Breda University of Applied Sciences

Engineering in Game Architecture and Design

Specialisation project

Data-driven game development - evaluating productivity in the current generation of game-engines

Author: Jurre de Baare

Supervisor: Robert Grigg

February 4, 2015
Abstract

This research investigates how data-driven game development middleware impacts productivity by analysing three pieces of middleware, to observe the influence of a data-driven approach on the learning curve and overall productivity. With the results of this study, although the developer is assumed to have C++ and scripting game programming knowledge, we found that the learning curve to be steepest on the Unreal Engine 4 due to its tool focussed approach, extensive user interface and need of context switching between tasks. The total production time on the final run, which assumed complete knowledge of the middleware used, found development times utilizing C++ was 1.2 times faster than Unity3D which was 1.9 times faster than Blueprinting in the Unreal Engine 4. This surprising result, although overlooking the difficulties that future game modifications may incur and how this scales to teams of developers, supports further investigation into how future data-driven engines interfaces in tools such as Unity3D and Unreal Engine 4, may be improved for efficiency.
# Contents

List of figures iv  
List of tables v  

1 Introduction 1  

2 Game Engine Evolution 2  
  2.1 Origin ............................................. 2  
  2.2 Current Generation Game Engines ................. 4  
    2.2.1 Business models ........................................ 4  
    2.2.2 Specialisation ........................................... 5  
  2.3 Components ........................................... 6  
    2.3.1 Graphics ............................................. 7  
    2.3.2 Audio ............................................... 7  
    2.3.3 Input ............................................... 7  
    2.3.4 Core ................................................. 9  
    2.3.5 Physics & collision ............................... 9  
    2.3.6 Animation ............................................ 10  
    2.3.7 Game Systems ....................................... 10  
  2.4 Holy Grail ........................................... 11  

3 Data-driven 13  
  3.1 Definition ........................................... 13  
  3.2 Data ............................................... 14  
    3.2.1 Code ................................................. 14  
  3.3 Data-driven Systems in Game Engines .............. 15  
    3.3.1 Game data ............................................ 15  
  3.4 Advantages of Data-driven Game Systems .......... 17  
  3.5 Challenges of Data-driven Game Systems .......... 21  
  3.6 Digital Content Creation (DCC) ..................... 23
4 Case Studies
  4.1 Experiment .......................................................... 28
     4.1.1 Criteria .......................................................... 29
     4.1.2 Assumptions ...................................................... 30
     4.1.3 Project requirements ............................................ 30
  4.2 Middleware Evaluations ............................................. 31
     4.2.1 Horde3D & Bullet Physics ....................................... 31
     4.2.2 Unity3D .......................................................... 32
     4.2.3 Unreal Engine 4 .................................................. 32

5 Results ................................................................. 34
  5.1 First Run ............................................................ 34
     5.1.1 Challenges ....................................................... 34
  5.2 Final Run ............................................................. 34
  5.3 Learning Curve ...................................................... 35
  5.4 Productivity .......................................................... 36
  5.5 Footage ............................................................... 38
  5.6 Task(s) Breakdown ................................................... 38

6 Discussion .............................................................. 42
  6.1 Learning Curve ....................................................... 42
  6.2 Productivity .......................................................... 43
  6.3 Bias ................................................................. 44

7 Conclusion .............................................................. 45
  7.1 Further Research ..................................................... 46

Appendices ............................................................... 47
  .1 Learning resource list Unreal Engine 4 ............................. 48
  .2 Learning resource list Unity3D ......................................... 49
  .3 Learning resource list Horde3D ....................................... 51
  .4 Learning resource list Bullet Physics ................................. 51

Bibliography ............................................................. 53
# List of Figures

2.1 Games created with the usage of the Doom engine (Doom, Heretic, Hexen, Strife) ........................................ 3
2.2 Gregory’s runtime engine components [43] ................. 8

3.1 Code versus Data: impact on final runtime behaviour .... 15
3.2 Absolute versus Comparative advantage (top to bottom): absolute advantage, comparative advantage, task colours 19
3.3 Age of Mythology [1] map editor ............................... 24
3.4 Shader forge UI [84] .............................................. 25
3.5 Extensive animation state system created in Unity3D’s Mecanim [95] ....................................................... 26
3.6 CryEngine’s flow chart editor, behaviour for a light-switch [22] 26
3.7 LuaEdit [62] ......................................................... 27

5.1 Time difference between runs (time in hours) ............... 35
5.2 Horde3D & Bullet approach time ............................... 36
5.3 Unity3D engine approach time .................................. 37
5.4 Unreal Engine 4 approach time ................................. 37
5.5 Total time share .................................................. 37
5.6 Captured footage compiled into one video .................... 38
# List of Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Learning curve statistics</td>
<td>35</td>
</tr>
<tr>
<td>5.2</td>
<td>Productivity statistics</td>
<td>36</td>
</tr>
<tr>
<td>5.3</td>
<td>Horde3D &amp; Bullet approach task breakdown</td>
<td>39</td>
</tr>
<tr>
<td>5.4</td>
<td>Unity3D approach task breakdown</td>
<td>40</td>
</tr>
<tr>
<td>5.5</td>
<td>Unreal Engine 4 approach task breakdown</td>
<td>41</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This thesis will look at the current generation of game engines in an attempt to address the question:

**How does the usage of a data-driven approach to game development impact the learning curve and productivity?**

The definitions of both what a game-engine and data-driven are in the context of this research will, firstly, be given. Followed by this research’s accompanying experiment for which the same project is developed with the use of three different pieces of middleware, is presented. The results will be assessed and discussed to determine if and how a data-driven approach impacts the productivity and learning curve of a game development team.
Chapter 2

Game Engine Evolution

The purpose of this chapter is to explain why and when the creation of game engines emerged, relating it back to how video games were created at the beginning of the digital era. Followed by taking a deeper look at current generation engines by dividing them into separate components and determining their purpose and functionality.

2.1 Origin

Game engines originate from the need for re-usability [11, 43]. In essence a game engine and the concept behind is: flexibility and re-usability [19]. Game engines form a foundation that can be extended and improved, and which can be used to produce multiple (types of) games [43, 47].

During the beginning of the game development era (year 1972), games were created as singular entities [26].

Because the capabilities of the target hardware were limited the gameplay was programmed with the hardware in mind [12, 26]. This had a big influence in the performance of the game, but also meant that the written code was tied to the game [43].

The capabilities of the target hardware were limited and thus the gameplay had to be programmed with the hardware in mind [12, 26]. This had the advantage of having a better run-time performance, but since the code was written for the specific hardware it was not re-usable. Meaning when a developer finished a title they need to start, almost, from scratch again code-wise.
In those days the scope of a game was small in comparison with current generation projects so starting from scratch was still possible [26], but as the industry grew so did the scope and scale of video games [26, 43]. This resulted in a need for re-usability, the development time grew and starting from scratch on consecutive projects was not feasible any more [36]. Minimizing the development time became important out of economical perspective, and so that the quantity and quality of games could be increased [12,37].

The term game engine was coined during the 1990s with the Doom Engine [29] which was the first major game engine. Created by id Software [54] it was the engine which was used to develop the popular first person shooter Doom. The Doom engine separated systems into components which formed the foundation for the creation of their games [30]. Graphics, audio, input and game mechanics all resided in their own components and the runtime behaviour started to be defined by data [43]. By doing this the engine could be reused for the development of multiple titles. Some examples of published titles that were developed with the use of the Doom Engine are: Doom [28], Doom II [31], Heretic [50], Hexen [51], Strife [91] are some examples. They all feature a different story and artwork, but share the common feel and look as shown in figure 2.1.

