaluation can be divided into two different groups. The first group are game designers, the intended users of the POC, and the second group are game programmers who can provide a more technical perspective on the tool and its building blocks. In total, 6 designers and 4 programmers used and evaluated the POC. Of the designers, 3 started with the Unity environment and 3 with the Frostbite environment, among the programmers 2 started with Unity and 2 with Frostbite. Details about the test subjects and their implementation times can be found in Table 7.1.

There were some overlapping of persons in the evaluation and persons in the interviews. D1 in the evaluation is the same person as S1 in the interviews, D3 is the same person as S5; finally, D5 and S2 are the same person.

7.2 Implementation Times and Statistical Significance

Of the successful attempts, all evaluation subjects spent less time implementing the double jump feature using the POC than they did using scripting. The median time for an implementation attempt in the POC was over 40 per cent shorter than the median implementation time with scripting. When counting an unsuccessful implementation attempt as the maximum allowed time of 45

ID	Role and Experience	Starting Environment	Scripting Result	Time (min)	POC Result	Time (min)
D1	Senior Game Designer, over 10 years	Frostbite	Gave up	5	Gave up	22
D2	Game & Level Designer, 2 years	Unity	Time Limit Exceeded	>45	Completed	24
D3	Senior Game Designer, over 10 years	Frostbite	Gave up	5	Completed	21
D4	Senior Game Designer, over 10 years	Frostbite	Completed	15	Completed	11
D5	Senior Game Designer, over 10 years	Unity	Completed	12	Completed	5
D6	Senior Game Designer, over 10 years	Unity	Completed	32	Completed	20
P1	Gameplay Programmer, 2 years	Unity	Completed	20	Completed	6
P2	Senior SW Engineer, over 10 years	Frostbite	Completed	11	Completed	9
P3	Software Engineer, 5 years	Frostbite	Completed	17	Completed	10
P4	Gameplay Programmer, 1 year	Unity	Completed	12	Completed	8
-	**Median, All Evaluation Subjects**	-	**Completed**	**18.5**	**Completed**	**10.5**

Table 7.1: Test Subjects and Implementation Times

minutes, the difference is statistically significant with a significance level of 0.01 in a non-directional Wilcoxon signed-rank test.

7.3 Evaluation Subjects' Impression of the POC

The general impression of the POC was positive. All test subjects except D1 said that it was easy to understand how the state machine works and how they can edit it. Only D1 said that he did not know how a state machine usually works and therefore he thought it was difficult to use the POC. D3, D4, D6, P1 and P3 said it was easy to understand the hierarchy for deciding the starting state. D1, D3 and D6 suggested that there should be a way of "zooming out" to see the relation between states. D3 and D4 also suggested that one should be able to enter the editor for another state by double-clicking on a transition node.

The importance of the order of the nodes was unexpected. D1 and P1 pointed out that it might cause confusion that the order nodes are placed in matters to how the state behaves. D1 and P1 said that in new versions of Schematics the order does not matter, and they made it so because those who have less programming experience find it easier. However, no one in the test said that they disliked that fact that the order mattered. D5 even said that he liked that the order mattered and that this will result in cleaner states.

The current version of the POC causes node repetition. D3, D4, D6, P1 and P3 all said that the current implementation of the POC forces you to repeat a lot of nodes. P1 said that "A similar design in code would not be accepted since code repetition is a bad habit and in code review, I always fail others for making such mistakes". D3, D4, D6 and P1 suggested that one should implement sub-states in the state machine that should execute simultaneously as its super-state. D6 also suggested that one should have the possibility to use references to states as sub-states. To motivate, he exemplified: "If I have an Attack state and want it to work the same way when on the ground and while jumping, I should not have to create two sub-state clones, instead I should just be able to create a reference to the Attack state".

All evaluation subjects liked the idea of prototyping using the POC. D2 was very positive about the POC and thought it would make a good addition to their prototyping work already in the current version. The rest thought it would be a great tool for prototyping if some changes were made to the POC, the most common suggestions concerned sub-states, debugging functionality and design of the building blocks.

The POC might not provide enough fidelity for the latest AAA titles. On the question of whether the POC would be a good tool in production or not, the answers were less uniform than the question about prototyping. All evaluation subjects except P2 and P3 thought it would make a good tool in production if suggested changes were made. P2 and P3, on the other hand, said that decoupling animation from the logic of a character would not allow for enough fidelity in the animation. P2 explained "It certainly is possible to separate logic and animation for a character, in fact, it is probably the most common way of doing it and we used to do that as well, but when you want to create a highly detailed game like Battlefield V, where every frame in an animation is considered, you need to combine animation and logic".

7.4 Evaluation Subjects' Opinion of Scripting in the Unity Environment

The impressions of scripting were mixed. In the Unity environment, the evaluation subjects used scripting to build the character controller instead of visual programming, see Section 4.4. The impressions of this method were not as positive as for the POC. All evaluation subjects except D1 and D3 said that they liked knowing what their editing options were. D4 explained it best: "You can write code and there is not much more to it, from the beginning, you know your options and the editing potential you have". D4, D5, D6 and P1 also said that the control and editing possibilities are better when using code compared to visual programming.

