September 10, 2018

Extending a game engine with custom tools

Samu Kovanen

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo September 10, 2018

Thesis supervisor and advisor:

Prof. Perttu Hämäläinen

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Samu Kovanen

Title: Extending a game engine with custom tools

Date: September 10, 2018 Language: English Number of pages: 4+40

Computer, Communications and Information Sciences

Professorship: Computer Games

Supervisor and advisor: Prof. Perttu Hämäläinen

Video games are primarily made using game engines nowadays with an ever increasing abstraction on the details of individual components. The research on the software development methods, software architectures and team and project management is a vast area of interest and has been applied to game development topics widely. However, there has been less focus on how to utilize and manage the individual game development components from the perspective of the entire development team and the creative process.

In this thesis the game development is examined through individual game development components called tools. A definition of a tool is presented and their usage and presence in different popular game engines is explored. The tools are categorized to built-in, 1st party and 3rd party tools and their benefits and use-scenarios are compared against each other. In addition the thesis presents and adapts a "Tools Focused Development" methodology, which proposes a set of guidelines that aim to improve the possible benefits gained from developing and utilizing tools during development.

The thesis also analyzes several built-in, 1st and 3rd party tools of a popular Unity game engine in the context of tools focused development, and their features and limitations are documented from the usage and management perspective. Finally the thesis reflects on how tools development was present in the development of a critically acclaimed commercial game Bucket Detective, and how their use affected the final completed game. The tools were found to be the source of emergent game design, and improved non-programmer participation in creating content along with making adapting to design changes easier from programmer standpoint.

Keywords: game engines, game development, game development tools, game design

AALTO-YLIOPISTO
PERUSTIETEIDEN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Samu Kovanen

Työn nimi: Pelimoottorien laajennus mukautetuilla työkaluilla

Päivämäärä: September 10, 2018 Kieli: Englanti Sivumäärä: 4+40

Game Design and Production

Professuuri: Tietokonepelit

Työn valvoja: Prof. Perttu Hämäläinen

Työn ohjaaja: Prof. Perttu Hämäläinen

Videopelien kehitys tehdään nykyaikana pääasiassa käyttäen pelimottoreita, mikä on nostanut yksittäisten komponenttien ja yksityiskohtien abstraktiotasoa. Ohjelmistokehitysmetodit, ohjelmisto-arkkitehtuurit sekä tiimi -ja projektihallinta ovat olleet laajoja tutkimusalueita pitkään ja niiden tuloksia on sovellettu pelikehitykseen monella tapaa. Yksittäisten pelinkehityskomponenttien hyödyntäminen ja käyttö ovat kuitenkin jääneet vähemmälle huomiolle koko kehitystiimin ja luovan prosessin näkökulmista.

Tässä diplomityössä pelien kehitystä tutkitaan "työkalujen"kautta, joilla kuvataan yksittäisiä pelinkehityksen komponentteja. Työssä esitetään määritelmä työkaluille ja niiden käyttöä ja laajuutta selvitetään suosituissa pelimoottoreissa. Työkalut kategorisoidaan sisäänrakennettuihin, ensimmäisen ja kolmannen osapuolen työkaluihin ja näiden hyötyjä sekä käyttöskenaarioita vertaillaan toistensa välillä. Lisäksi työ esittelee ja soveltaa "työkaluihin perustuvan kehitystavan", joka on joukko ohjesääntöjä, joilla pyritään parantamaan työkaluista saatavaa hyötyä ja helpottamaan niiden kehitystä.

Työssä analysoidaan myös muutamia sisäänrakennettuja, ensimmäisen ja kolmannen osapuolen työkaluja Unity pelimoottorissa työkaluihin perustuvan kehitystavan kontekstissa, sekä dokumentoidaan näiden ominaisuudet ja rajoitukset myös käytön ja hallinnan kannalta. Lopulta työ käsittelee miten työkalut olivat käytössä kaupallisen Bucket Detective -pelin kehityksessä sekä miten niiden käyttö vaikutti lopputulokseen. Työkalujen käytön huomattiin tuottavan uusia suunnitteluideoita, parantavan muiden kuin ohjelmoijien osallistumista sisällöntuotantoon sekä helpottavan ohjelmoijien mukautumista suunnitelmien muutoksiin.

Avainsanat: pelimoottorit, pelikehitys, kehitystyökalut, pelisuunnittelu

Contents

1 Introduction

Software development is an inherently evolving field with new applications built on higher abstraction levels every passing day. Increasing the level of abstraction is a natural consequence of the human nature to avoid work, and therefore it's reasonable to expect this trend continue to the unforeseeable future.

Game development is a field of software development which has seen this same trend over the course of history. Most games are unique pieces of art which are made to represent something the creators want the players to experience first-hand. But while the experiences games attempt to convey are often completely unique, the technologies that build them up are based on common software foundations. Studying and developing these foundations to achieve the end-result with the highest efficiency has therefore become one of the core focuses of game development.

This thesis discusses developing various custom extensions and tools for game engines during game development. Most modern games are built on game engines, which are pieces of software that provide various high-level tools for developers and designers to use. These built-in tools provided by the game engine alone are rarely enough to materialize the unique visions and ideas for games, so using either 1st or 3rd party tools to augment the engine feature set becomes a necessity.

How the developed tools are utilized in game development is heavily dependent on the developer team structure. Larger game studios generally have specified developers who focus solely on developing 1st party tools for the studio to use. Smaller studios on the other hand tend to avoid developing 1st party tools due to the increased development cost and instead focus on utilizing 3rd party tools or developing problem-specific solutions.

This thesis takes a look at various tools using a *tools focused development methodology* that aims to improve the potential benefits of using tools within game development and make it easier for programmers to develop tools using a set of game engine agnostic guidelines. The thesis provides a case study of extending a widely used Unity game engine, discussing the limitations of Unity's built-in tools and features and the benefits and challenges of extending the Unity engine with both 3rd party and 1st party custom tools.

Finally, the thesis discusses the results of developing custom tools for *"Bucket Detective"*, a commercial game created by myself and Jesse Barksdale. The game development phase featured various 1st party custom Unity tools such as a physics based character controller and physics interaction systems, which contributed to realizing various emergent game features during the development, which ultimately reinforced the positive public reception of the game.

2 Background

2.1 Previous research

The research on games development has seen a growth recently due to games' increased socioeconomic effects in the modern world. Large portion of the research is centered around adapting the software development methods from traditional software development to fit games development. Several studies [8] [6] have found scrum based agile methods to fit games development well due to the small sprints proving to be effective for testing prototypes and features. This kind of momentum shift towards more agile methods has started 30 years ago from the modified waterfall methods[9] that were found to suit actual customer feedback loops better. However, recent studies[10] tend to conclude that the choice of the software development process depends largely on the project itself, which of course is a natural outcome when looking at software on the whole. For games development projects though, the overall trend seems to be towards agile methods working out the best.[11]

Engine tools development has been a central part of game development since the early days. Jonathan Blow wrote[4] about the discrepancy of game development tools and software tools in 2004, describing how the external tools used are not exactly great for developing games. His points still hold true to today, although Blow himself and recently the Unity engine have started to do work towards creating compilers that focus solely on games development. Most notable external tools' difference to today are the reduced cost of external tools and game engines themselves, with many external software providing low-cost options that even small self-published developers can afford.

The benefits of developing in-engine tools have also been noticed since long ago. The developers at Turbine focused heavily on engine tools creation and found them to be useful for the whole development phase. [5] They cite improved iteration speed and deeper involvement of artists as important benefits of making in-house tools. They also found there to be issues with the tools' quality assurance and documentation, but overall the effort spent on developing in-house tools paid off. Other developers have also proposed[1] a portable architecture for developing tools within multiple engines, echoing the usefulness of re-usable and modular components.

Game development has seen overall little love from the FOSS (Free and Open Source) communities. Scacchi has written[7] about the role of FOSS within game development, but there seems to be no major advances within this field to the date. While there are multiple open source game engines available, currently the most competitive ones are still considered commercial and closed source. This could be due to how Jonathan Blow also mentions there's a large amount effort and expertise required for developing games, which contributes to the need to focus more on the real-world business side of making content and assets.

2.2 Modern game development

Before mid 2000's, creating games was considered to be a process of first creating your own game engine and then using that to build the actual game on top of it. [4] With the release of new professional openly available game engines like Unity, Unreal and Source, the game creation process started to move towards expanding these available engines instead of building your own from the scratch.

The release of openly available game engines has meant that many of who had only dreamed of making games can now produce tangible results with considerably less effort. This time can be called the beginning of democratization of game development, which has lately brought upon many fundamental changes to the field. Most importantly, the democratization has nowadays skyrocketed the amount of released games on multiple gaming platforms, making the competition fierce and the amount of openly available game development resources, materials and tools higher than ever.

The rising popularity of game development has made many large studios to also turn to using openly available engines[2], as commercial engines accelerate the development process by providing many large features like cross platform support pre-implemented. Trying to stay competitive with an in-house own engine is also proving to be ever more difficult due to most game engines' technology stack having developed internally so far, that trying to implement all modern engine features from the scratch is proving to be infeasible except to the largest game studios with their own sizeable engine teams.

2.3 Game development using game engines

Game engines are software environments that provide developers with a collection of tools to handle complex features, such as asset management, rendering pipelines, physics or graphical features. These tools enable developers to achieve common game creation tasks fast and allow developers to focus more on implementing their own game specific features and mechanics. The extent to which game engines provide these tools varies wildly, with some engines only providing modular standalone code components and others providing a full visual editing environment with many game genre specific features pre-implemented. Some game engines are made to suit some use-cases better than others, and it's not feasible for any engine to provide a feature set or abstraction level that would fit all possible game projects.

Game creation software environments which are more oriented towards non-programmers are known as Game Creation Systems, while programmers often work with the more complex Game Engines. There is often a trade-off between the amount of freedom in customizing the engine and the amount of features it provides. For example the RPGMaker game creation system is solely made for making 2D RPG-genre games, while the Cocos2D game engine is not the most feature-rich engine to use for

making 3D games. Understanding these engine limitations is of utmost importance for developers, as trying to complete tasks that the engine is not designed for tends to require a lot of extra work. When making features that challenge the limits of the engine, utilizing 1st and 3rd party tools can be immensely helpful.

2.4 Game development tools

One way to look at game engines is that they're simply a collection of tools that help achieve implementing a specific game feature. Be it an editable 3D view of the game world, automatic asset importer, a math library or the physics engine, all of these can be considered to be game development tools. This thesis defines *game development tool* as the following:

> *Game development tools are software components that provide various re-usable customizable functionalities*

The definition is an adaptation of widely known component based software engineering to the field of game development. The definition generalizes software component interfaces as functionalities, and also implies encapsulation similarly to components in the sense that the user of a tool does not have to know it's inner workings. The main difference of game development tools and traditional software components can be found in the tools' requirement for customizability.

Indeed what makes a tool is how they're meant to provide customizable features. Customizable in this context is defined as the output or the effect of the tool changing depending on developer input or the running environment. For example, a script to explode the player character when the player touches a red coin is not a tool, because it does a single, specific feature that is not customizable. However, if the script was changed to cause a modifiable effect when a given type of unit touches the entity the script is connected to, it would become a tool within the definition.

Using this definition it naturally follows that virtually any single modular part of a game engine can be considered to be a tool. The definition also does not exclude external programs, for example 3D modeling software from the scope of the definition. This makes sense, because if someone implemented for example a feature-rich 3D modeling tool inside a game engine, fundamentally it wouldn't differ from the functionality of using an external software.