![Doom, Heretic, Hexen, Strife](image)

Figure 2.1: Games created with the usage of the Doom engine (Doom, Heretic, Hexen, Strife)
2.2 Current Generation Game Engines

During the past two decades game engines have grown into the foundation for every game made. Gameplay has been decoupled from code and game developers have moved towards a data-driven approach [43]. With a data-driven approach, data determines the runtime behaviour of an application and in the case of a game engine the runtime behaviour is the actual gameplay. Because the game logic is not intertwined with the rest of the game engine the codebase can be reused for consecutive or simultaneous game projects [69]. Reusing the game engine allows programmers to focus on extending functionality and improving quality of the existing code instead of having to start from scratch. The actual gameplay and content of the game is created designers and artists.

2.2.1 Business models

There are several differences between the current generation game engines used for the creation of games [43]. The applied business-model is one, who can develop a game using the game engine. In general, companies create a game engine for the genre of their game(s). They focus on a feature-set that their game requires [12] and create a tool-chain for the development team to use. In general this type of game engine, called an in-house engine [43, p. 27], is used exclusively within their own company. Publishers generally choose to use an in-house game engine across their studio(s), an example of this is the Frostbite engine [39]. The Frostbite engine is developed and used by EA [33] for the simultaneous creation of multiple titles: Battlefield [10], Need for Speed [68] and Plants vs. Zombies: Garden Warfare [78] are some examples.

Another approach for a company is providing a game engine as middleware [67]. They focus on the creation of a flexible and powerful game engine and license it out to other companies. Licensees use the middleware for the creation of their game, which gives them the advantage of not having to create and maintain a game engine themselves. The team of a middleware company, commonly, exists out of programmers only, as they tend to only focus on the technology.

The Unity3D engine [96] is an example of a freemium business-model [76, p 96]. The developer Unity Technologies chose to offer a free license with a limited feature set while also having a premium plan available [98]. By doing
this they attract a number of developers, currently 2.9 million \[97\], who in
general move to the premium plan in the long run. Epic games \[34\] aims to
achieve both since they are creating their own title Unreal tournament \[100\]
while also having made their Unreal Engine 4 \[99\] licensable.

Game engines can also be a result of an open-source project. In such a
project, commonly, a small team works together improving and extending
the feature set of the game engine \[59\]. The community around the project
is continuously testing the engine and reporting the flaws they come across.
Examples of open-source rendering engines are Horde3D \[52\] and Ogre3D
\[72\]. Developers using the open-source game engine can tailor it to their
need because they have access to the source code \[24\]. The project, however,
is the result of the combined experience and skill-sets of the contributors,
quality and consistency can vary \[65\]. And since it is a non-profit venture
there commonly is a lack of or minimal quality control \[89\].

\subsection*{2.2.2 Specialisation}

Games come in different genres, so the engine that is used to develop a game
is often specialized towards the specific game genre. Different game types
mean different requirements for each type \[43\]. For example: First-person
shooters require smooth camera movement, while a MMORPG \(^1\) game re-
quires networking capabilities. A game engine that supports a wide variety
of genres will have a large and complex codebase with a high level of ab-
straction which makes it hard to create and maintain. Due to this problem
in-house game engines are commonly are commonly geared towards the type
of games developed with it. For example Maxis \[64\] created the Glassbox
engine \[41\] which is focussed on the type of games they develop: simulation
games. SimCity \[86\] and the Sims \[92\] are both simulation games that have
been developed with the Glassbox Engine.

Middleware developers on the other hand have to keep a variety of game
genres in mind when developing their middleware. Since they do not have
the privilege to gear their middleware towards a specific genre, they cannot
anticipate the best and worst case runtime scenarios their technology will
encounter. This means that their game engine in comparison with a special-
ized game engine towards a specific game genre, in general, will have a less
optimal runtime performance. This can also be the reason why most of the
triple A game companies choose for an in-house game engine geared towards

\(^1\)Massively multiplayer online role-playing games
their game genre instead of middleware.

Game engines are still merely, however, a tool to develop a game, it does not have defined runtime behaviour, which is the part that varies between different games [60] argues:

In today’s modularly constructed games, the game engine refers to that collation of modules of simulation code that do not directly specify the game’s behaviour (game logic) or game’s environment (level data).

The runtime behaviour of the game developed with a game engine is, almost entirely, defined by data. This makes a game engine a data-driven system meaning that different data will result in different runtime behaviour. [43, p. 11] states:

A data-driven architecture is that what differentiates a game engine from a piece of software that is a game but not an engine. When a game contains hard-coded logic or game rules, or employs special-case code to render specific types of objects, it becomes difficult or impossible to reuse that software to make a different game.

All the data that is required by the game engine such as: 3D models, 2D textures, dialogue files, level layouts, scripts will drive the runtime behaviour of the game engine, which according to Gregory is the definition of a video game, he emphasises: “A video game is the runtime behaviour of a game engine as a result of data driving the engine.”

By altering the data, developers can achieve different runtime behaviour and with a new set of data even can create a different game, this shows the re-usability of the game engine. The changes in gameplay which can easily be achieved by altering the data makes the game engine flexible. Both of these values are the needs from which game engine originate: flexibility and re-usability.

### 2.3 Components

Game engines are composed of individual components which all have an important purpose. An overview of runtime components that can be part of a game engine is shown in fig. 2.2. This partitioning of functionality is a reason why most game engines can be seen as component based applications [38].
Allowing components to be interchangeable serves the purpose of flexibility. Changes in one specific system can be achieved by changing its matching component and without interfering with the rest of the system.

Section 2.3.1 till section 2.3.7 will give additional information for the most common and important components from fig. 2.2.

### 2.3.1 Graphics

The graphics component is commonly the largest and most complex component within the game engine. It fulfills the most important role rendering the final image that is presented to the user, which has to be performed fast and with a high degree of quality. The graphics component is built upon a graphics API \(^2\) such as OpenGL [74], DirectX [27] or a platform specific API. The level of complexity is the result of its broad capabilities since it has to be able to deal with different rendering approaches e.g. 2D sprites, Voxels, Forward and Deferred rendering.

### 2.3.2 Audio

Being responsible for the audible feedback and ambiance of your game means the audio component has to simulate all virtual sounds. The sophistication of audio components has grown over the years. It started with the Unreal Engine which had a very basic audio component, nowadays a wide range of audio middleware that handles the audio part are available such as FMOD [35] are available. Some developers choose to use audio API’s such as OpenAL [73] to build their own audio component.

### 2.3.3 Input

Player interaction with the virtual world is handled by the input component. It processes the input of the available HID’s \(^3\) e.g. keyboard, mouse, controller or touch screen. It has to be able to cope with different kinds of input and multiple inputs at the same time. Input components provide developers with a clean interface to the hardware signals. The input component takes the

---

\(^2\)Application programming interface. [5]

\(^3\)Human interface devices [53]
Figure 2.2: Gregory’s runtime engine components [43]
raw hardware signal and maps, smoothes and clamps it to readable values to be used by the developers [43].

2.3.4 Core

The core component of a game engine is a collection of utilities that are often categorised as core systems. The utilities are used throughout the engine and examples are:

Memory management
System memory is often allocated using a custom written memory manager. This to ensure high performance memory allocation and a deeper understanding of possible memory problems [43, 81].

Mathematics
Games consist of many mathematical systems and developers often choose to write their own vector, matrix, quaternion, interpolation and smoothing systems. Another approach is to integrate middleware like GLM [42] or Eigen [32] who provide these mathematical systems.

Data structures
Data structures [102] such as linked lists, dynamic array, hash maps etc. are often created by developers to fully utilize the target platform’s capabilities and minimize memory allocations [43] instead of relying on middleware such as STL [90] and Boost [13].