There are difficulties related to coding. D1, D2, D3, D4 and P4 all said that it is hard to get an overview of a character controller made with code and that anyone without decent coding experience will find editing it overwhelming. D1, D2, D3 and D6 also said that it is too easy to get stuck on small details when coding.

Scripting is good for small prototyping projects. D3, D4, D5, D6, P1 and P3 said that they believed that the method in the Unity environment would work well for prototyping, however, P2 and P4 said that the prototype would have to be very small for the method to be sustainable. P2 stated "You can use scripting if you build a game by yourself, but as soon as you start involving other people it becomes unsustainable. For instance, look at this example, we have nearly 700 lines of code and all we can do is run, jump and double jump, you can imagine what happens as soon as we add advanced behaviours".

Scripting becomes unsustainable in large projects. The issue of code becoming unsustainable in larger projects repeats itself in production. Only D4 and D5 thought that scripting could be used in the production phase of a game, while D1, D3 and all the programmers said that it would become uncontrollable in a large project. The issue, as the programmers described it, was that in game development, it is common that team members with limited programming experience want to tweak parts of the game to get the correct feel. Therefore, there exists a demand for easy ways of tweaking the game, and in case one let people with limited programming experience modify the game source code the code quality would quickly become poor.

7.5 Preferred Method

When the evaluation subjects had used both environments, they were asked which of the methods for building a character controller they preferred. All game designers except D5 and D6 preferred using the POC in Frostbite over scripting in Unity. Although D1 did not complete an implementation in either environment he said “I prefer the POC, if I had time to learn about state machines and more about the tool, I’m sure I would come to like it, I know how to program, but it always takes me more time than it would using visual programming”. D2 was indivisibly positive about the tool, but not negative about scripting either. D3 was positive about the POC, but negative about scripting with the motivation that he did not like to program.

D4 was positive about both environments, but preferred the POC since he believed it would be more scalable and better suited for large projects than scripting. D5 preferred scripting over the tool with the motivation “You will always have more control with code than you would with visual programming”. D6 also preferred scripting with a similar motivation, but he added that he thought it was a difficult choice and that the POC would probably be the better tool when working with other designers.

Among the programmers, 3 preferred the POC and 1 preferred scripting. P1 preferred scripting, he said that he had always been coding and that he liked doing so. P1 also said that while he was not negative about the POC he knew that it could never provide the same control and flexibility as code can. P2, on the other hand, preferred the POC, his motivation was that scripting will not scale to the level necessary to produce a prototype for a triple-A title. P3 and P4 were positive about both environments, but they preferred the POC. P3’s motivation was that he liked working with state machines and P4 said that "The tool simply feels more modern than code".

In total, seven out of the ten evaluation subjects preferred using the POC over scripting. The most reoccurring reasons why, were that the POC was easier to use and more scalable for larger projects. Three evaluation subjects preferred scripting with the motivation that it provides more control and flexibility than the tool could.

7.6 Applying the POC Design on Other Game Features

When having tested the POC, the evaluation subjects were asked if they thought its design could be applied on the development of other game features. All evaluation subjects could think of at least one other feature that could be designed to work as a state machine. D4, P1 and P2 went as far as to say that anything in a game could be designed to work as a state machine. D4 and P2 even said that this could work well while P1 said that he did not think one should try to design everything in a game to work as state machines.

The most common examples of game features that evaluation subjects thought could benefit from being designed as state machines were AI, quests and physical objects. D3 said that "We already constructed the enemies in Mirror's Edge Catalyst to work as state machines and that worked well, but it would be great if we could control all physical objects like this. For example, if I have a physical object and it catches on fire it might enter a new state and get new properties, we have no real way of controlling this today". Other suggestions of game features that could be designed as state machines were game modes, weapon systems, shaders, light and sound.

7.7 Building Blocks and Categorization

Feedback on the node categories was positive. In the POC that was shown to the evaluation subjects, all building blocks or nodes, in the state machine belonged to 5 categories: Actions, Conditions, Input, Boolean operators and Transitions, which only contained the Transition node. All evaluation subjects thought the categorization made sense and only minor complaints were made on the categories. D2 and D3 thought that Input should be named User-Input and P2 thought that Actions could come to have sub-categories if one added more nodes to it.

Feedback on the building blocks was mixed. The feedback regarding the building blocks or nodes as they are referred to in the context of the tool was less positive than the feedback on the categories. There was a general consensus that the current nodes were too specific and could be replaced or at least complemented with more general nodes. For example, the nodes ApplyGravity and ApplyJumpMovement could be replaced with a single node

ApplyForce which with the use of direction parameters could be made into both jump and gravity behaviour nodes.