There are many types of tools used in game engines. Some tools have their own UI, while others are used via programming interfaces. Modern game engines have invested heavily into increasing the amount of available tools by increasing the number of built-in tools and also by including asset marketplaces into the engine ecosystem.

Overall there are 3 types of tools used in game engines:

- Built-in tools
- 1st party tools
- 3rd party tools

From these, built-in tools mean the tools available integrated into the game engine itself. These tools usually do not need to be installed separately, are generally easy to use and are meant to be the basis of all other tools or game features.

1st party tools are tools developed by the makers of the game themselves. They're tools that implement more specific or improved features for the game or the engine than the built-in engine tools provide. These tools tend to not be as finalized or easy to use as built-in engine tools, as they're usually created for the developers of the game and not the general public.

3rd party tools are tools developed by other unrelated 3rd party developers. These tools often provide similar functionalities as engine tools, but differ in that their quality and support varies greatly. Some 3rd party tools are commercial and require a purchased license to be able to use them in the game.

2.5 Using tools for developing games

The wide definition of a tool means they're used in many different ways when developing a game. In general their use in development is more about adapting the mentality of creating and using tools to solve problems, rather than just attempting to solve the problems directly. It's possible to have tools for multiple different purposes, for example ones that solve a specific problem, tools that improve/optimize an existing feature or workflow or simply tools that allow the customizability to provide new inspiration for ideas.

The integration of tools into the development process can be done in different ways. Larger game studios tend to have specialized Tools programmers, whose main purpose is to write tools for accomplishing different tasks and features. These tools are then used by other developers or designers within the studio to implement the required functionality for the game. This kind of work split is especially useful for experimenting with the potential of tools and enabling coming up with new emergent features for the game even late during the development.

Smaller game development teams tend to focus less on creating tools and instead solve the problems directly. This is often due to lack of development workforce, due to the team not seeing the potential in re-usability and customization of the tool or due to the smaller scale of developed games. When the developers would likely end up using the tools themselves it's difficult to reason the slight level of indirection in the feature implementation over a specific solution.

Finally it's also possible to simply do a compromise in the game design to make the design suit already existing and available tools instead of carrying out the implementation of a new feature. This is especially useful if the feature is insignificant compared to the required effort or if no 3rd party tools are available for solving the issue. Some compromises are often inevitable during the development of a game, but at wrong places they can lead to various problems for the finished game. This thesis does not discuss the further implications for the game design or team management of doing these decisions.

3 Game software architectures and development processes

The choice of game engine for developing a game is often a sum of many factors. Different games and development teams have different requirements for game engines, and some engines are more suited for specific software development models than others. The most important factors when deciding the engine choice are usually the size of the game and the familiarity of the development team with the engine.

It's possible to categorize game engines to different engine types based on how much they abstract out the complex details of development. The engine abstraction is often most importantly implemented via the selection of engine programming language, the engine feature set and how deep access the engine allows to developers. We can define more abstract game engines as "High-level engines", while engines that abstract the least details can be defined as "Low-level engines".

It's not possible to categorize an engine explicitly as high-level or low-level, as the standards of abstraction in software development and the software itself change over time. An example of this is the widely known programming language C++, which was once considered to be a high-level programming language, but nowadays is considered to be close to hardware low-level programming language.

3.1 High and low level engines

High and low level engines both have several trade-offs game developers need to take into account when selecting an engine. Increased development time from more complex programming languages, paradigms and architectures are often the main contributors to the overall feature implementation cost. However, abstracting too much out can also be detrimental to productivity, as trying to implement low level game features in a high level engine may require bypassing the engine or programming language limitations.

High level engines often provide scripting interfaces for high-level programming languages. This allows developers to write their game code in high level languages

that abstract away some programming details, usually reducing bugs and improving productivity for simple tasks. [12] The increased level of abstraction on engine features can also make it easier for non-programmers to contribute their work directly inside the engine, reducing some of the workload on programmers.

Low level engines on the other hand often provide their scripting interfaces in low-level programming languages. In addition they tend to allow highly granular access to the engine feature set and source code, making it possible to tweak, modify, fix or optimize the engine features directly. This kind of increased control comes at the cost of increased development time and expertise requirements, although some studies have criticized the research on the effects of different programming languages on productivity[13]. However, when working in game engines the use of low-level programming languages comes with responsibilities that require more advanced programming techniques and demand more focus on software architectures to keep the codebase maintainable.

High and low level engine features directly translate to their applicability to different software development models, which are used to guide the software project's life cycle and manage the development resources. Game development usually doesn't follow traditional software development models in a verbatim manner due to the increased focus on game feel over well defined project requirements. Still, knowing how to apply the principles of different software development models to different game engine workflows becomes increasingly important as team and project sizes grow.

3.2 Development models in low level engines

The increased feature implementation cost is commonly the biggest factor to take in account when developing using a low level engine. This thesis examines the widely used Unreal Engine, which as of writing could be considered a low level engine. Unreal Engine is a commercial open-source engine written in C++ which features modern advanced rendering capabilities and various higher level constructs inside the engine such as the Blueprint visual scripting system.

Feature development in Unreal is generally recommended to be split between it's Blueprint visual scripting system and writing C++ script files. This means that more abstract concise features such as "Open Door" could be handled as single visual nodes in blueprints, while the actual logic for the nodes could be handled in C++ scripts. This kind of development split implicitly encourages the creation of re-usable blueprint nodes in C++, but the added level of indirection in feature implementation may become difficult to grasp for less experienced developers.

Feature implementation cost of Unreal engine stems from this split. Writing C++ code introduces layers of complexity to programming that are not present in high level programming languages. Manual memory management and the complex language specification of C++ are common bug hazards[14] that risk increasing feature

development time. Combined with the longer iteration times from compiling C++ code this means programmers need to have well thought game design and script architecture design laid out before it's wise to actually implement a feature. This slightly resembles the methodology of the Waterfall development model, where the development usually proceeds from defining requirements to implementation to integration and testing.

3.3 Development models in high level engines

In high level engines the feature development cost is generally small for simple tasks that fit the engine's feature set. Historically the Unity game engine has been considered a high level engine, although recently it has started to morph it's feature set into lower level territory. This originates from changes like removing support for alternative higher level programming languages, opening more of the engine's source code to the public and implementing features that work close to the hardware like custom render loops and supporting data-oriented programming models.

Compared to the low level Unreal the primary feature implementation methods in Unity are still more abstracted, which is the reason it can be considered a high-level engine. The C# programming language used in Unity doesn't require manual memory management and has considerably less complex language specification than C++. This means more programmers can develop features for games as long as the features are suitable to be implemented in a automatically memory managed high level language.

If the game design requires features that are hard to translate to the context of a high-level engine or programming language, the feature implementation cost of high level engines can increase tremendously. For example, Unity engine did not support low level access to the rendering code until very recently in 2018, which is over 10 years after the initial engine release. This made it very hard to create games that are graphically competitive with commercial custom engines and other low level engines, due to not having detailed access to basic features like shadow mapping or rendering path code.

Among the more abstracted programming languages high level engines often also provide many common features and tools pre-integrated to the engine, which can make iterating simpler tasks fast. Being able to iterate features fast by more programmers is directly translatable to agile development methods, which emphasizes fast iterations, bug-free software and quickly responding to the ever changing project details. Agile methods are popular within the field of modern software and game development and their effects are widely documented. [6] [8]

3.4 Developing tools in high and low level engines

High-level engines do not implicitly promote tools-based game development. This is because of their highly agile nature and generally smaller feature implementation cost, it's more attractive for developers to create quick specific implementations rather than spend more resources on designing and creating a re-usable customizable tool. There are several consequences to this, which can be considered to be unique to game software projects.

It's common that games are developed for a customer client, but it's also common that all the features of the game are not directly documented in the customer's project documentation. For example, it's possible that the game design addresses the issue of character moving around and the environment type, but doesn't consider the possibility to interact with physics or does it have to support walking on an uneven terrain with small obstacles.

This means it's very attractive for developers to develop a solution for the very specific design document problem, instead of thinking about the feature as a possible source of more gameplay features. If the development process was tools focused instead, creating a character controller would be more about making a customizable, re-usable character controller that is then simply adapted to work for the scope of the design. This would fundamentally change the problem scope for developers, making them instinctively think about corner cases and more generalized solutions. This in turn would bring more ways for other team members to experiment and iterate with the concept, making it more likely for emergent game features and new ideas to pop up. Bringing this kind of mentality to high-level engines requires more conscious effort from the developers, who must weigh the possible potential of the feature against the increased implementation cost.

Low-level engines like Unreal have larger tendency to promote tools based development. For Unreal this is largely due to the work split between their Blueprint visual scripting language and native C++ code, which naturally drives programmers to create C++ solutions that they themselves or designers use from blueprints. Creating gameplay functionality in Unreal has higher implementation cost than creating gameplay in an engine like Unity, simply from the increased architectural complexity and code compilation times. This increased cost can automatically force developers to think more about creating logical modular pieces of code that can be reused around the project and even tweaked by other non-programmers from the blueprint system.

4 Developing a toolset for game engines

Adapting the mentality of developing tools to solve problems over developing specific solutions is a choice akin to applying a software development process to a software project. The difference comes from the fact that for tools this selection can easily be done on a per feature basis, instead of deciding on the whole game project. In most cases there are no meetings held or collaborated code design phases for programmers when developing new features for a game project. Instead it's the programmers responsibility to select the correct methodology for implementing a feature.

4.1 Reasons to develop tools

When developing a game it's common to think about using the built-in tools of the engine to the fullest extent when thinking about features. This is what often gives certain game engines their characteristic feel to the graphics and common features. For example, as of writing it's easy to spot a 3D game made with the Unity engine, as very nearly all Unity games use the same cascaded shadow map technology in their real-time shadowing.

Pursuing uniqueness is one of the core principles of game development, and developing a custom toolset for a certain engine can give game studios their own unique characteristics to their games. For example John Nesky has talked about the significance of details like how game cameras use whiskers to avoid line of sight issues are in games. [15] These small, nearly invisible features like how your character behaves when you run into a wall are features that are likely to be reused from game to game and indirectly reinforce the studio's brand.

Built-in engine tools are made to be customized to the fullest extent, which can allow even non-programmers to create unique content with them during the development process. Applying this same level of customizability and ease of use to 1st party developed tools can potentially have the same effect, making it possible to experiment with combining features completely unrelated to the original game design. This way tools have the potential to reduce the workload of programmers and improve the final game features due to increased iterating and overall experimentation.

4.2 1st and 3rd party tools

Tools focused development processes do not always have to incur the increased development cost, as the implicitly modular nature of tools allows easier outsourcing of the work. Modern commercial game engines have realized the potential of outsourcing tool development just like assets have been outsourced since the early days of game development. This is visible with the growth of various 3rd party tools within game engine marketplaces.

The outsourced 3rd party tools have recently boosted the productivity and output quality of game development teams significantly. These 3rd party tools often suffer less from the engine-characteristic nature of built-in tools, and typically their usage is difficult to notice in the final game. Since the tools are often sold at a certain engine's marketplace, they tend to solve specific issues the engine might have with the built-in tools or provide features that the engine is completely missing. However, the 3rd party nature of these tools often necessitates various testing and quality assurance procedures to see if they are fit for the particular project.