2.3.5 Physics & collision

The physics component makes the movement within a virtual world feel real, by running extensive calculations it simulates physical behaviour [9]. Physics are tightly coupled to the collision because without collision detection objects would inter-penetrate and interaction with the game world would be impossible [43]. Middleware is almost always used for the physics and collision components due to the high level of complexity and importance of speed. Havok [48], PhysX [77] and Bullet Physics [17] are examples of available physics and collision middleware.
2.3.6 Animation

The animation component brings the characters and environment of your game alive. Simulating sprite, rigid body, skeletal and vertex animations with smoothing and interpolation is the purpose of the animation component [23, 43]. The animation component is coupled to the physics component because the animations can change the collision shape or animation of ragdoll physics [105].

2.3.7 Game Systems

The game systems component is a broad one which can be split up into sub-components, but for the sake of clarity it is grouped as game systems. Game systems often consist of a scripting system, event system, AI 4 foundation and game object structures and handling. Gameplay systems and rules are created using these game systems as a foundation, on top of which gamespecific code can be written.

Game object(s) handling

A system and structure to define and keep track of the game objects within the game. Typical types of game objects are:

- static objects
- playable objects/characters
- non-playable objects/characters
- interactive objects

This subsystem handles most of the behaviours within the game, it commonly takes care of spawning and destroying of game objects, connecting the game objects to the low-level systems of the game engine and simulating the behaviour of agents, but is not limited to these tasks [43, p. 712].

There are multiple approaches to this subsystem, two examples are object-centric and property-centric. With an object-centric approach each object has a set of attributes and behaviours that are encapsulated within the class of which the object is an instance [43, p. 716]. In a property-centric approach each object is represented by a single unique id and the its properties are distributed across many data sets for which

4Artificial intelligence [14]
the id is a key [43, p. 730]. A property-centric is very similar to a relational database [80, p. 16].

**Scripting system**
A scripting system is commonly used in order to allow developers to create gameplay and content faster and with more ease. Most scripting language do not require an extensive and long compilation process, and changes can be made by modifying and reloading the script file.

**Event system**
An event system is used for the communication between game objects and are commonly found in event-driven architectures. An event system is similar to the observer programming pattern [70] and is used to improve cleanliness of the codebase. It allows communication through events or messages which have a specific type and optionally extra data. They get send to every game object and if one has a handler set for its type it will act accordingly.

**AI foundation**
Because the amount of AI within games grew, foundations for AI became more common as part of a game engine. The emerging AI patterns that can be found in almost every AI system make up the AI foundation of a game engine. Examples are:

- agents
- navigation
- pathfinding

### 2.4 Holy Grail

The concept of a holy grail game engine is currently still a concept and it might stay like that for a while. Game design, hardware, I/O \(^5\) devices etc. are and will be constantly evolving as time passes by, and consequently the specific requirements for a game engine. Tackling this problem is possible, but taking it too far will create a vicious cycle. For example: when all your engine code is abstracted and can be interfaced via a scripting language, systems will be built utilizing and changing every aspect of the game engine. This system will end up being a small game engine itself and defy the original purpose of the actual game engine.

---

\(^5\)Input/Output
The line between what should be available and what is needed is something hard to grasp and will continue to form a complicated decision during the process of creating and maintaining game engines.
Chapter 3

Data-driven

This chapter defines the term *data-driven system*, how its concept ties into game engines and what the advantages and challenges of using them are. It also incorporates the definition of data, types of data commonly found in games and shows how code differs from data.

3.1 Definition

The concept of a data-driven system is all around us, and can be found in and outside of the technological realm it just takes on different forms. The sole definition of data-driven according to [25] is:

- Progress in an activity is compelled by data, rather than by intuition or personal experience.

The definition of a data-driven outside the technological realm is:

**A requires B to produce C**
- When A is absent B cannot produce C.

**B requires A to produce C**
- When B is absent A cannot be combined to produce C.

**C is the result of the combination of A and B**
- When A and B are combined C is the result.

**C\(^1\) is the result of the combination of A\(^1\) and B**
- Given a different variation of A in combination B will result in a different C.
Definitions for this research:

- A is data
- B is an application
- C is the application’s runtime behaviour

When the previous statements are related to the context of this research a game engine, then A is data defining a game, B is the game engine and C is the runtime behaviour/gameplay. The game engine (A) cannot function without appropriate data, and with the suitable data (B) it produces an output behaviour in the form of a game (C). If the data is changed different behaviour will emerge.

Relating the statements to an example outside of our realm e.g. a car engine (B) without fuel (A) will not run, but without an engine the fuel has no purpose. Both parts cannot function without the other.

3.2 Data

Data as a concept is everywhere especially in the computer industry. Data can define anything, actually nothing exists without it. An executable computer program itself is data it contains instructions for your CPU\(^1\) to interpret and execute. Data in the context of data-driven applications is an external source of information which defines the runtime behaviour of the application. The application itself is a small computer, it interprets the data and executes accordingly.

3.2.1 Code

Code is data and is stored in files. It however differs from data used to drive the behaviour of data-driven applications. An example of this difference using the C++ language: C++ code is compiled and linked to create an application, it impacts the behaviour of the application at compile-time. After the compilation it simply has no use. During the run-time of an data-driven application it is data combined with the compiled code that dictates the run-time behaviour. Figure 3.1 demonstrates this process.

\(^1\)Central Processing Unit
Checking the correctness of the code is part of the compilation process. Typographical errors, incorrect naming etc. are cherry-picked from the code and are required to be fixed before the compilation process can be performed successfully. This gives the user a safe-guard since the code has to be correct and suitable for compilation. This, however, does not mean it will work as planned since runtime errors will break the behaviour or crash the program. Data does not have a standardized form of error checking. This means the developer will have to inspect and check every part of the data used for driving the application’s behaviour, because even a small flaw can result in incorrect behaviour and crashes.

Making changes to the final behaviour can be a matter of altering your data, restarting or reloading and seeing your changes. This all without the need of rebuilding the application, and this is what makes data so powerful. The impact on a small codebase is minimal, but if your application takes for example two hours to build having a faster way to change the runtime behaviour will be desirable. Data fulfils this job, as you can alter, swap or create new data to change the application’s runtime behaviour without making a new build.

### 3.3 Data-driven Systems in Game Engines

#### 3.3.1 Game data

Data that drives the aspects of behaviour for a game takes on many forms. A summary of common types is given below:
CHAPTER 3. DATA-DRIVEN

Model data
Data that is exported from a 3rd party modelling package; created by artists. It defines an object by the vertices which make up its shape. Most of the current generation games are 3D environments and require an extensive amount of 3D models to fill the world with.

Texture data
Data that is created by artists with the use of 3rd party tools such as Photoshop. 2D textures can be used for as part of material definitions for models or can be part of the general user interface (GUI).

Shader data
Data that defines the rendering technique for specific (sets of) models or procedures.

Material data
Material definition for specific (sets of) models. It defines which shaders should be utilized to render the model, optional variables for shading and which texture corresponds with the model.

Textual data
Textual interaction and item names are grouped together and stored in files. The files can be edited and reloaded to see the changes. This proves to be very useful for typing mistakes and localization of games: each language has its own dialogue files to provide multiple languages in the game.

Level data
The levels of games have to be created, tested and tweaked according to a certain level design. The placement of objects, spawn points, triggers etc. are stored in level layout files. There is no standardized format for level files, thus most developers create their own which is tied to the game engine.

Particle data
Particles give a sense of movement and add visual effects to a game, and they are made up out of constraints and rules. When combined they form the particle effect, these combinations are stored within data. Size-, velocity-, acceleration-, colour- and rotation- (over-time) are examples of properties particle files can contain.

Configuration data
General variables related to the game are stored in configuration files for example: language, resolution quality settings.
Animation data

There are two types of animations:

**Vertex animations**

Different sets of vertex positions are stored and are animated by blending between the various sets. This type of animation is memory extensive because for each frame all the vertices of the mesh are stored [43].