The evaluation subjects provided the same type of feedback in concern to the condition nodes. For example, the NegativeVerticalVelocity? node could be replaced with Getter and Compare nodes. Regarding the ApplyInAirMovement and ApplyOnGroundMovement nodes the evaluation subjects were less certain about their answers, on the one hand, these behaviours could be built using general nodes, but on the other hand, it would have been quite difficult.

Summary. In total, all evaluation subjects thought that the specific nodes should be replaced or complemented with more general nodes. D3, P1 and P2 thought that no specific nodes should exist by default, while the rest thought that both options should be provided. D1, D4, D5, P1, P3 and P4 thought that there should be some way of creating specific nodes in the tool. D5 suggested that one should have a Script node that any specific behaviour could be coded into. D1, D4, P1, P3 and P4 suggested that it should be possible to select a cluster of nodes and transform it into a new specific node. D4 and P3 thought that any new nodes that are created by a user should be forced to belong to one of the categories that existed now.

7.8 Suggested Changes to the Tool

This section summarizes the change suggestions made by the evaluation subjects. If one wanted to build a tool like the one evaluated in this chapter and use it in a real game development environment, the changes described in this section should be made.

7.8.1 Sub- and Reference-States

Adding sub- and reference-states are, according to the ten evaluation subjects, one of the most important changes to be made to the tool. The sub-states should be placed below its super-state in the listing to the left and execute at the same time as the super-state. Reference-states should work like sub-states in the sense that they are declared below a state and execute simultaneously as its super-state. But instead of being a new state, a reference-state should be an uneditable link to another state. It should be possible to reference sub-states and create sub-states to sub-states without limitation.

Sub- and reference-states can be modelled using Cheng's and Krishnakumar's Extended Finite State Machine Model, see Section 6.3. For example, if

the transition from the Walking state to Jumping is expressed as $(Walking, f_2')$ $\rightarrow (Jumping, u_2)$ where $f_2' : (Jump)$ and $u_2 : (ApplyJumpMovement,$ $ApplyGravity, ApplyInAirMovement)$, the transition from Jumping to DoubleJumping can be expressed as $(Jumping, f_3') \rightarrow (DoubleJumping, u_2)$. The enabling function f_3 is modelled as $f_3' : (Jump)$. Although $f_2 = f_3$ in this case f_2 cannot be inherited since Jumping is not a sub-state to Walking. However, the update transformation u_2 can be inherited since Jumping is a super-state to DoubleJumping.

Sub- and reference-states will need to be represented in a new way in the state listing. For example, it would make sense if sub- and reference-states are indented compared to their super-states and that reference-states are listed with another text-style. Although it is not necessary to have an enforced naming standard for sub- and reference-states in the editor, the animation anchor points they create need to have names that make it easy for animators to work with them.

It is also likely that one will want to have the possibility to override certain super-state functionality in a sub-state. The easiest way of doing this would be with a checkbox in the editor to override either the entry conditions, actions and/or transition conditions. One can also imagine a way to group nodes within a section and then decide to override only a group in a sub-state.

Figure 7.1 displays a blueprint of the tool with the suggested sub- and reference-state changes. In the blueprint, the DoubleJumping sub-state and the inherited ApplyJumpMovement action node are selected. The properties in ApplyJumpMovement are greyed out because the node is inherited and cannot be modified. All nodes in the entry conditions and actions sections are inherited and thus displayed with dotted lines. The super-state's transition conditions are overridden and can be created anew in the sub-state.

7.8.2 General Nodes

One of the most central changes that need to be done to the tool is to add general building blocks, or nodes as they are referred to in the context of the tool. There will be a need for several categories of new nodes, the most important are presented in this section.

Accessors for character data. One getter and one setter node for character data, the specific values can be set with parameters. It is likely that one wants to have the option to manipulate stats and modifiers in the state machine. For example, one might want the character to take more damage while hovering