Using outsourced 3rd party tools in game development has allowed even non-programmers to step into the realm of game engines from game creation systems. The wide variety and feature rich selection of 3rd party tools at game engine marketplaces has made it even possible to assemble complete games without any programming required, albeit this usually comes at the cost of making compromises in the game design. Generally the quality of these games is often considered to be subpar to games with dedicated programmers, but it underlines an important paradigm shift in the level of abstraction within game development field.

Nearly all unique game designs still require dedicated programming to get the required features implemented. Whether to use 1st party tools or 3rd party tools for a specific feature implementation requires pre-research on available 3rd party tools. In many cases there is little reason to develop a custom solution if a suitable 3rd party solution can be found. As the number of available 3rd party tools and their quality increases, it's interesting to consider what is their role in game development in the future.

4.3 Developing 1st party tools

Developing customizable reusable 1st party tools takes more time than writing out a specific solution to a problem. If the programmers working on tools are employees, the direct monetary costs of producing results this way is often high. In addition there is often a large amount of risk variance, as the development cost is fully dependent on the feature itself and programmer experience.

The cost doesn't come without benefits though, as completed tools are often valuable assets as is. The 1st party tools usually solve game design problems exactly with no compromises on the design-side, and their re-usable nature and usage benefits that reach non-programmers can bring valuable new features, ideas and productivity to the development process. In addition to this, 1st party tools are easiest to extend, update and debug, as the developer of the tool is usually locally present at the development team. As game engines go through new versions and the design evolves, these traits are extremely valuable to the completion and quality of the game.

It follows that the downside of developing 1st party tools is that if the programmers who implemented the tool leave the team, the tool may be hard to maintain. 1st party tools can be less documented, less extendable and harder for other programmers to pick up in case the tool requires more development or has to be extended. However

it's possible to mitigate these issues with heavily modular, self-contained tools development.

4.4 Using 3rd party tools in production

The cost of developing 1st party tools is the main reason many game projects desire to utilize 3rd party tools over 1st party ones. Not only this, due to the 1st party tools' heavy reliance on programmer expertise, sometimes 3rd party tools outperform or have higher quality and customizability than 1st party tools have.

Still, applying myriads of 3rd party tools to a game project is definitely not a silver bullet, as in the end they tend to create unnecessarily large and complex development projects. Many 3rd party tools often provide many more features than just implementing the feature they were originally applied for, meaning that the codebase and the game project become littered with features and functionality that is not used anywhere. Stripping these features out of the tools is often difficult, and this dead code may create difficult issues when the engine or the tools themselves are updated.

It's also common that 3rd party tools are heavily commercialized. The current trend is that the cost of these solutions is going down due to the increased supply, but high-end 3rd party tools still cost large amounts of money or require royalties that are out of scope for smaller game studios, or may become significant expenses for larger game studios. The way 3rd party tools are developed to appeal to most possible customers means that they might not implement the exact design the game design document specifies, instead opting for the game design to adapt into the capabilities of the tool instead.

3rd party tools rarely give access to the source code, meaning all the vast benefits of open source development are lost. This means that extending 3rd party tools or fixing their bugs becomes nearly impossible for game developers, and trying to get the original developers to include suggestions or act on bug-reports may take extremely long time. 3rd party tool licenses don't generally consider what happens to them in case the tool becomes abandoned by the original developers, meaning that projects relying heavily on certain features might face unexpected potentially even fatal roadblocks in development.

Even with all these fallbacks, outsourcing implementation of game design to utilizing many 3rd party tools has definitely found it's place in modern game development. Some complex features that were often programmed by game studios themselves are nowadays nearly universally outsourced to a 3rd party tool. Physics engines are one of the earliest examples of this, with popular engines like Nvidia PhysX, Microsoft's Havok and open source Box2D or Bullet Physics finding their way to nearly every game utilizing physics that ships nowadays.

5 Tools focused development

5.1 Methodology

In this thesis I have adapted a methodology I like to call *Tools focused development*, which I've found to help programmers achieve the various benefits behind developing game engine tools. It's not a programming architecture because the way it's applied depends on the implemented feature, the game engine used and the programmers' way of writing code. However it is an engine-agnostic set of guidelines which can affect the way some of the game code is structured in a concrete way.

When creating new features for a game or utilities for the game engine, the programmers working on the problem have to choose a method to implement the feature in question. At this point the workflows of different people start to diverge radically depending on their experience, the engine they're using and whether or not they're following any software development architectures in their code. It's also at this point when the tools focused methodology can begin to help the overall development process and decision making. Creating tools shouldn't be a rigid architecture one needs to follow to succeed, but instead an implicit way to encourage the customizability and modularity of the code via structural guidelines.

The methodology adapted in this thesis can be broken down to the following 3 points:

- Maximize encapsulation
- Minimize setup
- Favor customizability

When these 3 guidelines are applied together they can spontaneously help the game features programmers are working on to turn out to be useful tools. Alternatively, one can examine already completed features from the guidelines' perspective and see potential issues or improve their usage within game development environment. To make useful tools one usually must satisfy all 3 guidelines, or else the features instantly lose out on a large portion of the benefits of tools focused development.

Many of the principles of tools focused development can be attributed to applying well studied programming paradigms to game and game engine development, but the different use scenarios of various tools within the field of game development mean it's useful for programmers to have a certain kind of mindset when developing tools to fully utilize their possible benefits.

5.2 Maximizing encapsulation

Encapsulation within programming languages is considered to be a sign of easily portable, modular code[16], and the same principles apply to game development and developing game engine tools. In it's core encapsulation is all about separation of functionalities by making modular pieces of code that do their work on a well defined set of data. When applied to tools focused development and game development, encapsulation often comes down to programming game scripts that do one thing well, and contain their functionality and working data inside the script.

Game scripts often have to work on the scene graph, or the scene hierarchy, to achieve some kind of visible effect to the game world. This often implies creating code that goes far out of it's location in the hierarchy to do something on separate elements. Very commonly this happens with elements that contain sub-elements that represent some smaller part of the whole element. For example, it's common for a UI button elements to contain various sub-elements that might define the button text, the visible graphic or the interactable area.

Applying encapsulation to these kind of elements implies encapsulating both the element's scene hierarchy and the code that operates on it. In practice this comes down to making self-contained elements that contain all their functionality on a top-level element, and any sub-elements are managed under this top-level element. The code that works on the whole element should be placed in the top-level element, and it should aim to work only with elements that are it's children in the scene hierarchy. Constant restructuring and rearranging of elements in the scene graph is common within game development, and this makes it possible to change the location of the top-level element in the scene graph without affecting it's self-contained functionality. If the code interfaces with content outside the children of the top-level element, it automatically loses it's encapsulation status as it doesn't self-contain it's functionality anymore.

When the code and the scene hierarchy for the top-level element are structured in a self-contained modular way, it's easy to remove, add or change various sub-elements or functionalities from the whole element. This kind of flexibility allows the elements to be re-used in multiple different projects and contexts with little effort, already achieving one of the defined goals of tools focused development. Making highly encapsulated features contained within modular elements is also immensely helpful for being able to achieve the second guideline of the methodology, which is minimizing setup.

5.3 Minimizing setup

Software engineers and game developers commonly work with various software that needs to be set up in a certain way as documented by the creator of that software. Tools within game engines should generally avoid this kind of prerequisite work, to make it also possible for non-programmers to use the developed tools without stress of breaking something or any other unneeded mental load. Games are inherently creative arts, and likewise it wouldn't work for example for the benefit of a painter, if he had manually assemble his paint brushes every time he wanted a different result on the canvas.

Game engine tools should aim to work in a plug & play manner: be quick to apply and to remove. Most often for implementation this already implies well defined encapsulation within the tool, so that the functionality of the tool doesn't depend on something else being done or present beforehand. Applying this in game engines would mean avoiding certain commonly used ways to define references within code, namely hard coded references, manually assigned field references or using names as references. Unfortunately avoiding all of these becomes extremely hard on a larger scale, as smart management of object references is one of the most important goals of various widely applied architectural programming paradigms.

Instead of simply choosing the least bad solution, tools focused methodology encourages efficient use of the scene hierarchy to manage references, as it can already work as a structure for encapsulating features within elements and their sub-elements. Most commercial game engines implement efficient serialization of specified hierarchy trees and their inner references as re-usable objects called prefabs. By reusing these serialized small parts of hierarchy the programmers can restrict their code scope to work within the bounds of that prefab and achieve strong encapsulation by doing their assumptions within the scope of that prefab. For example, it would be possible to serialize a treasure chest element as a prefab as seen in figure 1, and then make an assumption that if the element has a sub-element named "Particles", it would always contain the particle emitter component of that chest element.

Going further, it's possible to make the top-level element procedurally generate the sub-elements it needs on-demand. This would enable the ultimate encapsulation, as the whole functionality of the tool can be contained within a single top-level element and it's code. The element would not even have to have sub-elements it depends on, making the pattern work on engines without a prefab system or efficient access to the scene graph too. Containing the sub-element creation within the top-level element makes it easier to hide reference management from the end user of the tool, makes it harder to break the tool by changing the scene hierarchy and also allows parametrizing the important values within a single script for easy modification.

Whatever solution is used, the end goal of minimizing setup can be achieved by clever utilization of the scene hierarchy, and it's also possible to make the code more resilient to end user changes and hierarchy errors. The primary idea here is the separation from traditional modularity of functionalities within scripts to modularity

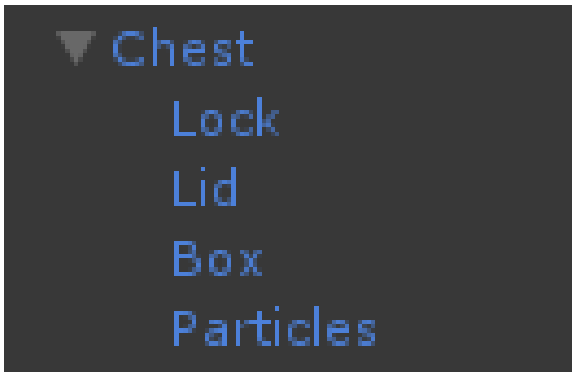

Figure 1: A part of the scene hierarchy, with the Chest being a prefab top-level element that contains all the other elements. The structure inside the prefab is static and self-contained, which means the programmer can safely make assumptions about the sub-elements in the top-element while still keeping their code encapsulated within the scope of the prefab.

of concerns of larger entities instead. The modular piece, the tool, should should be modular in regards to it's concerns, meaning that whatever global effect it may do within the engine, it should only depend on the things it manages itself. This kind of behavior makes it easy for programmers to create a single point of customization for the tool and making the tool easiest to use for non-programmers.

5.4 Favoring customizability

The final part implementing a useful tool requires is customizability, and this is also where most of the perceivable benefits of tools focused programming can realize. *Customizability* means the end effect of a tool changes depending on it's parameters or running environment. This means the tool must have tweakable parameters for the end user or provide a different meaningful effect in a separate running environment. For example, allowing changing the color of a UI button easily when looking at the buttons as tools would satisfy this requirement for the button. Alternatively, making a movement controller script that provides useful movement functionality in many different kinds of terrains would also satisfy the requirement for customizability.

In other words, *customizability* could be considered to be a trait of how wide spectrum of use cases the tool supports. This implicitly enables the use of the tool in different contexts, making it more likely for developers to come up with new gameplay or other interesting outcomes just from "thinking outside of the box" with the tool. It's for this same reason programmers should avoid constricting the use cases of the tool by for example limiting possible parameter value ranges or doing other compromises in flexibility of the tool on the code side. For example, as seen in the figure 2, the developer can make decisions that limit the usefulness and possible emergent features of the tool simply by changing the UI of the tool a tiny bit.