**Skeletal animations**

Skeletal animations alters the positions of a skeleton. This skeleton is connected to the mesh, and each bone only influences specific vertices. It simulates the human skeleton and skin. The human skeleton is simulated by changing the orientation of the bones and consequently the attached skin [58, 75].

Script data

Scripting files are similar to code, however, script files are interpreted at runtime by a virtual machine. Game engines expose functionality which can be invoked from within the script files. This allows the scripts to utilize functionality that would normally be invoked from within hard-coded behavior. Because of this the application of scripting is broad and it can define procedures and behaviour, examples of scripting languages used in game engines are: LUA [61], Python [79], and AngelScript [4].

3.4 Advantages of Data-driven Game Systems

Using a data-driven approach to game development has its advantages, advantages that are discussed in detail below:

**Absolute vs Comparative advantage**

When the programming team is responsible for both the engine technology and gameplay, they are focussed on what they do best. This means that in theory they will produce the high quality and fast results. A scenario like this is an example of Adam Smith’s absolute advantage theory [83]:

Smith’s thoughts on the division of labour constitute the basis for his theory of international trade. For him, it is the
division of labour that leads to the greatest improvement in the productive powers of labour. As a result of a more advanced division of labour, more output can be produced with the same amount of labour.

The context in which Smith applied his theory is different but the core stays intact: better skilled personnel will produce more output with the same amount of labour.

This means that when a development team focuses on what they do best, they will finish their tasks faster. Game development, however, is an iterative process and the tasks at hand are constantly repeating, changing and increasing which puts a strain on the productivity. Changes in design or art related to the codebase can require making a new build, which increases the pressure on the programming team during the development cycle [101]. This also means the design and art departments are dependent on the programmers, and this can result in losing valuable production time.

David Ricardo’s theory of comparative advantage [71] counters Smith, David argues:

When one nation’s opportunity cost of producing an item is less than another nation’s opportunity cost of producing that item. A good or service with which a nation has the largest absolute advantage (or smallest absolute disadvantage) is the item for which they have a comparative advantage.

His theory suggests that a development team will produce the most, in the least time and for the least production cost if each member performs the task for which he is relatively best suited. A data-driven approach allows for this because tasks are divided more efficient over the different departments and personnel and they perform the tasks in which they have a comparative advantage.
Figure 3.2: Absolute versus Comparative advantage (top to bottom): absolute advantage, comparative advantage, task colours

Figure 3.2 relates the two theories to the implementation of a gameplay element. The top time-line shows an absolute advantage approach, in this scenario the programmer handles the technical tasks and hard-codes the gameplay element. Leaving the designer with: coming up with the gameplay element, waiting for the implementation or performing unrelated tasks, testing and changing the design before sending it back to the programmer. The programmer is able to implement the element relatively fast but has to put his engine technology tasks on hold which means he is losing valuable time for each iteration of the gameplay element.

The bottom time-line shows the implementation of a gameplay element according to the comparative advantage theory, it does assume tools or a scripting language being available for the designer to implement the design. The designer might take longer to complete the initial implementation of the element, but they do not have to wait on the programmer before they can test the gameplay element. Along the way early changes in design could be made when problems arise, and all the while the programmer is able to focus on engine technology.

When this comparative advantage is used throughout the whole development team, tasks can be divided more efficient over the team members. Another advantage is that the designer, after going through
this process N times will specialise and become better, faster and will be able to finish more tasks within the same time-span.

**Iteration speed**

Data-driven systems allow for faster iterations because the runtime behaviour is defined by data [49, p. 47], and there is no requirement of re-compiling when a change has to be made regarding game design [3, 44, 101]. New functionality, extending the system or major changes, however, will require alterations in code and consequently the building process. In general, however, the change in task division increases the iteration speed as was shown in the previous paragraph.

**Empowers team**

Data-driven systems empower the complete development team to create behaviour [3, 44]. With the appropriate tools non-programming personnel will be able to create behaviour that without a data-driven system would require a programmer to hard-code [49, p. 47].

**Porting**

Supporting different hardware or porting a project requires, almost only, the system that is driven by data to be ported. The data is merely (binary) input for the system’s runtime behaviour and this means that in the case of a game engine the runtime behaviour of a game on two different platforms can be driven by the same data. The compiled code running on the platforms will differ and require multiple creations to work, one for each platform. But the data is created once and reused, commonly with small alterations, to drive the runtime behaviour across the different platforms.

**Codebase**

The engine’s codebase will be relatively smaller compared to a one containing hard-coded gameplay [3]. A smaller codebase means less files and interdependencies which could result in faster builds [87].

**Re-usability**

The codebase created for data-driven systems is very suitable for reuse. Not having to scrape all the hard-coded gameplay behaviour out of it but just having the power of throwing the old data out and starting with a clean sheet [103]. This means with a data-driven approach subsequent titles can be created with more ease, and the time for a first prototype will be slimmer as well.
Less interrupting

If tools are available and designers; artists are comfortable working with them team members will interrupt each other less the words: “why does it not work”. Additionally team members will not have to shift their focus and tasks, if the environment they are working is comfortable. Interruptions may seem harmless but Mark according to her research [63] on interruption states:

We found about 82 percent of all interrupted work is resumed on the same day. But here’s the bad news it takes an average of 23 minutes and 15 seconds to get back to the task.

Interruptions however do not only cost time but also impact the team’s well-being, Mark notes:

...we found that people scored significantly higher when interrupted. They had higher levels of stress, frustration, mental effort, feeling of time pressure and mental workload.

Reducing the chance of having to interrupt a team member with an unrelated question and having a clear work-flow will, thus, benefit both productivity and personnel.

3.5 Challenges of Data-driven Game Systems

Using a data-driven approach to game development also poses challenges:

Data interpretation

Allowing data to determine the runtime behaviour of an application it requires to be able to interpret the data it is given with. This means that the application has to be able to understand the data it is given. A layer that parses the data and converts it to behaviour fulfils this role. This layer is added to an already complex game engine architecture and, even when it is well designed, will impact your performance.

Complex codebase

The complexity of the system increases with the amount and variety of data the application needs to be able to interpret. This means the developers have to maintain and create a more complex codebase which requires programmers with knowledge of and experience with the creation and maintenance of data-driven systems.
CHAPTER 3. DATA-DRIVEN

Memory usage
All the data used to drive the behaviour of the application takes up memory, and this can cause problems on platforms where memory is limited. When a project is created for multiple platforms, the size and quality of the data can differ across them to overcome the memory strain. An example of this is 2D textures that are used for the rendering of a 3D model. Due to the amount of available memory being different from platform to platform, the resolution of a texture on for example the PlayStation 4 will be higher in comparison with a mobile platform. The hardware limitations and capabilities have to be taken into account when an application and data is created for different platforms simultaneously.

Dependencies
Dependencies between your data and or code can easily break the behaviour or even crash your application. If the data is spread across files, extra dependencies for the runtime behaviour of your application are added. Without proper management of all the data situations with broken builds and incorrect behaviour (bugs) can occur [44]. If merely one required file is missing errors will occur and when accumulated they will break the system. This means checks have to be in place to validate the data and notify the developer when something is wrong. Implementing these checks takes time and will decrease your initialisation performance, because loading more data and checking if every piece of it is legitimate requires extra processing time.

Tools and learning curve
Each new tool or content pipeline has a learning curve, meaning programmers have to work tightly together with designers and artists when designing and creating the tools to make sure the functionality meets their needs. And even when created with designer and artist insight still will have a learning curve, especially with new employees who have no experience with it at all. Overcoming this learning curve is hard as creating a intuitive tool requires extensive testing and iterations. Creating the tools will also take valuable time which can be limited in the development cycle [3]. The tools have to be kept up to date with new hardware and 3rd party software and has to be kept bug free during the development cycle [40].