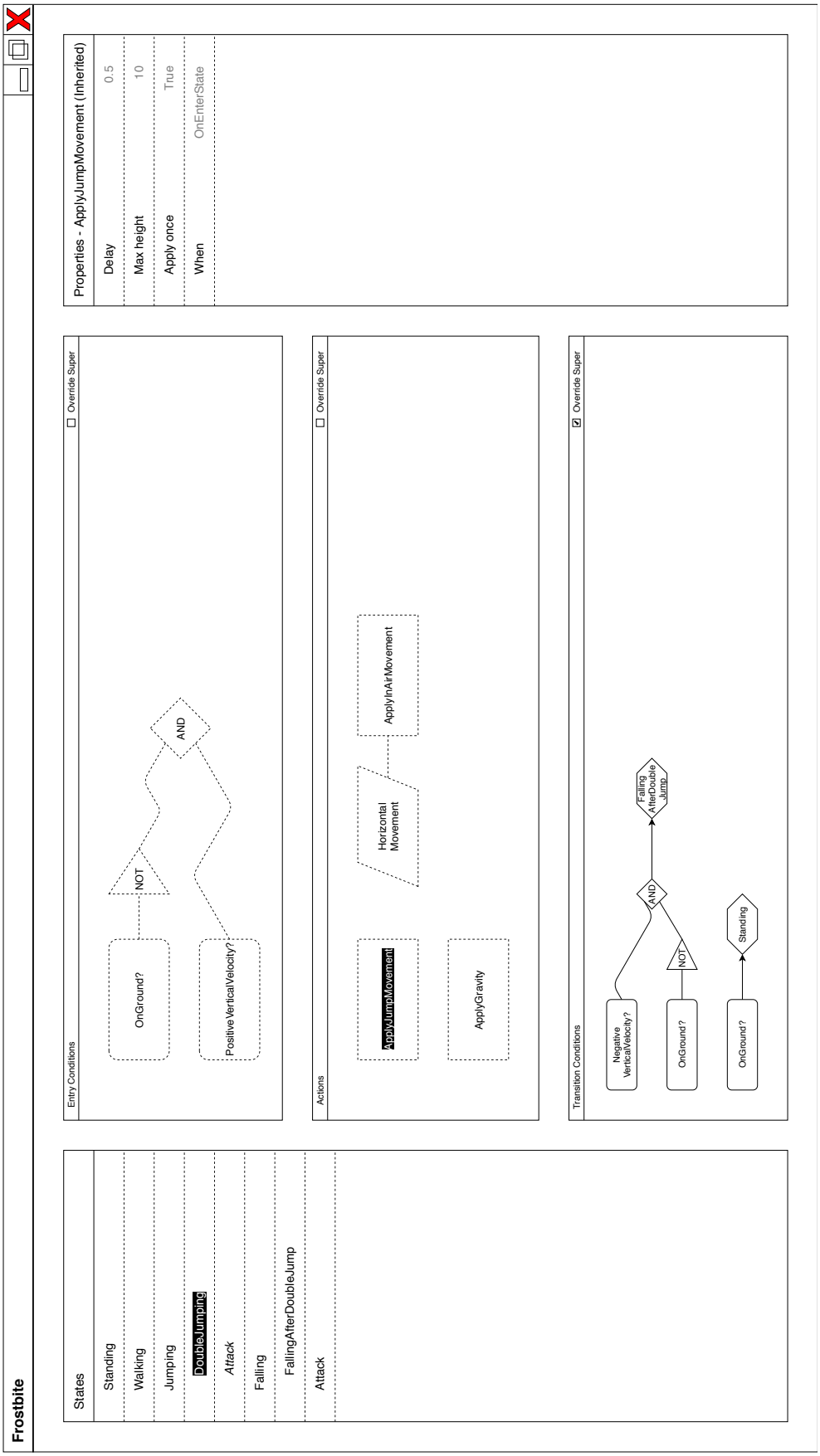

Figure 7.1: Blueprint of the character controller state machine editor with suggested sub- and reference-state changes.

and less while running.

Getters for character physics data. It will be necessary to have getters for player information like velocity and movement direction.

Getters for physical objects, materials and interactables. One will want to be able to get distances to and data about physical objects, materials and interactables in the game world.

Handles for interactables. One will want to have some way of making state transitions dependant on interactions with objects like levers and buttons.

Compare, arithmetic and mathematical function nodes. It will be necessary to have the possibility to make comparisons between variables, arithmetic calculations and apply functions like the power of, roots, min/max, sign, sine and cosine on variables.

Timers. One might, for example, want to exit a state after a certain time or amount of physic steps and will need nodes for this functionality.

Apply force nodes. To be able to replace the specific nodes that exist now, it will be absolutely necessary to have nodes for applying forces to the character and make the character apply forces to the game world and its objects. These nodes will require a variety of parameters like the strength of the force, force direction and time of force, they will also require inputs so that parameters can be set dynamically. In a more advanced version, one could imagine these values being manipulated with graphs similar to how audio can be manipulated with an equalizer.

Object transformers. One will want the possibility of rotating, scaling and directly move the character in the game world.

Camera handlers. Camera handlers are needed to make it possible to design the character controller with dependence on camera mode and vice versa. For example, a character might behave differently depending on if the camera is set to first or third person view, the camera might also behave differently if the character is in a battle state or not.

General user-input nodes. Instead of having an input node for jump, the node should be nameable and map directly to a button on the keyboard or controller. Analogue inputs will likely require its own type of user-input node.

7.8.3 Combining Node-Clusters into New Nodes

The evaluation showed that the evaluation subjects wanted some way of combing clusters of nodes into new nodes. The functionality for creating new nodes need be user-friendly in the sense that it should be easy and intuitive to create new nodes. One suggestion would be to make it possible to circle nodes in the editor with a drawing tool and have a “Create Node” – button that becomes clickable when a cluster of nodes has been selected. Created nodes should be able to have inputs and outputs. If input and/or output lines are crossed when circling a cluster, corresponding input and outputs should be automatically created for the new node.