Figure 2: Two different ways of implementing an UI for modifying a parameter. The bottom one restricts the values to the range 1-10, which prevents developers from even testing out unexpected outcomes that could happen with for example a value of 100.

For the end user, the UI for controlling the tool should be maximally simple, yet still allow changing every meaningful value and encourage parameter experimentation by making the user feel reassured he's not going to irreversibly break something by going wild with the values.

This same mentality extends to code interfaces too. When a value within the scope of the tool is meant to be able to change, it's better to make a clear method interface for modifying it instead of just leaving it as a publicly modifiable variable. This way the developer implicitly signals to other programmers that the value can be changed and it's meant to be done through the use of this method. Doing this the developers of the tool can also easily do parameter validation and make sure the changes occur at a valid state within the tool. Achieving this kind of control requires strong encapsulation within the tool, and favoring customizability within the code is in a way just an application of the encapsulated programming architecture.

6 Extending Unity engine with custom and 3rd party tools

Unity game engine is one of the most popular engines for developing games nowadays. This is largely due to it's vast platform support, and it's active and large development community which provides vast learning resources. Unity is also positioned in a sweet spot between high and low level engines with it's C# programming language and it's easily understood gameobject-component programming architecture. While historically only a commercial 3D engine, nowadays it has full support for 2D development and provides all built-in engine features for free.

6.1 Creating custom tools

The entity-component-system (ECS) is a commonly used programming architecture in game development due to it's focus on compositing features from modular components over more traditional inheritance patterns[23]. The ECS is not a well defined term, and it's usage regarding the "systems" part tends to vary between contexts. Unity's programming model resembles ECS in the sense that while entities map directly to gameobjects, components on the other hand usually contain both the data and the logic. This kind of split makes it very easy to implement simple functionality on a given gameobject at any time just by attaching a scripted component to it and modifying it's data from the editor.

Unity abstracts many built-in tools as components. For example a particle system is simply a component attached to a gameobject, and it does not require any external control, renderers or logic to work. Working with these kind of modular tools is an intuitive way to represent functionality attached to an entity. There are many different programming patterns available when programming inside Unity, but most of the time when creating gameplay functionality or tools in Unity the programmers directly write scripts that will be attached to an object to give it the wanted behaviour.

It follows that applying tools focused programming methods to Unity means creating re-usable scripts that allow non-programmers to modify the behavior from the editor inspector. In addition the scripts should provide clear interfaces for other scripts to modify it's data and interact with it as required. This sounds self-explanatory, but as project sizes start to grow, achieving this while managing to avoid a complex dependency jungle requires conscious effort or the help of different programming methodologies.

6.2 Limitations in built-in tools

Built-in tools in Unity are in general fairly well designed and are often easy enough for non-programmers to use. In most cases the tools can be used simply by attaching a script into a gameobject, and then modifying the relevant values in the inspector to gain the desired functionality. These tools are used in myriads of Unity games to implement various common functionalities games tend to require, and often using built-in tools can boost the productivity by a huge margin. However in some cases the various limitations built-in tools can have could be the source of production delays or different game design compromises.

Unity engine has a long history and some of the built-in tools have seen little updates over the years. There are also many tools that have various implementation compromises resulting from Unity engine's focus on cross-platform development, meaning many tools still rely on outdated DirectX 9 rendering techniques and only few are properly multithreaded. While the situation is getting better day by day as Unity updates their built-in tools, the current state is still that many engine tools lag far behind the state of the art.

Take for example the terrain tool of Unity, which is used to generate a height map based terrain and then manually customize that with various textures, foliage and repeating objects like trees. The tool has seen relatively few updates since it's introduction in Unity 2.0 over 10 years ago and is definitely showing it's age in regards to customizability and performance. It has various performance problems due to the terrain splat maps being rendered with multiple passes, and is very limited with customizing foliage visuals, object placement or manually modifying height map features. Due to these problems there are many 3rd party tools that try to fix, append and change the terrain tool functionalities with varying success.

Unity's AI navigation tool is another tool that is commonly seen as limited and problematic in real-world use cases. The tool was introduced in Unity 3.5 over 6 years ago, and contains a separate tool for baking a navigation mesh and then navigating that using navigation agents. While the tool is suitable for simple AI behavior, it's very difficult to use it for more complex AI routing due to various fundamental design issues. The agents are by default meant to use their built-in pseudo physics based system for providing movement to the agents, which greatly limits the available options when trying to implement custom movement or physical interaction behavior. While it's possible to work around this by disabling the agent's movement engine, this is hardly feasible due to the navigation API missing various important functionalities regarding querying the recommended routing path.

The built-in trail effect tool of Unity is another tool which can be difficult to utilize due to the limitations it has. Trails in games are often implemented by generating a procedural mesh and then rendering a stretched or a tiling texture on that, and Unity's trail tool is no different. However due to the way Unity's trail tool generates geometry using only a single point as a reference it's difficult to author the precision of trails during turns, which can generate jagged edges and produce various overlap

and geometry flipping artifacts. Additionally, Unity's trail tool does not support modern texturing techniques like UV mapping using bilinear interpolation. [17]

Finally, taking a look at the character controllers in Unity reveals they also contain multiple limitations and issues that make them difficult to apply to more complex games. Unity's built-in character controller is a traditional non-physical controller that deals with collisions manually. It's fully controlled via scripting and contains simple utilities for determining states such as is the controller grounded. There are multiple issues in it with regards to lack of more granular state information or with the manually defined physical interactions. These kind of issues usually cause the controller to feel stiff or rigid, resulting the games unconsciously feeling less satisfying to the player. There are ways to overcome these limitations, and the easiest methods often utilize the physics engine to deal with more complex interactions.

Because of the limitations often found in built-in engine tools, developers usually have to re-invent or improve the functionalities themselves. Fortunately Unity's rising popularity has brought about a wave of 3rd party tools that tackle the built-in tools issues, but even so finding a 3rd party tool for every issue is often impossible. Next I'm going to document how I've solved some of these problems I've encountered while developing games and commercial software using Unity, and also examine how the built-in tools and the solutions fare when compared against the tools focused development methodology.

6.3 Terrain

Modern terrain generation solutions are all about generating artist controlled content procedurally using various noise functions or machine learning methods. Unity's built-in terrain tool has no support for these, but fortunately allows access to the underlying data structures making extending the system possible. Terrain engines are complex systems that require various high-performance culling and rendering algorithms, so it's better to extend the well working parts of the existing system rather than build your own one from scratch. This is supported even further by the fact that there are many 3rd party tools available that extend the limited parts of the system.

There are 3 main issues with Unity's built-in terrain tool: it's hard to author the terrain, it's slow to render it and it's slow to render foliage on it. I have solved these issues using 3 different commercial 3rd party tools that were developed to solve each issue separately. There are complete terrain engine overhaul 3rd party tools available too, but since some parts of Unity's terrain engine are still competitive when doing traditional height map rendering it's often not required to replace it entirely.

For authoring the terrain geometry I replaced Unity's manual painting workflow with a fully procedural generator that allows a large amount of controllability called Terrain Composer 2[19]. Terrain Composer 2 provides a node based authoring system where you can use different generator functions to modify the height map in

multiple controlled steps. This kind of dynamic generation system is made possible by extensive use of GPGPU compute shaders that apply each node's function to the underlying height map texture in succession. The approach is fast enough to allow seeing the results of the modifications in real-time, which greatly speeds up iteration times even with complex terrains.

Terrain Composer 2 also supports texturing functionality using an intuitive rule based node system seen in figure 3. This interface allows generating a natural shape, slope angle or ground height based texturing on the fly without painstakingly manually painting each feature to the terrain. This kind of combination of procedural nodes and manual authoring of any important landmarks is leaps and bounds ahead of the manual terrain painting systems, which have issues with combining multiple terrains, generating natural-looking shapes and overall getting a pleasing quality to the texture painting.

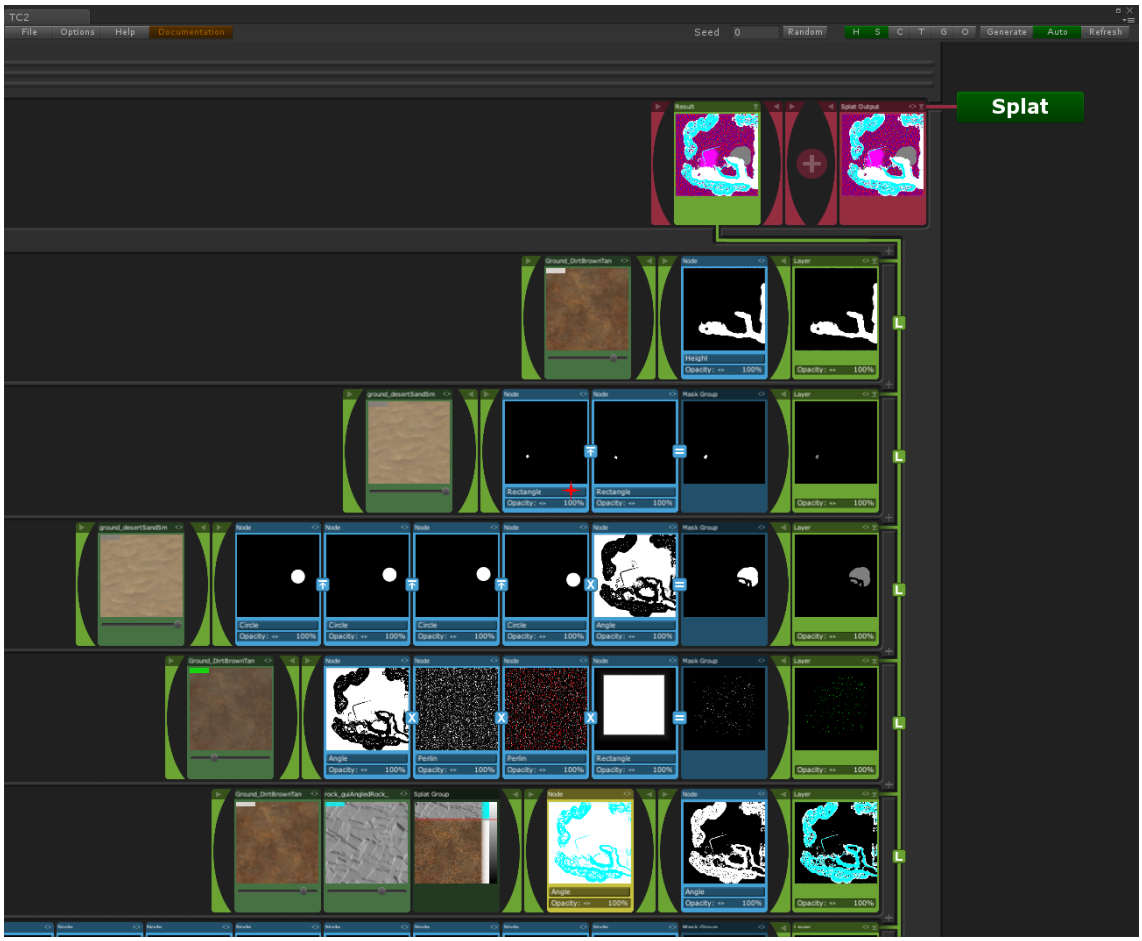

Figure 3: The splat map texturing node editing interface within Terrain Composer 2. The final output of the entire terrain splat map is the blended result of all rightmost vertical node groups. The output of each node is determined by the horizontally arranged nodes.