Limitations
The extent of what can be defined in data is great but will always stay limited. Technical or user-defined constraints dictate which parts
of the game can and may be defined by data. Problems can arise when a designer or artists tries something that, with the current tool set, is impossible and consequently the programmer will then have to implement the specific part in code [43, p. 793].

3.6 Digital Content Creation (DCC)

This section will take a look at how data used for the runtime of game engines is created and which common tools are required for it.

Levels
Layouts of the levels within your game can be defined in data, and altering the data is commonly performed in a visual way. This means designers/world builders will require a tool in which they can visually place/(re)-move objects from the scene, set-up triggers to start cut-scenes or add a spawn point for enemies etc. Level editors are commonly coupled to the engine’s API, so that the same components used to simulate the final look and feel of the level. The level is exported to a custom format which then can be interpreted and loaded by the game engine. For an example see fig. 3.3 which shows Age of Mythology’s [1] map editor.

3D modelling integration
Third party modelling packages used by artists such as Autodesk’s Maya [7], 3DS Max [6], Mudbox [8] and Pixologic’s ZBrush [104] are able to export the created data to specific formats. In most cases these formats are not directly compatible with the game engine, and converting the data or writing plug ins using third party SDK’s is required to make 3D data compatible with the game engine.
CHAPTER 3. DATA-DRIVEN

Figure 3.3: Age of Mythology [1] map editor

Particles

Particles add a sense of movement, visual effects and feedback to your game and these effects will have to be created and altered. Particle editors are used for this and are commonly coupled to the level editor for placement inside of a level layout. A visual editor gives a preview for the (technical)-artist who is responsible for the particle creation.

Material (shaders)

Shaders used to be written by hand. The task of setting up materials and shaders, however, has shifted towards the hands of specialised artists. They utilize tools to set up shader networks and create realistic materials. An example of such an editor is Shader Forge [84], created as an Unity3D extension, it enables artists to link and set-up material/shader nodes with direct visual feedback. See fig. 3.4 for Shader forge’s UI.
CHAPTER 3. DATA-DRIVEN

Animation
Animators will have to be able to quickly see their animation running in the game engine since the outcome will not always be the same as within their 3rd party animation package. Setting up blend trees and animation states by combining animations can also be required for the game. An example of an animation composer is Unity3D’s mecanim [95], see fig. 3.5 which shows the editor’s UI and an extensive animation state system.

Figure 3.4: Shader forge UI [84]

Scripting
Scripting can determine a wide variety of behaviour. Creating and editing scripts can be approached in two ways: visual scripting or textual scripting.

Visual
Visually creating scripts is usually achieved with flow graph editors. Nodes define specific aspects of behaviour and conditions to start executing e.g. setting the position of an actor or start executing when the actor collides with the floor. Implementing and supporting a wide range of nodes takes a lot off time but will enable designers and artists to create complex behaviour. An example is CryEngine’s flow graph editor which can be seen in fig. 3.6.
Textual

Textual creating script can be performed in any text editor, but commonly extensions are written or utilized to enhance the support e.g. auto-completion, early-on warnings, syntax highlighting etc. An example is LuaEdit [62] which can be used to create lua scripts, it supports debugging the created scripts, has auto completion and more. See fig. 3.7 for LuaEdit’s UI.

Figure 3.5: Extensive animation state system created in Unity3D’s Mecanim [95]

Figure 3.6: CryEngine’s flow chart editor, behaviour for a light-switch [22]
Figure 3.7: LuaEdit [62]
Chapter 4

Case Studies

This chapter will introduce the design of the experiment and the game engines which are being evaluated.

4.1 Experiment

The case study for this thesis consists of recreating one, relatively small, project with the use of three different pieces of middleware and approaches. Each of them has a different level of data-driven systems. The three chosen pieces of middleware are:

- Horde3D (version 1.0.0) [52]; Bullet Physics (version 3.x) [17]
- Unity3D (version 4.5.4f1) [96]
- Unreal Engine 4 (version 4.4.3) [99]

As mentioned each middleware will be used to approach the project differently, specifically the way in which behaviour will be created.

**Horde3D; Bullet Physics**

This approach is focussed on programmers. The two pieces of middleware will be incorporated in a C++ project, created with Visual Studio 2010, that contains the hard-coded behaviour.

**Unity3D**

Unity3D is between the other two approaches, focussed on both artists; designers and programmers. Behaviour will be created using the tools
to the maximum extent they allow for, and the remaining behaviour will be implemented with C# programming.

**Unreal Engine 4**

The Unreal Engine 4 approach is focussed on designer/artist roles and is the most data-driven and UI intensive approach. Their provided visual scripting language Blueprints and their accompanying tools will be used to create all the behaviour.

The creation of these projects will be performed twice, the first run (section 5.1) serves as an initial estimate of time spent on the project without prior experience with the project or middleware. It also addresses the problem of having no prior experience with specific pieces of middleware, by having to look up learning resources for knowledge about their usage.

The gained experience and insight on which improvements will result in the best and fastest result are iterated up until a final run (section 5.2). The final run is the development of the game, using the respective middleware, where all the knowledge, skills and steps for creating the game are known to see how much time is spend solely on the steps. It will be recorded and analysed to reveal differences across the three approaches (see section 4.1.1).

### 4.1.1 Criteria

The case study will be assessed according to criteria which are related to the learning curve and productivity of each approach.

**Learning curve**

- time taken using learning resources (first run)
- amount of learning resources required
- ratio of time taken spent on learning resource in first run versus the final time improvement

**Productivity**

- difference in production time between the first and final run
- total time spent
- time spent in categories
  - UI interaction
4.1.2 Assumptions

For the research it is assumed that:

- The user is at an advanced experience level with C++ and C#.
- The user has developed multiple games and implemented gameplay repeatedly.
- The user has advanced experience with Horde3D.
- The user has no experience with Bullet Physics.
- The user has intermediate experience with Unity3D.
- The user has no experience with the Unreal Engine 4.

4.1.3 Project requirements

The test-case project performed with every individual piece of middleware is a 3D time-trail racing game similar to Micro machines [66] or Track mania [94]. Each feature covers a particular aspect of the engine(s) Unreal Engine 4, Unity3D and the C++ version (Horde3D; Bullet Physics) used for the comparison. The required gameplay elements are as followed:

- vehicle movement
  - steering
  - torque
  - gear box
  - handbrake
- environment
  - track
  - checkpoint system
• visuals
  – first person camera
  – third person camera
  – top down camera
  – countdown sequence

• UI
  – RPM indicator
  – gear indicator
  – speed indicator
  – checkpoint indicator

4.2 Middleware Evaluations

4.2.1 Horde3D & Bullet Physics

Usage

As mentioned in section 4.1 both the libraries Horde3D and Bullet Physics will be used in combination with C++ programming. The project will be created from the ground up after which the requirements will be implemented. During this process some boilerplate code for interfacing with both of the libraries.

Middleware Summary

*Horde3D*

Horde3D is a small open source 3D rendering engine. It aims to offer decent graphics performance while having a small code and data footprint. Their motto is: “Simplicity is the highest form of sophistication”.

It is written in an effort to create a graphics engine that offers the stunning visual effects expected in next-generation games while at the same time being as lightweight and conceptually clean as possible.
 Horde3D has a simple and intuitive interface accessible from virtually any programming language and is particularly suitable for rendering large crowds of animated characters in next-generation quality.

**Bullet Physics**
Bullet Physics is an open source physics library under the Zlib license [93]. Bullet is used in cinematography [85], 3D authoring tools and is commonly found integrated into game-engines [18].

### 4.2.2 Unity3D

**Usage**
The project will be recreate with a combination of C# scripting and the available tools from Unity3D. This approach is a meet in the middle solution for creating the project, partially C# and partially data-driven.