It would be desirable if there existed an easy way of sharing nodes, states and complete character controllers between Frostbite projects. It is possible to imagine a function similar to Unity’s asset store, however, it is likely that a lot of what will be uploaded will depend on other parts of its origin project. Although there likely will exist a degree of self-sanitizing like on the asset store there are not enough Frostbite users working with character controllers for this to be reliable.

Most likely it would be best to just allow nodes to be shared since they have the least risk of being dependant on other parts of their origin projects. It would also be good if someone who maintains the tool has the possibility to add nodes from other projects to the native set in the tool.

7.8.4 Usability Improvements

To improve the usability of the tool and make it more viable in a real game development setting the following additions should be made:

Functionality for “zooming out”. To get a better overview of the state machine it should be possible to "zoom out" and see the entire state machine in a way similar to the tool created by Madsen et al. [25]. Since it is likely that this view will be cluttered in a state machine with many states, sub-states and reference-states there should be options for highlighting and hiding different kinds of states.

Debugging functionality. The tool needs some sort of debugging functionality. In the first version, it would likely be enough to be able to follow the state machine transitions in a zoomed-out state machine view where the current state is highlighted. However, the optimal would be to implement full debugging functionality similar to what is found in an IDE. This should include stepping functionality where executed nodes are highlighted and a possibility to inspect values that are sent between nodes.

Grid for placing nodes. Since the node-order matters and it should continue to matter, a grid for placing nodes in the editor is necessary to avoid confusion.

Double-clicking for transition. It should be possible to open the editor for a new state by double-clicking a transition node in a previous state.

Chapter 8

Discussion

This chapter puts the results presented in Chapters 5, 6 and 7 in relation to the research questions posed in Chapter 1. Chapter 8 also discusses the study's limitation and threats to its validity.

8.1 Results and Relation to Research Questions

The goal of this project was to find an answer to the question:

How can a set of reusable building blocks that would allow game designers to quickly prototype gameplay ideas be composed?

In finding an answer to this question, four research questions were examined. Each question will be discussed in this section.

RQ1: *What is a set of building blocks that would allow game designers to quickly prototype gameplay ideas?*

The study has shown that it is difficult to know what a set of building blocks that would allow game designers to quickly prototype gameplay ideas is. This is because the building blocks can be many different things depending on the aspect of gameplay prototyping. The interviews suggested that it would be most useful to focus on the aspect of creating character controllers and it is for that aspect this project can give an answer.

A set of building blocks that would allow game designers to quickly prototype gameplay ideas is a set of nodes in a tool that builds character controllers.

The tool builds the character controller to work as a state machine where actions in a state and transitions between states are editable through a visual programming language that uses the nodes. The building blocks, or nodes as they are referred to in the context of the tool, should be functionally small and general like accessors for character data, handles for interactables, comparers, object transformers, camera handlers and nodes that apply forces to the character and the game world. For more details about the tool and its building blocks, see Chapter 6 and Section 7.8.

RQ2: *What is the closest equivalent to reusable building blocks in Unity?*

This question was investigated with a state-of-the-art analysis. The analysis showed that Unity makes it easy to use and reuse both small and large parts of a game. All objects in a Unity game are constructed with what are called components, and objects easily can be turned into so-called prefabs which are then easy to use in other projects. For example, a prefab can be an entire player character and the components it is built with can be an input handler, a character controller, an animator, a mesh and a texture map.

Unity users have the possibility to upload their components and prefabs on the Unity asset store. It is easy to download and use both prefabs and components from the asset store and use them in a project. Although larger in concept and more complex in functionality than the building blocks proposed by this project, the closest equivalent to reusable building blocks in Unity is prefabs and components.

RQ3: *Where in a game engine should reusable building blocks be implemented?*

The building blocks are implemented in the runtime component of the game engine and the tool in the game engine's tool suite. In detail, the building blocks are implemented at a high level in the engine. To refer to the model presented in Chapter 2, the building blocks exist in the game specific subsystems layer. The character controller the tool creates is also placed in the game specific subsystems layer.

RQ4: *What is a suitable design that makes reusable building blocks easy to work with for game designers?*

This question goes back to the first research question. The evaluation showed

that the tool described in Chapter 6 had a good design to work with reusable building blocks. To summarize, the tool builds the character controller to work as a state machine where actions in a state and transitions between states are editable using a visual programming language. The nodes in the visual programming language are the building blocks and these should be functionally small and general like setters and getters for character data, handles for interactables, timers, object transformers, and nodes that apply forces to the character and the game world it exists in.

8.2 Limitations and Threats to Validity

At the beginning of this project, the goal was to make it possible to prototype gameplay faster by facilitating reusable building blocks. This thesis presented the concept of a tool that uses reusable building blocks for creating character controllers. The evaluation of the concept proved that it was a good concept and that it will be able to make it possible prototype gameplay faster by facilitating reusable building blocks.