Like many 3rd party tools, Terrain Composer 2 isn't without it's issues. It's built

on a complex framework of authoring tools that are difficult to fix and update to newer Unity versions in case the developer is late with the updates. Additionally, the framework doesn't contain support for drawing e.g. roads to the terrain with ease, so it's often required to augment the feature set of the tool using custom spline systems that draw roads as a post-process step. However on the whole this kind of added work is a small price to pay to update from the manual terrain authoring workflow to a procedural one.

After getting the geometry and texturing done, the next step was to replace the Unity's standard terrain shader, which is heavily performance-bound by the use of several legacy DirectX rendering techniques. Terrain texturing is often done using splat maps, which are textures that contain the weights for how much of which texture to blend at a given height map position. Unity built-in shader's issue with this is that it requires 1 new shader pass per splat map used, which means that using more than 4 different textures in your terrain becomes extremely expensive performance-wise. Nowadays this kind of rendering is cheaper to achieve using the DX10+ texture array feature, which allows the GPU to have more than one texture active at the same time.

There is a 3rd party terrain rendering tool for Unity called Microsplat that achieves terrain rendering using the modern methods[20]. Microsplat is made to completely replace the Unity's terrain rendering with it's own and it achieves it magnificently performance and feature-wise. It's use of texture arrays allows using up to 16 textures using only 1 shader pass and it's plug-and-play nature setup combined with easy extending via other plugins makes it a very competitive tool even for AAA game development. It also allows modifying the shaders it uses internally to your own rendering style needs, making it complete all guidelines of tools focused development with ease.

However, committing to a rendering technology like Microsplat may have consequences later down the development when Unity updates the engine. As of writing Unity is planning to add support for several different core render loop implementations that are completely incompatible with the current rendering system, making any rendering based assets obsolete until the developers update them. Ultimately all 3rd party tools end up being a gamble against their continued support and the engine features evolving possibly even past them, making it vital to developers to follow the game engine feature road map and make outsourcing judgements based on that.

The final issue to solve with Unity's terrain tool was rendering the terrain foliage. Unity has a built-in support for painting various foliage details and grass on the terrain, but the built-in solution is problematic due to not scaling well to procedural generation systems, not having much room for customization and requiring a lot of manual work to make the terrain lush with details. Unity supports rendering trees made using a popular Speedtree suite of tree generation tools, but these also have multiple performance issues with no support for proper batching or performant billboarding.

Replacing foliage systems with custom tools is extremely time consuming due to their very performance intensive nature. The systems that render the terrain contents must be able to handle culling, serialization and rendering of multiple millions of grass blades and possibly hundreds of thousands of smaller foliage details, all with very strict performance requirements that tend to require implementing complex GPGPU solutions all-around. There is another competitive 3rd party tool called Vegetation Studio[21], that handles all of this and more.

Vegetation Studio is an asset that replaces the tree and foliage rendering systems in Unity's terrain engine completely. It's similar to the Terrain Composer asset in that it handles authoring the placement of foliage using procedural rules that allow creating natural foliage formations based on the existing terrain data. Additionally Vegetation Studio can handle rendering large amounts of foliage and grass thanks to it's wide use of GPGPU culling routines and efficient serialization. Due to these features Vegetation Studio makes it possible to create terrains that can handle extremely high amount of foliage and render it at the state of the art performance.

By selectively replacing parts of the terrain rendering with 3rd party tools and taking only collision handling and geometry optimization from Unity's built-in asset, it's possible to create different kind of terrains that suit many different use-cases and graphical styles. The 3rd party tools used here are well encapsulated regarding their intended purpose and do not interfere with other gameplay scripts or functionalities. The tools also require very little to no setup, their source-code is available and Unity's asset store license permits their modification for personal purposes. Overall this makes augmenting the built-in terrain tool of Unity a no-brainer at least until Unity reveals their own renewed built-in terrain tool, that's as of writing only looming in Unity's development road map with an unspecified date.

6.4 Navigation

Unity contains a tool for implementing 3D AI navigation behavior in arbitrary 3D environments. The workflow of the tool is similar to other 3D navigation solutions, where you first bake a navigation mesh and then use different path query algorithms for traversing the mesh. Unity's solution for baking a navigation mesh is special in that it hard-codes the different agent size radiuses to the navigation mesh, making only certain predetermined size agents work properly for traversing. Additionally Unity's built-in navigation mesh generation tool does not provide any ways for developers to post-process or manually tweak the generated mesh, making it nearly unsuitable for any kind of complex terrain or vaguely defined surfaces and spaces. This issue alone can make it necessary for game developers to look into alternative solutions for providing navigation.

Navigation, like foliage rendering is a heavily performance reliant feature that usually requires a lot of optimization to not slow down the game's frame rate. It's especially demanding due to normally having to support path queries from dozens of navigation

agents simultaneously and also having to provide local evasion procedures for agents. The navigation mesh the agents traverse is a normal triangulated 3D mesh that is overlayed on top of the game world and represents the areas the agents are able to navigate in. In algorithmic sense, the triangles of the mesh are considered to be graph nodes, with each node connecting to 0-3 other neighboring nodes. These nodes are then traversed using various graph search algorithms, with A* search algorithm being the natural industry standard choice due to it's great performance characteristics.

Even with good A* implementations the graph search operation often become excessively expensive for real-time games due to the search space being a very large 2D graph. This often necessitates multi-threading the graph search operations and asynchronously updating the queried path on the fly, because the agents need to start moving towards the currently best known route immediately. In addition to this, it's often required that the agents can avoid other agents so they don't bump and push each other and possibly block movement routes in games. This kind of agent avoidance is called *local avoidance* and is often implemented using variations of flocking algorithms. Unity's built-in tool has no support for local avoidance, and creating a performant custom pathfinding solution with a well working local avoidance makes 3rd party tools look very attractive to developers. There are few game-engine agnostic navigation tools available, but for Unity there's also a Unity specific 3rd party tool called *A* Pathfinding*, that can manage all navigation functionality and is found to work extremely well in many kinds of games.

A* Pathfinding is a 3rd party asset that contains a module for baking a navigation mesh and a rich scripting API for doing queries on the generated mesh. It supports multiple sized agents and works well for arbitrary sized game worlds while also outperforming Unity's built-in tool. Making agents for it is as simple as attaching a single script on the object that requires navigation, and implementing more detailed behavior with additional modifiers, as seen in figure 4. It's navigation mesh authoring tools combined with a logical scripting API for writing agent behavior make it an easy choice for replacing the entire navigation system in Unity.

Usually when writing AI behavior in games, the principal pathfinding question the agents want to know at a given time is *"which way should I go to reach my target?"*. Unity's built-in navigation API does not have a straightforward way of getting this information without constantly doing expensive full-path recalculations in a single game frame. This kind of querying is often unusable for real-time games because the path calculation could take several game frames to complete, making the game stutter each time a path is recalculated. A* Pathfinding provides a clear asynchronous API for doing these queries, which is easy to integrate to developers' custom AI behavior trees.

Overall the issues in Unity's built-in navigation tool and the inability to extend it in any ways make completely replacing it a standard procedure for larger games. A 3rd party tool like A* Pathfinding is both modular and encapsulated due to focusing on it's only job which is navigation. It's nearly fully self-contained code-wise, which

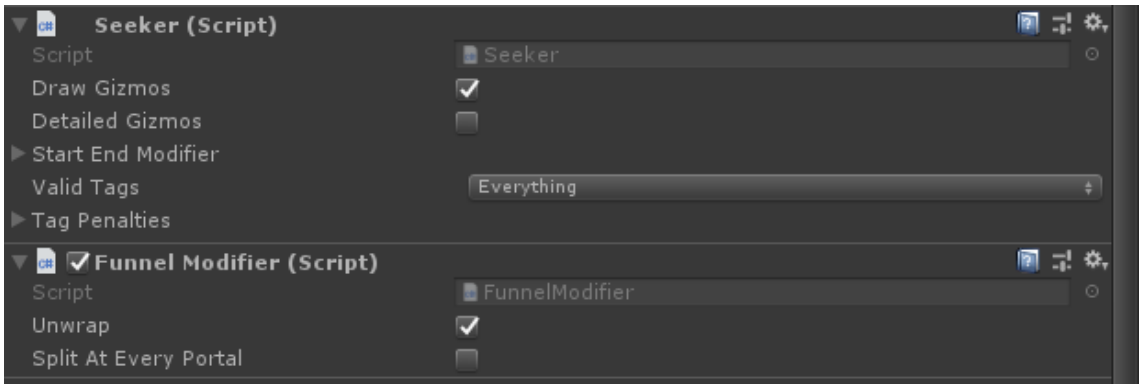

Figure 4: A* Pathfinding agent scripts in the Unity editor inspector. Using the agent navigation API requires simply querying the Seeker script with the wanted routes. The routes can be post-processed for better quality using different modifiers.

makes it resilient against engine updates. Additionally it doesn't pollute the codebase with useless functionalities, requires no setup at all and can be integrated easily to existing AI behavior code, which makes outsourcing game navigation to the tool a good choice when examined through tools focused development guidelines.

6.5 Trail rendering

Game developers often find themselves needing a solution for rendering trails for moving objects in 3D space. There are several good ways to render trailing effects in game engines, and one of the simplest and most popular methods is to simply use a particle emitter with a high enough emission value to make the particle stream look like a solid trail. However, this kind of solution doesn't scale well to longer trails due to severe pixel shader overdraw issues and is often completely unusable on performance bound platforms like mobile phones. Popular engines such as Unity and Unreal both have a built-in support for more elaborate procedural mesh based trails, which are based on a dynamic mesh that is generated as the object moves.

Dynamic mesh solutions are the standard go-to solution for more complex trails over the simple particles, but these methods tend to have other issues with managing the looks of the geometry in a 3D space. The dynamic mesh is normally generated between 2 anchor points that represent the ends of the object that should have trails, and the mesh is updated with new vertices as the anchor points have moved a certain threshold distance in space. This means the generated mesh may look rough when the objects move and turn fast due to insufficient sampling, and additionally there can be overlapping mesh segments at steep turns.

The standard procedure for smoothing fast trails is to use one of the many interpolation methods for making up additional points between the sampled points, which is something Unity's and Unreal's built-in trail renderers do not support by default. Fortunately it is a relatively small task to write a custom trail system with better

interpolation methods such as quadric or cubic Bezier curves, simple Euler interpolation or use one of the many available 3rd party assets or toolsets for all of this. The 3rd party solutions for trails generally perform well for simple tasks, but tend to have little room for customization with regards to interpolation methods and rendering.

Another trail rendering feature as of writing none of the 3rd party trail solutions for either engine provide is the support for *bilinear texture mapping*. Bilinear texture mapping is a relatively little used technique for interpolating mesh texture coordinates using bilinear interpolation on quads instead of the standard way of using barycentric interpolation on triangles. It's not related to *bilinear texture filtering*, which is a method for smoothing the samples of a 2D texture.