**Middleware Summary**
Unity3D is developed by a company that goes under the same name: Unity [96]. It started off as an OS X platform game engine but grew into a widely used multi-platform engine. Unity3D can be used by anyone to develop a game with. Using it with a free license has its limitations and the Unity Pro license comes at a price of $1,500 USD or $75/month for twelve months of Pro usage. Access to different platforms is made easy in Unity3D, however Pro or special license is required for certain platforms (iOS, Android and consoles).

### 4.2.3 Unreal Engine 4

**Usage**
The project will be recreated in a data-driven way using the Unreal Engine 4’s Blueprints [55], and in this fashion all the game behaviour/gameplay will be created using UI interactions and visual scripting. This approach tries to create a flexible system that will allow for quick changes in gameplay.
Middleware Summary

The Unreal Engine 4 is developed by Epic Games [34]. As the name implies it is the fourth game engine Epic Games developed. Developers can utilize the Unreal Engine 4 for a monthly subscription of $19 and five percent of the revenue. With this you will get access to the entire engine: the tool suite, all their features and the whole codebase. The subscription allows you to develop games for the following platforms: Windows, Mac, iOS and Android. Consoles as with Unity3D are limited to certified developers only.
Chapter 5

Results

5.1 First Run

The first run had the purpose of showing an initial estimate of the time spent for each approach. It had the considerations of prior levels of experience regarding programming, scripting and the use of the middleware. This also meant going through the learning curve for Bullet Physics, the Unreal Engine 4 and Unity3D.

5.1.1 Challenges

During the process of completing the first runs one main problem surfaced, because of having no prior knowledge of the middleware learning resources were required to complete the first run. A list of the learning resources for each approach can be found in Appendices section .1, section .2, section .3 and section .4. The Unreal Engine 4 and Unity3D approach suffered the most time loss, which had an impact on their first run times.

5.2 Final Run

The final run was performed to be able to compare each individual approach with a zero knowledge deficiency. Every aspect of the project had already been created once and a list of steps required to complete the project from
start to end was formed. By doing this the inexperience with the Unreal Engine 4, Unity3D and C++ with Bullet Physics was negated.

5.3 Learning Curve

<table>
<thead>
<tr>
<th></th>
<th>C++ (Horde3D; Bullet Physics)</th>
<th>Unity3D</th>
<th>Unreal engine 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time first run (hh:mm)</td>
<td>18:00</td>
<td>26:00</td>
<td>30:00</td>
</tr>
<tr>
<td>Number of learning resources</td>
<td>16</td>
<td>33</td>
<td>27</td>
</tr>
<tr>
<td>Total time second run (hh:mm)</td>
<td>01:22</td>
<td>01:34</td>
<td>02:58</td>
</tr>
<tr>
<td>Percentage of initial time</td>
<td>7.60%</td>
<td>6.00%</td>
<td>9.86%</td>
</tr>
<tr>
<td>Total time first run - total time final run (hh:mm)</td>
<td>16:38</td>
<td>24:26</td>
<td>27:00</td>
</tr>
<tr>
<td>Time per resource (minutes)</td>
<td>58.71</td>
<td>43.70</td>
<td>60.00</td>
</tr>
<tr>
<td>Percentage time improvement per resource</td>
<td>5.56%</td>
<td>2.85%</td>
<td>3.34%</td>
</tr>
</tbody>
</table>

Table 5.1: Learning curve statistics

Figure 5.1: Time difference between runs (time in hours)
5.4 Productivity

<table>
<thead>
<tr>
<th></th>
<th>C++ (Horde3D; Bullet Physics)</th>
<th>Unity3D</th>
<th>Unreal engine 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time final run (hh:mm)</td>
<td>01:22</td>
<td>01:34</td>
<td>02:58</td>
</tr>
<tr>
<td>Percentage of initial time</td>
<td>7.60%</td>
<td>6.00%</td>
<td>9.86%</td>
</tr>
<tr>
<td>Percentage UI interaction tasks</td>
<td>0.00%</td>
<td>42.70%</td>
<td>55.20%</td>
</tr>
<tr>
<td>Percentage testing tasks</td>
<td>16.70%</td>
<td>15.30%</td>
<td>11.50%</td>
</tr>
<tr>
<td>Percentage coding/visual scripting tasks</td>
<td>83.30%</td>
<td>42.00%</td>
<td>33.40%</td>
</tr>
<tr>
<td>Percentage of initial time</td>
<td>7.60%</td>
<td>6.00%</td>
<td>9.86%</td>
</tr>
<tr>
<td>Percentage of time spend on non-technical tasks</td>
<td>16.70%</td>
<td>58.00%</td>
<td>66.70%</td>
</tr>
</tbody>
</table>

Table 5.2: Productivity statistics

Figure 5.2: Horde3D & Bullet approach time
CHAPTER 5. RESULTS

FIGURE 5.3: Unity3D engine approach time

42.7% UI interaction
15.3% Testing
42% Coding

FIGURE 5.4: Unreal Engine 4 approach time

55.2% UI interaction
11.5% Testing
33.4% Coding

FIGURE 5.5: Total time share

Horde3D & Bullet approach
Unity3D engine approach
Unreal Engine 4 approach
5.5 Footage

The captured footage, see fig. 5.6, is an addition to this report and demonstrates performing the three final runs visually. It shows an overview of the task performed and time spent for each approach, and also visualizes the changes in task distribution (UI interaction, testing, coding/visual scripting).

Figure 5.6: Captured footage compiled into one video

5.6 Task(s) Breakdown

The recorded footage of the second run for each approach has been broken down into the individual performed tasks. The amount and type of tasks vary per approach, and are shown in table 5.3, table 5.4 and table 5.5.
## Table 5.3: Horde3D & Bullet approach task breakdown

<table>
<thead>
<tr>
<th>Task name</th>
<th>Time</th>
<th>% coding</th>
<th>% testing</th>
<th>% UI interaction</th>
<th>% of total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting up track</td>
<td>2:24</td>
<td>90</td>
<td>10</td>
<td>0</td>
<td>3.5</td>
</tr>
<tr>
<td>Setting up cart</td>
<td>1:34</td>
<td>80</td>
<td>20</td>
<td>0</td>
<td>1.5</td>
</tr>
<tr>
<td>Setting up goals</td>
<td>0:24</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0.6</td>
</tr>
<tr>
<td>Adding wheels</td>
<td>1:28</td>
<td>90</td>
<td>10</td>
<td>0</td>
<td>2.1</td>
</tr>
<tr>
<td>Adding rims</td>
<td>3:52</td>
<td>90</td>
<td>10</td>
<td>0</td>
<td>5.6</td>
</tr>
<tr>
<td>Car input handling</td>
<td>3:28</td>
<td>70</td>
<td>30</td>
<td>0</td>
<td>3.7</td>
</tr>
<tr>
<td>Checkpoint handling</td>
<td>1:24</td>
<td>60</td>
<td>40</td>
<td>0</td>
<td>1.7</td>
</tr>
<tr>
<td>Checkpoint rendering functionality</td>
<td>3:20</td>
<td>50</td>
<td>50</td>
<td>0</td>
<td>4.8</td>
</tr>
<tr>
<td>Checkpoint and lap count functionality</td>
<td>1:20</td>
<td>30</td>
<td>70</td>
<td>0</td>
<td>3.1</td>
</tr>
<tr>
<td>Countdown functionality</td>
<td>4:34</td>
<td>70</td>
<td>30</td>
<td>0</td>
<td>5.4</td>
</tr>
<tr>
<td>Countdown timer functionality</td>
<td>3:24</td>
<td>70</td>
<td>30</td>
<td>0</td>
<td>3.1</td>
</tr>
<tr>
<td>Countdown timer behaviour</td>
<td>4:38</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>1.7</td>
</tr>
<tr>
<td>Lap HUD text</td>
<td>3:20</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>1.2</td>
</tr>
<tr>
<td>Time to string functionality</td>
<td>1:12</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>1.2</td>
</tr>
<tr>
<td>Checkpoints HUD text</td>
<td>3:20</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>1.2</td>
</tr>
<tr>
<td>Laptimes HUD text</td>
<td>0:48</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0.8</td>
</tr>
<tr>
<td>Checkpoints HUD text and camera switching</td>
<td>3:56</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>5.2</td>
</tr>
<tr>
<td>Camera switching functionality</td>
<td>4:40</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>6.7</td>
</tr>
<tr>
<td>Camera switching key input</td>
<td>4:40</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>6.7</td>
</tr>
<tr>
<td>Camera switching key input</td>
<td>2:24</td>
<td>80</td>
<td>20</td>
<td>0</td>
<td>3.7</td>
</tr>
<tr>
<td>Camera switching key input</td>
<td>4:24</td>
<td>50</td>
<td>50</td>
<td>0</td>
<td>6.4</td>
</tr>
</tbody>
</table>