Although successful, the project still had limitations. The biggest limitation was that the project did not tackle all aspects of gameplay prototyping. It was known on beforehand, that working with all aspects of gameplay prototyping would be a too large problem to fit the scope of a Master's thesis project. As a result, a choice was made, deciding that the target aspect should be decided alongside questions about what the building blocks should be, in the project.

To answer what the building blocks should be and what aspect of gameplay prototyping the rest of the project should focus on, interviews with game designers were conducted. The interviews gave credible results, but it is possible that the interviews could have been complemented with code analysis of older projects. This would have given a different perspective, but since the project supervisor at EA DICE said that this would be impossible or at least very difficult and time-consuming, it was decided that interviews would suffice.

The project also included an analysis of the state-of-the-art Unity game engine. The purpose of the state-of-the-art analysis was to find out what makes Unity so good for prototyping. The state-of-the-art analysis also aimed to find out if Unity has something similar to reusable building blocks and in case it has, what they are. The analysis also came to serve as inspiration, for these purposes, and inspiration, in particular, it would have been good to look into more engines than just Unity. However, due to limited time, it was not possible.

The interviews provided several options of aspects of gameplay prototyping to target, for example input mapping, character controller, UI and HUD. It would have been interesting to target more aspects that one, but the available time did not allow that. Since the interviews and current solutions suggested that it would be most useful to create something for the development of character controllers, this aspect was targeted.

A POC of a tool for creating character controllers with the help of reusable building blocks was implemented and evaluated in a controlled experiment. It would have been interesting to see how the POC performed in a real project, but it could not be done for two major reasons. The first reason is that it would have meant a big risk for EA DICE to use an untested tool in a real project. Also, since it was the first test and the nodes the tool had were limited, a synthetic test was more suitable.

The fact that the POC was not tested in a real project, poses a threat to the validity of the project. It is also possible that the task the evaluation subjects performed in the evaluation was very suitable for the POC and that other tasks would have provided another result. The game designers and programmers who used the POC suggested that it would be suitable for other tasks as well, but without testing other tasks in reality, it is not possible to know for sure.

The evaluation resulted in a number of suggested changes to the tool and the building blocks it used. It would have been interesting to implement the changes and test the tool again. However, due to the time constraints of the project, this was not done.

Chapter 9

Conclusion

This chapter summarizes the project and presents suggestions for future work and research.

9.1 Summary

This project set out to find how a set of reusable building blocks that would allow game designers to quickly prototype gameplay ideas can be composed. To find an answer to the question, interviews were conducted before a proof-of-concept was created and evaluated. The study made it possible to give an answer to the question, at least for the aspect of character controllers.

To conclude, the set of reusable building blocks are a set of nodes that are used in a tool for building character controllers. The building blocks, or nodes as they are referred to in the context of the tool, should be functionally small and general like setters and getters for character data, handles for interactables, timers, object transformers, and nodes that apply forces to the character and the game world is exists in. The tool builds the character controller to work as a state machine where actions in a state and transitions between states are editable through a visual programming language that uses the nodes.

In the perspective of game engine architecture, the building blocks are implemented in the runtime component of the game engine and the tool in the tool suite. The building blocks are implemented at a high level in the engine, in detail, the building blocks exist in the game-specific subsystems layer at the top of the hierarchical architecture the runtime component is constructed as. The character controller the tool creates is also placed in the game specific subsystems layer.

The approach proved successful. Looking at the quantitative measure-

ments of the evaluation, five out of six game designers managed to implement a double jump ability in a character controller using the tool. This can be compared to three out of six when the game designers instead used scripting to implement the double jump ability. The implementation times were statistically proved to be shorter when using the tool compared to scripting. The median value showed that both designers and programmers could, by using the tool, implement the double jump ability in nearly half the time it took them to do it with scripting.

The qualitative parts of the results were also positive to the approach. Seven out of ten evaluation subjects preferred using the tool over scripting for creating character controllers. The results also showed that ten out of ten evaluation subjects thought the approach was good and liked the idea of prototyping using the tool. The qualitative results also suggested that the concept used in the POC could be applied to other aspects of game development and provided ideas for future work and research.

9.2 Future Work

As given by the results and limitations of this project, there exists several areas of future work. The logical next step would be to implement the changes suggested in Section 8.1 and then evaluate the tool again in a more advanced test. The implementation would be a heavy task and setting up a more advanced test would also require a lot of work. Doing that work is well motivated though. The current version of the tool already received positive feedback and if the suggested changes prove successful one will end up with a powerful tool for prototyping and creating character controllers.

As suggested by the evaluation, it is likely that designing other game features to work as state machines is a good concept. Creating and testing state machine based tools for building and designing AI, quests and physical objects like vehicles and weapons constitutes three different topics for future research.

During the interviews, three interviewees said that poor UI and Head-up Display could cause gameplay prototypes to be unfairly dismissed by playtesters who, because of the poorly made HUD, do not understand the prototype. In contrast, two other interviewees said that UI and HUD can be ignored when creating gameplay prototypes because those elements are not very important. This contradiction invites anyone willing to do the research to find out who is correct.