Bilinear texture mapping can be desirable in some situations like trail rendering, due to it's resilience against producing warped textures in non-uniform quad shapes. It can be implemented for example by tightly packing the vertex shader input structure on every frame with 4 3D points that represent the quad the vertice is part of. These points are then transformed to screen-space in the vertex shader and passed to the pixel shader, where the shader can bilinearly interpolate the resulting texture UV value using the screen-space position of the fragment and the 4 bounding points. The resulting texture UV value maps smoothly inside the quad, producing a distortion free texture mapping solution that works for all kinds of quad-based meshes as seen in 5.

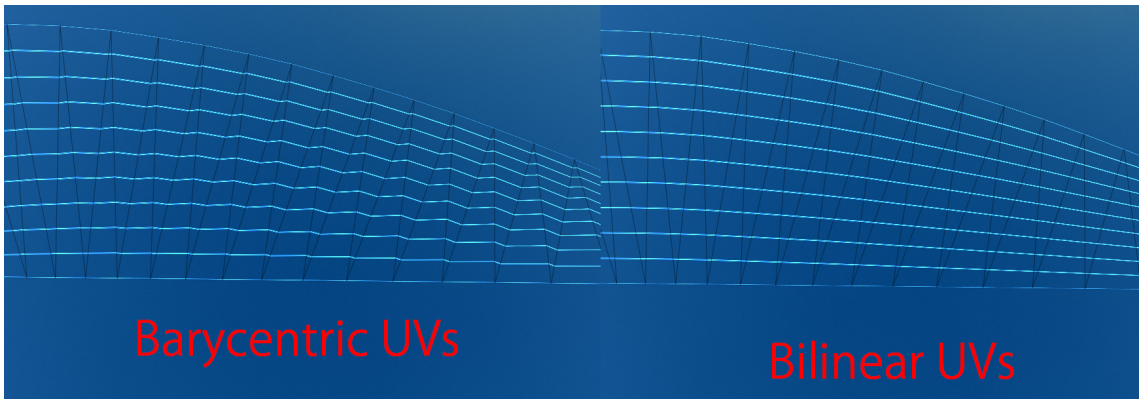

Figure 5: Figure showing the difference between the bilinear and the traditional barycentric UV mapping methods. Black lines show the triangles of the procedural mesh. With barycentric UV's the lines are distorted due to triangles being insufficient to represent the texture coordinates of straight lines.

Trail rendering is a commonly encountered feature in game development, which leaves one wondering why the available built-in or 3rd party solutions don't provide easy customization support for features like alternative interpolation methods or bilinear rendering. The performance characteristics of bilinear rendering are more demanding than those of traditional barycentric interpolation, but due to the small amount of overdraw in rendering procedural mesh trails the differences are often minor in real

use cases. Trail rendering is often kept as it's own well defined modular component, which makes writing a custom tool for trails an attractive option compared to built-in or 3rd party tools.

6.6 Character controller

Character controllers, which are the basic components that move the player are one of the most important parts of games as they tend to define the overall gameplay feel of games in a subtle manner without ever directly presenting their functionality to the players. In this thesis *character controller* is defined as the core component that determines character's movement behavior when it's controlled externally and when it physically interacts with different terrains or other objects. Character controller is the component that defines the movement limitations and state determination of the character and is usually one of the key components of character based games that still doesn't get much explicit attention. Instead character controllers work as a silent elephant in the room that everyone subconsciously acknowledge but rarely point their focus to.

The players use the character controllers all the time, experimenting and testing out their possibilities with different terrains and different gameplay situations. They're what can for example make possible different gameplay elements, make the player think of alternative strategies on the fly, or can provide emergent gameplay from finding unexpected interactions with the environment. Interestingly these same principles also apply to game developers during their development phase too, making them experiment and test out possible new game design ideas that challenge the capabilities of character controllers as the development progresses.

Most character controllers are done by moving a physics engine primitive in the world according to player input. It's possible to augment the interaction by applying physics engine interactions to the primitive, which is nowadays the more common way of improving character controller interaction behavior due to the amount of work needed to get custom physics solutions working well in different interaction situations.

Many available 3rd party tools attempt to create different kind of character controllers, but these solutions are often more targeted to be character frameworks rather than just character controllers. The frameworks provide many different character movement related abilities, such as support for hopping over different types of obstacles, ladder climbing mechanics or other character systems like animator integration, character inventory support and so on.

These 3rd party character frameworks tend to fall short on a fundamental level by not focusing enough on the character's movement and all the unspecified small interactions with the environment. Instead they opt to be only capable of accomplishing the exact features they're hard-coded to do, making writing custom character controllers still a standard practice in game development. I've developed a character controller in the

past that focuses on providing robust and logical movement and interaction behavior with any kind of environment[18], and next I'm going over how a robust character controller tool like this can be useful to game developers during the development phase.

In character based games the movement of the character is generally the prerequisite for all gameplay mechanics. The player progresses the game by moving a character around in levels that are designed to be traversed by the character, and usually the levels are filled with different content relating to the setting of the game. This random content and the general layout of the level may appear in any shape or form in games, making testing the controller behavior in every corner a huge undertaking during development. An example of this is shown in figure 6, which shows a level containing many small rocks player can collide with, an enemy unit and overall uneven terrain.

Figure 6: Modern game terrain is cluttered with small rocks and other obstacles. In character games the way the character movement handles these is of utmost importance. Picture from the game "God of War" (SIE Santa Monica Studio, 2018), which received extremely high critical acclaim upon it's release.

It's common to find small gaps, steppings or other random level content in games where the characters can get stuck or seemingly can't pass a tiny stepping, rock or a gap in the environment. These are the types of geometry issues that players subtly immediately notice, hurting the immersion for the player. The same applies during development when level designers test the layout for the level and may not notice all issues arising from the rough object placement, uneven surfaces, cavities or holes. Having a robust controller can help with these issues without making the developers even think about the problems, saving both nerves and development time with less testing and tweaking.

A physically based character controller has the additional benefit of supporting various physics interactions. If implemented properly, a character controller can support common mechanics like moving platforms, physical impacts and physical interaction with other characters without implementing any specific functionality for these features. The obvious benefit here is the reduced development time, but another less direct benefit also comes from the reduced complexity space within code and interaction behavior. When the programmers don't have to write specific functionality for a feature, the chances of that feature interfering with other game systems such as AI are also reduced. For example, a moving platform or an elevator is difficult to integrate to AI systems if it requires special precautions in the code when used, but if it's just a moving physical object it's easier to write a robust behavior logic for an AI agent.

Additionally a robust physical character controller is less likely to encounter common bugs found when developing environment interactions. For example instead of having undefined behavior when a door closes with the character standing at the door frame, or not supporting physical interaction like buoyant platforms on water, these kind of features just work logically in physical character controllers without the need for any additional code. An ideal character controller is nothing but a single encapsulated component as illustrated by figure 7 that requires no setup from the developer side, and is driven by a simple external script specific to the game.

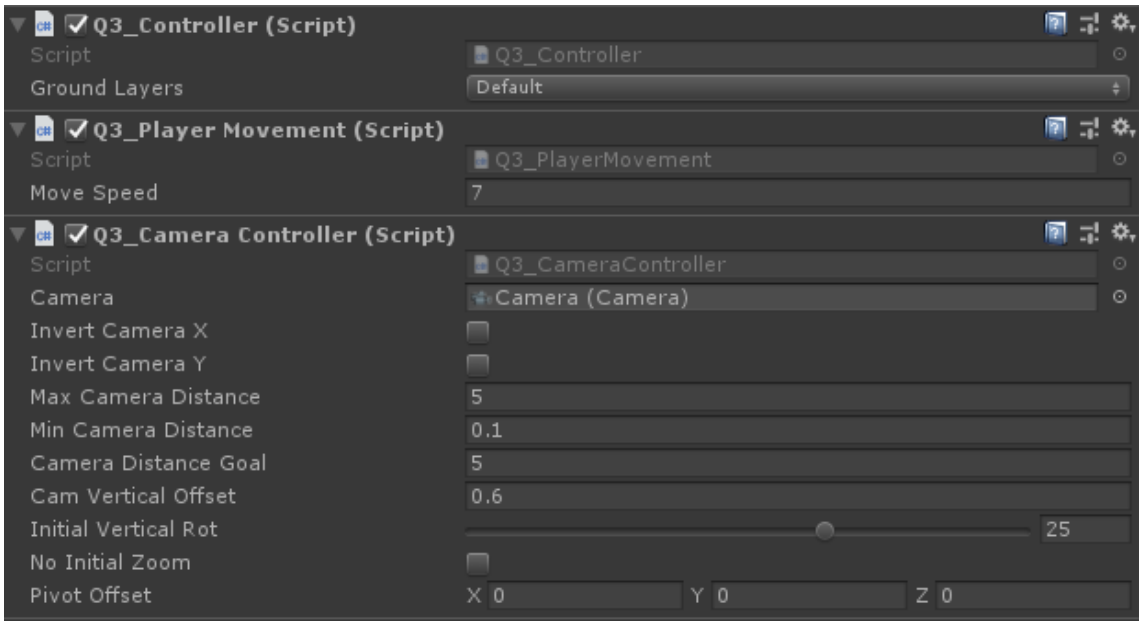

Figure 7: Unity inspector view of an entire character framework in a simple game. The character controller is entirely self-contained in the "Controller" script, which is driven by the "Player Movement" script.

Developing a physical character controller that feels good to use takes more development time than using a 3rd party tool or a naive implementation, but there exists a good amount of literature on the subject with algorithms for selectively

countering the unwanted effects of physics engines. One could expect the 3rd party tools to achieve a good level of robustness in the near future, but as of writing they're generally more focused on providing a wide array of features instead of focusing on generalized movement and physical interaction solutions. The development benefits of robust controller combined with possible emergent game design arising from the behavior that encourages experimentation make developing a custom tool for core character controllers an attractive choice for developers.

7 Tools development in Bucket Detective

Bucket Detective is a commercial 3D first-person indie adventure game me and Jesse Barksdale created mostly during the late 2016. The game was released on the Steam digital games distribution platform on February 16th 2017, and received critical acclaim from both games journalists and customer reviews. Popular websites like RockPaperShotgun, PcGamer, Gamecritics and Vice all covered the game and praised it's dark themes and weird humorous take on serious subjects. On Steam the game has as of writing nearly 90% positive customer review rating, which can be considered very good on the platform considering customer reviews tend to be very critical towards issues in games[22].

The development of Bucket Detective was completed over the course of 17 months with mostly 2 people working on it; me as a generalist and programmer, and Jesse as an artist and designer. Later in the development the project was aided by the inclusion of Mikael Immonen for 3D art assets and Denis Zlobin to design and create the soundscape of the game. The development team did not use any of the common software development methods during the development, which in hindsight might have been detrimental to the planned release date of the game. The game was originally planned to be released in the spring of 2016, but the release was delayed due to not having enough time scheduled for the development amongside other university work. Having used some kind of framework for managing the work could have been useful for scheduling purposes, but the idea wasn't even considered because of the small team size and indie nature of the game.

The indie nature of the game was ever present during the entire development, which used a mostly head-on approach for implementing features. This worked out to be surprisingly effective because the overall design of the game was clearly laid out in a detailed game design document before the development even began in 2015. Some features that weren't accurately depicted in the game design document were heavily prototyped and naturally evolved to generalized tools. Additionally we were able to use some tools I had developed earlier for various other projects in the game, which turned out to be extremely useful and also created new emergent game features during the development.