---

**Chapter 5. Results**

39
<table>
<thead>
<tr>
<th>Task name</th>
<th>Time</th>
<th>% coding</th>
<th>% testing</th>
<th>% UI interaction</th>
<th>% of total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting up track and cart</td>
<td>1:04</td>
<td>1,1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Setting up track materials</td>
<td>2:40</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Setting up cart body/behaviour</td>
<td>4:32</td>
<td>0</td>
<td>30</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Adding cart script/behaviour</td>
<td>4:32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Setting up cart properties</td>
<td>0:40</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Adding wheel script/behaviour</td>
<td>4:48</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Adding first person camera</td>
<td>4:52</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Camera switching behaviour</td>
<td>0:40</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Adding checkpoint positions</td>
<td>2:32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Camera switching checkpoint behaviour</td>
<td>2:32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Adding checkpoint script</td>
<td>2:32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Setting checkpoint system script/behaviour</td>
<td>2:32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Lap (time) behaviour</td>
<td>2:32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Red/green light behaviour</td>
<td>2:32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Setting up HUD data</td>
<td>2:32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Time to setting functionality</td>
<td>2:32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Setting lap times HUD data</td>
<td>2:32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Checkpoint HUD text</td>
<td>2:32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Checking lap times HUD text</td>
<td>2:32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Setting up damped camera</td>
<td>2:32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Gear behaviour</td>
<td>2:32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 5.4: Unity3D approach task breakdown
<table>
<thead>
<tr>
<th>Task name</th>
<th>Time</th>
<th>% coding</th>
<th>% testing</th>
<th>% UI interaction</th>
<th>% of total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Importing car</td>
<td>1:12</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0.7</td>
</tr>
<tr>
<td>Setting up car materials</td>
<td>9:48</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>5.9</td>
</tr>
<tr>
<td>Importing track</td>
<td>2:00</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>1.2</td>
</tr>
<tr>
<td>Setting up track materials</td>
<td>8:00</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>4.8</td>
</tr>
<tr>
<td>Adding track blueprint</td>
<td>2:00</td>
<td>0</td>
<td>30</td>
<td>70</td>
<td>1.2</td>
</tr>
<tr>
<td>Setting up cart body collider</td>
<td>1:12</td>
<td>0</td>
<td>30</td>
<td>70</td>
<td>0.7</td>
</tr>
<tr>
<td>Adding cart blueprint</td>
<td>0:48</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0.5</td>
</tr>
<tr>
<td>Adding camera</td>
<td>4:00</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>2.4</td>
</tr>
<tr>
<td>Adding game mode blueprint</td>
<td>1:12</td>
<td>0</td>
<td>20</td>
<td>80</td>
<td>0.7</td>
</tr>
<tr>
<td>Setting up wheel colliders</td>
<td>1:24</td>
<td>0</td>
<td>30</td>
<td>70</td>
<td>0.8</td>
</tr>
<tr>
<td>Adding front/back-wheel blueprints</td>
<td>2:00</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>1.2</td>
</tr>
<tr>
<td>Adding animation blueprint</td>
<td>1:36</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>1.0</td>
</tr>
<tr>
<td>Setting up input mappings</td>
<td>2:00</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>1.2</td>
</tr>
<tr>
<td>Setting up car input</td>
<td>4:00</td>
<td>70</td>
<td>20</td>
<td>10</td>
<td>2.4</td>
</tr>
<tr>
<td>Camera switching functionality</td>
<td>8:00</td>
<td>70</td>
<td>30</td>
<td>0</td>
<td>4.8</td>
</tr>
<tr>
<td>Positioning cart on track</td>
<td>1:36</td>
<td>0</td>
<td>10</td>
<td>90</td>
<td>1.0</td>
</tr>
<tr>
<td>Setting up gamemode blueprint</td>
<td>1:36</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>1.0</td>
</tr>
<tr>
<td>Countdown behaviour</td>
<td>9:12</td>
<td>70</td>
<td>20</td>
<td>10</td>
<td>5.6</td>
</tr>
<tr>
<td>Adding and setting up cart HUD blueprint</td>
<td>1:24</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0.8</td>
</tr>
<tr>
<td>Screen ratio behaviour</td>
<td>1:36</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>1.0</td>
</tr>
<tr>
<td>Countdown HUD text</td>
<td>10:00</td>
<td>50</td>
<td>30</td>
<td>20</td>
<td>6.0</td>
</tr>
<tr>
<td>Green/red lights behaviour</td>
<td>9:36</td>
<td>50</td>
<td>30</td>
<td>20</td>
<td>5.8</td>
</tr>
<tr>
<td>RPM bars HUD behaviour</td>
<td>15:24</td>
<td>70</td>
<td>20</td>
<td>10</td>
<td>9.3</td>
</tr>
<tr>
<td>Adding checkpoint positions</td>
<td>10:00</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>6.0</td>
</tr>
<tr>
<td>Checkpoint behaviour</td>
<td>12:00</td>
<td>75</td>
<td>5</td>
<td>20</td>
<td>7.2</td>
</tr>
<tr>
<td>Lap behaviour</td>
<td>6:36</td>
<td>90</td>
<td>0</td>
<td>10</td>
<td>4.0</td>
</tr>
<tr>
<td>Speed and gear HUD text</td>
<td>9:24</td>
<td>80</td>
<td>10</td>
<td>10</td>
<td>5.7</td>
</tr>
<tr>
<td>Lap HUD text</td>
<td>5:00</td>
<td>80</td>
<td>20</td>
<td>0</td>
<td>3.0</td>
</tr>
<tr>
<td>Time to string functionality</td>
<td>9:00</td>
<td>90</td>
<td>0</td>
<td>10</td>
<td>5.4</td>
</tr>
<tr>
<td>Lap times HUD text</td>
<td>9:00</td>
<td>60</td>
<td>30</td>
<td>10</td>
<td>5.4</td>
</tr>
<tr>
<td>Checkpoint HUD text</td>
<td>5:00</td>
<td>80</td>
<td>20</td>
<td>0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

Table 5.5: Unreal Engine 4 approach task breakdown
Chapter 6

Discussion

Performing the case-study supplied insightful results on the learning curve and productivity criteria which will now be discussed.

6.1 Learning Curve

In terms of the learning curve the amount of resources and difference in time spent on each run are the most important statistics to look at. In the first run the Unreal Engine 4 clearly required the most time, however this is in line with the assumption of having no prior knowledge of the Unreal Engine 4. The Unity3D approach comes in second place behind the C++ approach which only took 25% of the total time spent across the three approaches.