Regardless of the answer to the question proposed in the paragraph above, UI and Head-up Display will always exist in complete games. According to

the interviews, creating UI and HUD for games is a complicated and time-consuming task. Considering these facts, there is certainly potential for creating a tool that makes this easier.

Bibliography

[1] Jason Gregory. *Game Engine Architecture*. 2nd ed. ISBN: 978-1-4665-6006-2. Taylor & Francis Group, 2015.

[2] Wikipedia. *Spacewar!* Accessed: 2019-02-22. URL: https://en.wikipedia.org/wiki/Spacewar!.

[3] Tom Wijman. *Mobile Revenues Account for More Than 50% of the Global Games Market as It Reaches $137.9 Billion in 2018*. Apr. 2018.

[4] IBISWorld. *Global Movie Production & Distribution Industry: Industry Market Research Report*. Tech. rep. IBISWorld, Aug. 2018.

[5] Joris Dormans. *Engineering Emergence Applied Theory for Game Design*. ISBN: 978-94-6190-752-3. Creative Commons, 2012.

[6] DICE. *About Us*. Accessed: 2019-02-22. URL: http://www.dice.se/about.

[7] EA. *Frostbite*. Accessed: 2019-02-22. URL: http://www.ea.com/frostbite.

[8] Wikipedia. *Video game controversies*. Accessed: 2019-03-23. URL: https://en.wikipedia.org/wiki/Video_game_controversies.

[9] Katie Allen. *Technology has created more jobs than it has destroyed, says 140 years of data*. Accessed: 2019-03-23. URL: https://www.theguardian.com/business/2015/aug/17/technology-created-more-jobs-than-destroyed-140-years-data-census.

[10] Gareth McAllister. *Video Game Development and User Experience*. ISBN 978-3-319-15985-0. Springer, 2015.

[11] Manjula et.al. "Software Engineering Challenges in Game Development". In: *International Journal for Research in Applied Science & Engineering Technology* 4 (XI 2016). ISSN: 2321-9653, pp. 555–559.

[12] Murphy-Hill et.al. “Cowboys, Ankle Sprains, and Keepers of Quality: How Is Video Game Development Different from Software Development?” In: *International Conference on Software Engineering 2014*. 2014.

[13] Osborne-O’Hagan et.al. “Software Development Processes for Games: A Systematic Literature Review”. In: *21th European Conference on Systems, Software and Services Process Improvement, CCIS*. Springer-Verlag, 2014, pp. 182–193.

[14] de Vargas et.al. “Software Engineering Processes in Game Development: a Survey about Brazilian Developers’ Experiences”. In: *Proceedings of SBGames 2016*. ISSN: 2179-2259. 2016.

[15] Aleem et.al. “Critical Success Factors to Improve the Game Development Process from a Developer’s Perspective”. In: *Journal of Computer Science and Technology* 31 (2016), pp. 925–950.

[16] Nordmark and Wang. “Software Architectures and the Creative Processes in Game Development”. In: *14th International Conference on Entertainment Computing* (2015), pp. 272–285.

[17] Almeida and da Silva. “A Systematic Review of Game Design Methods and Tools”. In: *ICEC 2013*. 2013, pp. 17–29.

[18] Pascarella et.al. “How Is Video Game Development Different from Software Development in Open Source”. In: *Proceedings of MSR ’18: 15th International Conference on Mining Software Repositories*. 2018.

[19] Musil et.al. “Synthesized Essence: What Game Jams Teach About Prototyping of New Software Products”. In: *International Conference on Software Engineering, 2010*. 2010, pp. 183–186.

[20] Charrieras and Ivanova. “Emergence in video game production: Video game engines as technical individuals”. In: *Social Science Information* (Apr. 2016), pp. 1–20.

[21] Eelke Folmer. “Component Based Game Development – A Solution to Escalating Costs and Expanding Deadlines?” In: *International Conference on Software Engineering, 2010*. 2010, pp. 183–186.

[22] Anderson et.al. “The Case for Research in Game Engine Architecture”. In: *Conference on Future Play: Research, Play, Share, Future Play*. 2008.

[23] van der Vegt et.al. “RAGE Reusable Game Software Components and Their Integration into Serious Game Engines”. In: *ICSR 2016: Software Reuse: Bridging with Social-Awareness*. 2016, pp. 165–180.

[24] Wolmir Nemitz. “Cristina: A Tool for Refactoring Source Codes of Game Prototypes into Reusable Codebases”. In: *Proceedings of SBGames 2016*. 2016, pp. 18–25.

[25] Madsen et.al. “FURG Smart Games: A Proposal for an Environment to Game Development with Software Reuse and Articial Intelligence”. In: *NDT 2012, Part I, CCIS 293*. 2012, pp. 369–381.