In hindsight the development could've benefited heavily from more tools focused development style and experimentation, instead of just implementing the abstract

features depicted in the game design document. Even smaller tools with tiny amount of customization managed to turn to completely new features when the designer could freely tweak the values. One example of such feature was the distorted censored text as seen in figure 8, which was actually just normal text using the same wobbly text tool as any other text but with parameters set to extreme values. The entire birth of this new text feature was simply a result of experimenting with a developed tool in unexpected ways.

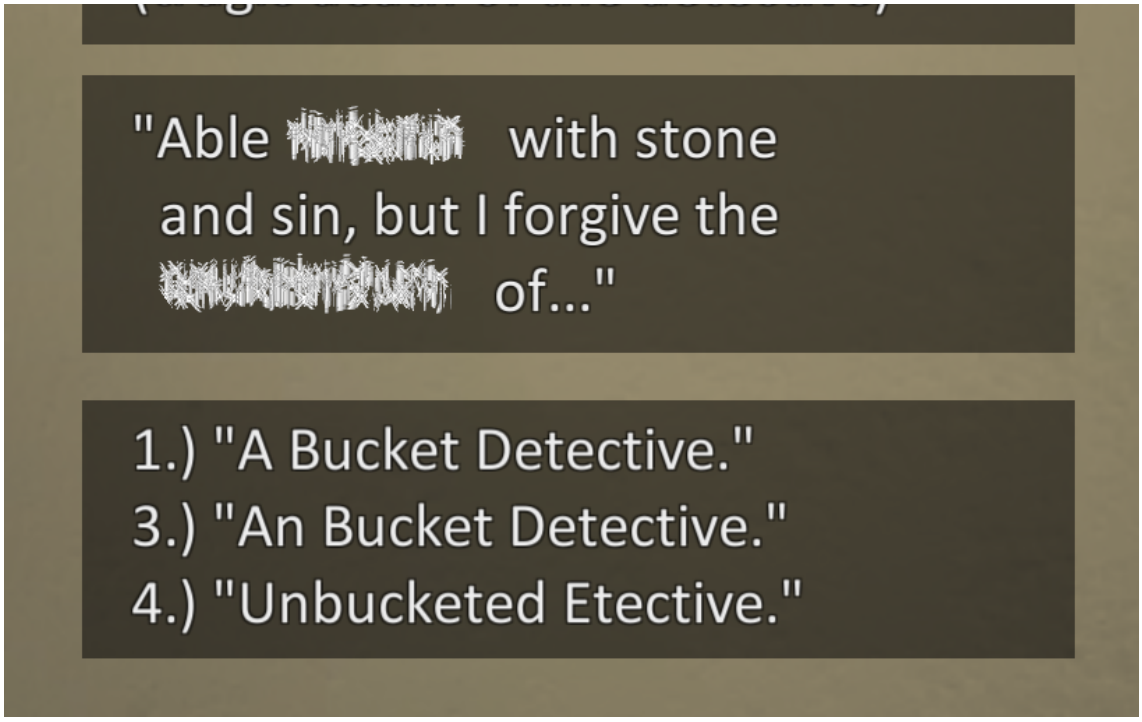

Figure 8: Unique animated distorted text effect born from simply going over the intended values of a normal wobbly text effect.

7.1 Custom character controller

The movement of the main character in Bucket Detective was implemented using a custom physics based character controller I had developed earlier and have written a thesis paper about.[18] The controller turned out to be very effortless to integrate to the workflow, working simply by plugging it in and driving it from a movement script. During development the robustness of the controller features were proved by the fact that the core character controller code stayed the same without any functionality changes during the entire development life cycle.

Even though the levels in Bucket Detective are not very complex geometrically, using a robust controller was able to save development time in places that often require manual tweaking or custom scripts to work as intended. One such place was a typical door, which is one of the features in games that often require more development

time than one might imagine. Creating a door that opens with an animation and does not break immersion in an atmospheric 3D game is not easy due to various interactions it can have with the player. A physical character controller allowed us to create doors that behaved in a logical manner when they hit the player, when the player hit them or when the player was in between them when they were closing as seen in figure 9. Traditionally these issues are solved with fades or making the door non-interactable during animations, but these can have issues with immersion or when the door returns to being interactable with the player clipping with it. Instead of having to deal with these issues that are difficult to find out early and hard to fix later, they were never even present by having a logical physical interaction with the object in the first place.

Figure 9: The doors in Bucket Detective do not close if the player primitive is preventing it. This is a natural consequence of a physical system and required no additional work to implement.

The character controller proved to be extremely useful in implementing the Urn puzzle of Bucket Detective too, in which the player is repelled by a strong force if he tries to bring 2 special urns together due to their magical powers. The repel effect affects the player only when holding an urn, and had to work similarly between the urns outside of the holding state too. Implementing this effect was trivialized by the fact that it was simply possible to add physics engine forces to both objects that were supposed to be repelled. In traditional character controllers implementing this kind of repel effect would often cause various jitter effects and/or various degrees of loss of control from applying different conflicting object offsets, and these issues would have to be fixed by programmers manually in order to produce a satisfying movement

result. These kind of implicit development benefits one does not have to think about are often not emphasized enough in game development, because programmers can spend countless hours tweaking small details like these that in the end define the overall feel of the game.

After the main development was completed, the character controller made it possible to quickly implement an emergent fun easter egg -like feature to the game. One day I had an idea that the game would technically work just fine even if played completely upside down with reversed gravity, so I tried reversing the gravity from downwards to upwards instead. Indeed nothing in the game broke and the game was completely playable, even if the ceiling of the building had bad geometry or difficult obstacles at places. This gave birth to a speed-run mode, where the player could freely manipulate the gravity of the game and attempt to complete it in fastest time possible with a timer visible, as seen in the figure 10. The entire mode wouldn't even had crossed our minds hadn't the controller been robust enough to work in any terrain and in any orientation, and because of it the game had a lot of tangible value added to it using only few hours of development time.

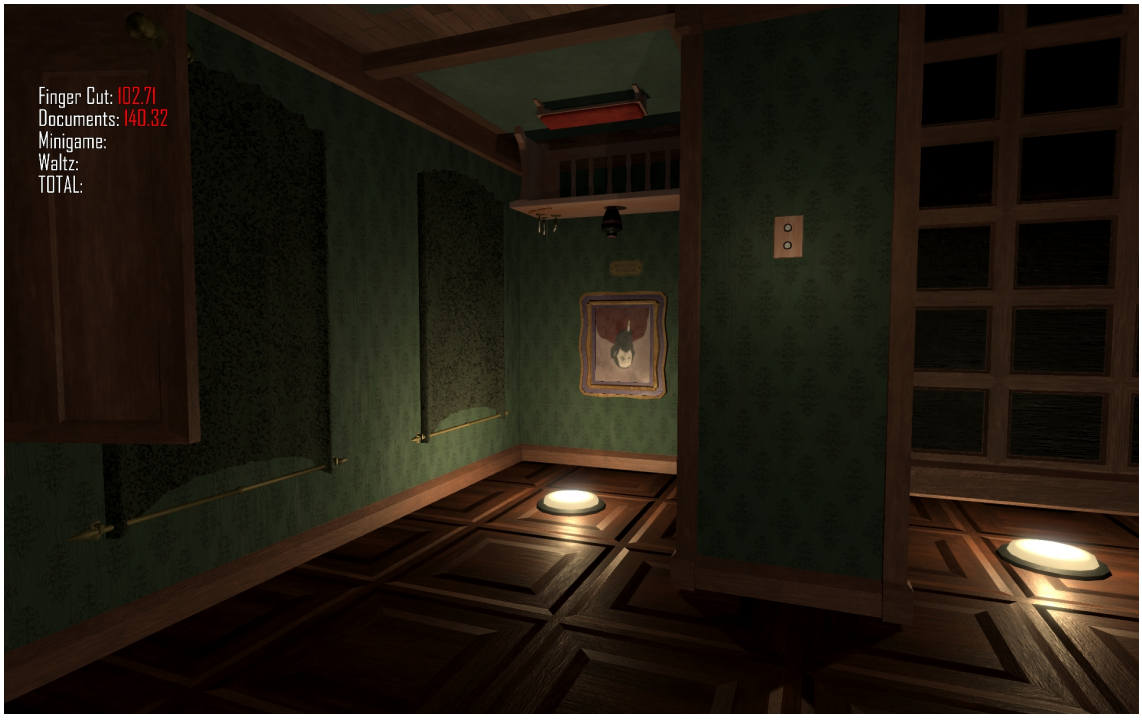

Figure 10: The speed-run/gravity manipulation mode in Bucket Detective made it possible to play the game in surreal ways, for example standing in ceilings.

7.2 Custom physics based interaction system

Another tool that was used extensively during the development of Bucket Detective was a physics based interaction system for picking up and interacting with various items. In Bucket Detective it's possible to pick up one item at a time and carry it in your hands while you see it float in front of the character as seen on the left in figure 11. This was implemented using a fully physics based system that allowed the objects in hand to interact with the environment and also provided smooth physics based animations for picking and dropping objects. The concept for this kind of object carrying is not new, but generally there are no 3rd party tools available that provide features that are highly specific to certain scenarios, which was the case here. Therefore we ended up developing our own system for object interactions that ended up defining a large part of the feel for the game.

Figure 11: The physical interaction system in bucket detective. Left side shows a normal state while the right shows how held items dynamically reacted to the environment.

Our interaction system was special in that it was fully physics controlled even while the objects were held in hand, as seen on the right side of figure 11. Only few games implement item interaction systems with deep integration with physics, which contributed to giving Bucket Detective a unique impression for players. The physics based system also allowed us to make several unique interaction features like a magical paper physically falling off your hand when you leave a room without breaking the player's immersion and making it feel like a gameplay element too much. This worked particularly well because the player could see the paper falling off their hands in front of their eyes, and could try to fight the effect to an extent by trying to pick it up again before it hit the ground.

The interaction system also gave birth to several emergent gameplay features very late in the development. We noticed throwing things around was surprisingly fun because of all the physical interactions they made, so we added book objects to the game that had to be thrown in air and dropped front-side down to be able to read the backside, as seen in the figure 12. The throwing mechanic turned out to work so well that it was utilized with various achievements in the game. There were several silly moments in the game where players would often throw objects down the stairs

and got great fun out of that, so we implemented a secret achievement for doing that enough times. The physical nature of held objects also turned out to be surprisingly fun when people accidentally hit a wall when holding delicate objects and they made a thump sound effect. None of these features were envisioned by any of our earlier designs, and were made possible to integrate easily only by having such a natural interaction system.

Figure 12: A gameplay feature born from physics based interaction system. The book had to be thrown upside down on the table to be able to read the backside.

7.3 3rd party tools in Bucket Detective

The development of Bucket Detective utilized only very few 3rd party tools, with most of them surprisingly resulting a negative experience among the developers. The small usage of 3rd party tools was mostly a result of the mindset that Bucket Detective is a portfolio piece and it should represent the best we could create by ourselves, not just the best what we could assemble. In hindsight the benefits of this development mentality are questionable, as many companies looking to hire people working on games do not usually care about who has made a certain asset, just that it exists in a game made by that person. Even so, the knowledge gained from making custom solutions was definitely not in vain and it's always good to assume that there are no 3rd party tools to make the job easy for you.