[26] Weintrop and Wilensky. “Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms”. In: *Transactions on Computing Education* 18 (2017), p. 25.

[27] Maloney et.al. “The Scratch Programming Language and Environment”. In: *ACM Transactions on Computing Education* 10 (2010), p. 15.

[28] Cooper et.al. “Alice: A 3D Tool for Introductory Programming Concepts”. In: *Journal of Computing Sciences in Colleges* (2000).

[29] Resnick et.al. “Scratch: Programming for all”. In: *Communications of the ACM* 52 (2009), pp. 60–67.

[30] Powers et.al. “Through the Looking Glass: Teaching CS0 with Alice”. In: *ACM SIGCSE* 39 (2000), pp. 213–217.

[31] Haeberli. “ConMan: A Visual Programming Language for Interactive Graphics”. In: *ACM Computer Graphics* 22 (1988), pp. 103–111.

[32] North and Shneiderman. “Snap-together Visualization: A User Interface for Coordinating Visualizations via Relational Schemata”. In: *Proceedings of the Working Conference on Advanced Visual Interfaces*. 2000, pp. 128–135.

[33] Kim and Jeon. “Programming LEGO Mindstorms NXT with visual programming”. In: *International Conference on Control, Automation and Systems*. 2007, pp. 2468–2472.

[34] Unreal. *Blueprints Visual Scripting*. Accessed: 2019-04-12. URL: https://docs.unrealengine.com/en-us/Engine/Blueprints.

[35] CryEngine. *Introduction to Flow Graph*. Accessed: 2019-04-12. Jan. 2018. URL: https://docs.unrealengine.com/en-us/Engine/Blueprints.

[36] Frostbite. *Frostbite Documentation*. Accessed: 2019-04-02.

[37] Manker and Arvola. “Prototyping in Game Design: Externalization and Internalization of Game Ideas”. In: *Proceedings of HCI 2011 - 25th BCS Conference on Human Computer Interaction*. 2011, pp. 279–288.

[38] N. Gijsen. “Prototyping and Feedback Design in a Serious Game Context”. In: *7th IBA Bachelor Thesis Conference*. 2016.

[39] Jon Manker. “Designscape – A Suggested Game Design Prototyping Process Tool”. In: *Journal for Computer Game Culture* (2012), pp. 85–98.

[40] Gamedesigning.org. *The Top 10 Video Game Engines*. Accessed: 2019-03-25. URL: https://www.gamedesigning.org/career/video-game-engines/.

[41] Juan Linietsky. *Tweet from 28 Jan 2018 about Game Engines used in the Global Game Jam 2018*. Accessed: 2019-03-25. URL: https://twitter.com/reduzio/status/957837903425568769.

[42] Mono Project. *Scripting an Application*. Accessed: 2019-05-10. URL: https://www.mono-project.com/docs/advanced/embedding/scripting/#scripting-an-application.

[43] William Adams. “Conducting Semi-Structured Interviews”. In: *Handbook of Practical Program Evaluation*. 4th ed. ISBN: 978-1-119-17138-6. Wiley Blackwell, 2015, pp. 492–505.

[44] A.S. Cheng Kwang-Ting Krishnakumar. “Automatic Functional Test Generation Using The Extended Finite State Machine Model”. In: *30th ACM/IEEE Design Automation Conference*. 1993, pp. 86–91.

Appendix A

Interview Guide

A.1 Reminders for Interviewer

- Explain that any confidential information mentioned during the interview will not be published. It is okay to describe confidential information during the interview, but avoid explanations about themes and settings as this will likely be removed anyway.
- Try to steer answers away from being related only to creativity.
- If the interviewee has experience from different prototyping tools, get them to explain what has been good and bad in the different tools.

A.2 Questions

1. Could you describe any prototyping project you have participated in? Describe the project, your role, challenges, and things that worked well. It would be especially interesting if you could describe a project that you think worked extra well.

   In your general experience, not excluding the project you just described,

2. What are common activities when developing gameplay prototypes?

   (a) What are common activities done by a computer, either in a group or by oneself, that are especially time-consuming?

3. What are some of the biggest challenges in the development of gameplay prototypes?

   (a) What are some of the biggest software and implementation challenges?

4. How have tools hindered you in the development of gameplay prototypes?

   (a) What tools have hindered you?

   (b) Is it because they are difficult to use or do they miss functionality?

   (c) If they miss functionality, what functionality is missing?

5. What has been working well in the development of gameplay prototypes?

   (a) When implementation of new ideas has been easy, what made it easy?

   (b) When communication of new ideas was understood by everyone in the team, what tools and methods were used to express the ideas?

   (c) When cooperation in the team was good, what factors were involved?

6. To what extent are code and content from older project reused in the development of prototypes?

   (a) What kind of code and content is reused? What is not?

   (b) What makes code and content reusable?

   (c) Is there code and content that could be reused in principle but the current tools make the effort too big to be worth it?

7. Anything you wish to add?

TRITA EECS-EX-2019:331