One notable 3rd party tool that we attempted to utilize was SabreCSG 3D computational geometry library. SabreCSG is a library for generating 3D meshes in it's

Unity-integrated editor interface using various boolean operations and modification brushes based on Constructive Solid Geometry techniques. The library seemed useful for creating the basic geometry of the levels and we attempted to create several 3D objects and areas using it, but in the end the library turned out to be relatively prone to software bugs and constantly being incompatible with newer Unity versions. We tried to keep contact with the developer of the library and report all the issues we had, but the owner was relatively slow to fix the issues and respond to our queries. Eventually the amount of bugs forced us migrate away from SabreCSG models and use traditional external 3D modeling software instead.

Long time after we had given up on SabreCSG, the developer of SabreCSG announced that he didn't have time to develop the library anymore, and he made the library available for free and open sourced the project. It was a rational choice considering the amount of work that had gone into developing the library and it definitely worked for the best of the community. As of writing the tool is still maintained by active 3rd party developers, but the development seems to be focused on maintenance rather than new features, echoing some of the issues that tend to be present in less popular open-source projects.

Another 3rd party tool we attempted to widely use were the Allegorithmic's Substance materials that were a relatively new feature in Unity at the time of development. Substance materials provided a way to generate different procedural textures using a node-based editor and simple generation rules. In Unity engine these textures were handled as special Substance files, which contained the procedural generation information and allowed Unity to create the actual bitmap textures on the fly.

In practice we found Unity's support for Substance materials to be lacking for real development use. Loading the levels that utilized substance materials was extremely slow unless you had manually selected to "bake" the procedural information to the material texture. It was also common that the procedural materials would simply show as a solid color in the level and had to be manually re-baked by the developers to get the actual texture to show. Even when making builds of the game these issues stayed present and over time forced us to stop using substance files directly in Unity altogether. To migrate away from substance materials we had to go over each substance file and painstakingly replace each one of them with a manually exported bitmap image of the substance material texture, hurting our motivation and using up development time trying to fix the issues.

Eventually Unity removed the built-in support for Substance materials in the Unity editor in the 2017.3 version, making developers that use substance files require an external plugin maintained by Allegorithmic. This could be an indication that Unity wanted to shift the development of substance integration more towards Allegorithmic, making them look responsible for all the issues that might arise instead of Unity.

Final larger 3rd party tool we tried out was ShaderForge, which is a node based shader editor that works directly in Unity. This tool was taken out fairly quickly as I experienced the node structure for shaders became unintuitive and overly complex as

the size of the shaders grew. This is largely in line with the same behavior happening when Unreal engine blueprint visual scripts become large, so this could possibly hint a deeper trait of visual code editors.

8 Conclusion

It is possible to notice several benefits from treating individual game development software components as software tools. When defined as re-usable software components that provide various customizable functionality, it was possible to attribute multiple development benefits to the use of tools in the development process. This made it possible to further define a methodology for a tools emphasized game development workflow that can be used to analyze the benefits and limitations of various existing tools in game engines and 3rd party asset distributions, which can also work as a programmer guideline when creating 1st party tools.

Some benefits of tools were realized during the development process of Bucket Detective. Wherever the mindset of creating tools over specific solutions was applied we noticed clear benefits to the creative process and productivity when examined retrospectively. Additionally the usage of tools allowed a level of customization for the designers that helped the game reach the intended final look faster. The inclusion of more tools focused development didn't come without it's downsides though, and the features that focused on being generalized tools definitely took longer to develop than simpler solutions would've taken.

There are also many possible areas for improvement in the entire development process for the future. One of the core problems we had especially for the latter part of the project was a lack of concise scheduling, which resulted the work being done in erratic bursts at mostly random dates as seen in the figure 13. This was detrimental for the development for 2 main reasons: first the face-to-face development sessions between the 2 developers had to be arranged manually each time, which added burden to even begin doing the development. Second, the lack of scheduling made it easier to leave the project hanging for longer periods of time, as there were no deadlines or obligations to complete the work in a given time frame. Both of these could have been solved by adapting some sort of scheduling system, having external deadlines or trying to adapt a development method like for example scrum sprints.

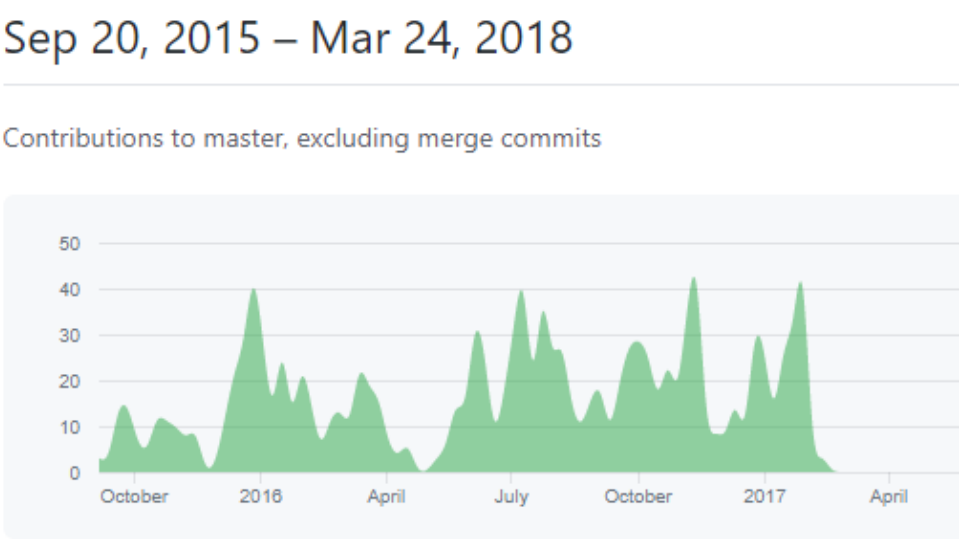

Figure 13: Commit frequency to the Bucket Detective Git repository shows the erratic development times. Some months have less than 5 commits while some go over 50.

Another area which could have been improved was the prototyping phase. Our mentality during the early development was that the game is kind of art and story focused, which locked us to a mindset that didn't encourage prototyping various gameplay mechanics or different presentation styles. Having an actual early prototyping phase could've changed some of the core ideas and mechanics of the game to possibly better directions. Another way to promote this kind of mentality would've been to focus more on tools development. That way the core ideas wouldn't have had to be locked down in place so early in the design phase, and the tools could've provided a way later in the development to experiment with different ideas.

Focusing strongly on tools development early in the development cycle seems like a very attractive idea for future game development projects. Investing time and creating extensive custom tools early on could pave a way for more creative and expressive game design ideas later on, inspiring the developers themselves to play around with the possibilities.

References

[1] Wim van der Vegt, Wim Westera, Enkhbold Nyamsuren, Atanas Georgiev, and Iván Martínez Ortiz, RAGE Architecture for Reusable Serious Gaming Technology Components International Journal of Computer Games Technology, vol. 2016, Article ID 5680526, 10 pages, 2016. http://dx.doi.org/10.1155/2016/5680526

[2] Wilson, J. (April 24, 2014). Even Hearthstone runs on Unity - and that's why it's already on iPad. *Venture Beat*, Retrieved from https://venturebeat.com/2014/04/24/even-hearthstone-runs-on-unity-and-thats-why-its-already-on-ipad/

[3] Church, Doug. "Object Systems: Methods for Attaching Data to Objects and Connecting Behavior". Game Developers Conference Proceedings, 2002.

[4] Jonathan Blow. 2004. Game Development: Harder Than You Think. Queue 1, 10 (February 2004), 28-37. DOI: https://doi.org/10.1145/971564.971590

[5] Frost, P. (August 20, 2003) The Tools Development of Turbine's Asheron's Call 2, Retrieved from https://www.gamasutra.com/view/feature/131224/the_tools_development_of_turbines_.php

[6] Schild, J., Walter, R., & Masuch, M. (2010). ABC-Sprints: adapting Scrum to academic game development courses. FDG. DOI: 10.1145/1822348.1822373

[7] W. Scacchi, "Free and open source development practices in the game community," in IEEE Software, vol. 21, no. 1, pp. 59-66, Jan-Feb 2004. doi: 10.1109/MS.2004.1259221 http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1259221&isnumber=28149

[8] Musil, Juergen & Musil, Angelika & Winkler, Dietmar & Biffl, Stefan. (2010). Improving Video Game Development: Facilitating Heterogeneous Team Collaboration through Flexible Software Processes. Communications in Computer and Information Science. 99. 83-94. 10.1007/978-3-642-15666-3_8.

[9] W. W. Royce. 1987. Managing the development of large software systems: concepts and techniques. In Proceedings of the 9th international conference on Software Engineering (ICSE '87). IEEE Computer Society Press, Los Alamitos, CA, USA, 328-338.

[10] Balaji, S. and Murugaiyan, M. (2012). Waterfall vs V-Model vs Agile: A Comparative Study on SDLC. International Journal of Information Technology and Business Management, [online] 2(1), pp.26-30. Available at: http://jitbm.com/Volume2No1/waterfall.pdf [Accessed 27 Nov. 2016].

[11] Christopher M. Kanode and Hisham M. Haddad. 2009. Software Engineering Challenges in Game Development. In Proceedings of the 2009 Sixth International Conference on Information Technology: New Generations

(ITNG '09). IEEE Computer Society, Washington, DC, USA, 260-265. DOI: https://doi.org/10.1109/ITNG.2009.74

[12] Pamela Bhattacharya and Iulian Neamtiu. 2011. Assessing programming language impact on development and maintenance: a study on c and c++. In Proceedings of the 33rd International Conference on Software Engineering (ICSE '11). ACM, New York, NY, USA, 171-180. DOI: https://doi.org/10.1145/1985793.1985817

[13] A. Stefik and S. Hanenberg, "Methodological Irregularities in Programming-Language Research," in Computer, vol. 50, no. 8, pp. 60-63, 2017. doi: 10.1109/MC.2017.3001257

[14] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in github. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014). ACM, New York, NY, USA, 155-165. DOI: https://doi.org/10.1145/2635868.2635922

[15] Nesky, J. 50 common game camera mistakes – and how to fix them. Retrieved from https://www.gamasutra.com/view/news/259610/Video_50_common_game_camera_mistakes__and_how_to_fix_them.php

[16] Alan Snyder. 1986. Encapsulation and inheritance in object-oriented programming languages. SIGPLAN Not. 21, 11 (June 1986), 38-45. DOI: http://dx.doi.org/10.1145/960112.28702

[17] Inigo Quilez. 2010. "Inverse Bilinear Interpolation". Retrieved from http://iquilezles.org/www/articles/ibilinear/ibilinear.htm

[18] Samu Kovanen. 2014. "Fysiikkasimulaation hyödyntäminen pelihahmon ohjauksessa" http://urn.fi/URN:NBN:fi:aalto-201405302006

[19] Nathaniel Doldersum. "Terrain Composer 2". Unity asset, Retrieved from https://assetstore.unity.com/packages/tools/terrain/terraincomposer-2-65563

[20] Jason Booth. "MicroSplat". Unity asset, Retrieved from https://assetstore.unity.com/packages/tools/terrain/microsplat-96478

[21] Awesome Technologies. "Vegetation Studio". Unity asset, Retrieved from https://assetstore.unity.com/packages/tools/terrain/vegetation-studio-103389

[22] the whale husband. "Bucket Detective". Steam store, Retrieved from https://store.steampowered.com/app/461170/Bucket_Detective/

[23] Tobias Stein. "The Entity-Component-System - An awesome game-design pattern in C++", Retrieved from https://www.gamasutra.com/blogs/TobiasStein/20171122/310172/The_EntityComponentSystem__An_awesome_gamedesign_pattern_in_C_Part_1.php