The amount of learning resources is, arguably, related to the learning curve of each approach. Even though there was no prior knowledge of the Unreal Engine 4 as a surprise Unity3D, on which there was an intermediate level of knowledge, required the most learning resources. The actual type of resources, however, differ the Unreal Engine 4 approach had more visual (video) resources; the Unity3D approach required relatively more textual resources. This is affirmed by the amount of time spent per resource, the average of the Unreal Engine 4 approach is 60 minutes while the Unity3D approach required 44 minutes per resource.

The Unity3D approach had the biggest time improvement between the two runs at 94% less than the original time. This means that beside the Unreal Engine 4, since the assumption was that the knowledge of the Horde3D engine
was advanced, Bullet Physics had the toughest learning curve. This is also demonstrated by ratio of learning resources for Horde3D and Bullet Physics, 2 vs 14.

6.2 Productivity

The least time for the final run was spent on the C++ version. This is the result of initial knowledge of programming and Horde3D but is also related to the fact that the C++ approach required minimal UI interaction. It was merely limited by the speed at which the code was designed and written, the other two approach, however, required additional UI interactions to achieve the same goals. The UI interaction is more friendly towards designers and artists, but for a programmer they become a hurdle. This also explains the differences between the first and final run as the UI interactions lead to inefficiencies which slowed down the process, and increased the final development time.

The hypothesis that a data-driven approach would allow for slimmer development times, thus, was proven wrong. Working with tools and their UI creates a barrier between the creator and the implementation where hard-coding the behaviour allows for access to the bones of your application. Because the programmer designs and implements the behaviour he does not have to take into account application specific requirements. Having to work with tools and abiding their specific set-up and systems in my opinion formed an overhead. Having to switch between environments, mindset and setting up behaviour in different ways e.g. visual scripting, placing objects and setting up links between objects put a strain on the development speed. With the C++ approach the behaviour was built from the ground up and a visual mind-map of every working part was created, which allowed for thinking ahead one or multiple tasks at a time.

The C++ approach was the fastest approach but taking into account the theories of Absolute and Comparative advantage (see section 3.4), it can be argued that it is not the best approach. Most of the time was spent on technical tasks, this means the time of the programmer was used for the implementation of all the behaviour. Comparing this to the Unreal Engine 4 and Unity3D approach shows that when available, delegating tasks to artists and designers will allow the programmer to spend respectively 67% and 58% of his time on other technical tasks.
The share of tasks that can be delegated to artists and designers with the Unreal Engine 4 approach can possibly grow towards 100%, because the visual scripting system has a shallow learning curve. This enables designers and artists to pick it up quickly and become 40% more efficient allowing the programmers to focus on improving or extending the engine technology. The Unity3D approach, however, still has a hurdle since C# does not have a shallow learning curve, quite the opposite actually, and it would take too much time to master it for it to become effective. The tools provided by Unity and Epic games impact the amount of interruptions, which could lead to a decrease in lost time and increase in overall productivity.

6.3 Bias

The case-study was performed having a technical background, so despite not having prior experience with Bullet Physics, Unity3D and the Unreal Engine 4, there is a sense of bias. The knowledge of structuring and creating gameplay systems was a significant advantage for creating the project. Sequentially performing the projects in the order of Unreal Engine 4, Unity3D and the C++ approach gave an advantage on the two latter during the first run. Because the project had already been split up into tasks for the prior approach(es) it was easier to perform the second and third approach. This benefit, however, was negated during the final run due to the experience from the first run and list of steps.
Chapter 7

Conclusion

Taking a data-driven approach to game development is the first step in a large process. A process that will require expertise in the field of data-driven systems, creation and maintenance of a powerful tool-set and team members that are able to work with them. The step, however, is one forward as the advantages in theory will outweigh its challenges and lead to empowerment of the development team.

A case-study was set up to weigh this hypothesis and with the use of three different pieces, or a combination, of middleware: Horde3D; Bullet Physics, Unity3D and the Unreal engine 4 a project was (re-)created. Performing the three case studies allowed for insight on how a data-driven approach impacts the productivity and learning curve during a development cycle in comparison with the old-fashioned, hard-coded, approach.

The learning curve of the middle-ware impacted the initial time lost in becoming productive with the tools. The user interface has the most influence on the usability of a tool, if it is too hard to comprehend the tool will decrease the productivity due to inefficiencies as was the case with the Unreal Engine 4. Since the Unity3D project was created with both tools and C# the final run was faster compared to the Unreal Engine 4.

Hard-coding gameplay behaviour with the use of Horde3D; Bullet Physics, however, surprisingly yielded the fastest results, but it would not be viable for the scope of current generation game projects. Residing to data determining the run-time behaviour is a more feasible approach to the development of current generation games. It allows for higher productivity due to the improved division of tasks across the development team and decrease in the amount of interruptions occurring throughout a day.
Data-driven systems have emerged and they have started to settle into the video games industry. They will, however, remain subject to improvement over the upcoming years. Because the demand for content will increase and the time-span between two releases shrink, re-usability and flexibility are a must to remain feasible as an industry. A hard-coded approach, thus, is not viable for the development of current-generation games on a large scale and data-driven will be the way to go.

A data-driven approach to game development requires utilizing tools, however, a cluttered UI and context shifting can lead to inefficiencies. This means the tools require an intuitive UI in both the aspects of usability and context a point that may, in the future, be looked into and improved. The Unreal Engine 4 has made a, relatively successful, attempt at creating middleware that is accessible for the variety of roles within a development team: designers, artists, programmers etc. Unity3D, however is an excellent approach for game development with a small but seasoned technical team. Building upon both Epic Games and Unity’s work is essential for coming closer and closer to the holy-grail engine.

7.1 Further Research

The research has shown some significant differences across the three approaches but there still is territory to be explored. Because the case-studies for this research have been performed by a one-man team the actual time gain from a data-driven approach was hard to measure and prove. The scope of the project has also been limited to gameplay behaviour and not taking into account the final rendering, audio and game-feel. Measuring the performance of a multi-person team with the different approaches would give greater insight into productivity and learning curves, taking a look at other measurement domains would also be interesting.

A domain that could be researched is flexibility of the different approaches, how costly is it time wise to change various a parts of the game e.g. gameplay, sounds, UI, visuals etc. This in my opinion is one of the domains in which a data-driven approach will prove to be superior. Due to their data-driven core, swapping a model is only a matter of changing it through the UI, and a part of a level could be altered by a level-designer using an editor. Assuming that the value of having tools to alter data in the end will yield a significant productivity increase.
Appendices
.1 Learning resource list Unreal Engine 4

- https://docs.unrealengine.com/latest/INT/Engine/Content/FBX/ImportOptions/index.html
- https://docs.unrealengine.com/latest/INT/Resources/ContentExamples/Materials/index.html
- https://docs.unrealengine.com/latest/INT/Shared/Editor/MapErrors/index.html
- https://docs.unrealengine.com/latest/INT/Engine/Physics/Vehicles/VehicleContentCreation/
- https://docs.unrealengine.com/latest/INT/Engine/Content/Types/StaticMeshes/LightmapUnwrapping/index.html
- https://www.youtube.com/watch?v=9yrHqMBynlw
- https://www.youtube.com/watch?v=t82xeOPW6Z4
- https://www.youtube.com/watch?v=7gwU0UPENA
- https://www.youtube.com/watch?v=mikG9ZuM2gY
- https://www.youtube.com/watch?v=Eh03YShs9L8
- https://www.youtube.com/watch?v=ig1ZcSaWjcI
.2 Learning resource list Unity3D

- https://www.youtube.com/watch?v=TOOi2XGFRX0
- https://www.youtube.com/watch?v=W1Qk1fFsqwE
- https://www.youtube.com/watch?v=hIWb7jpcyA
- https://www.youtube.com/watch?v=eDlDhzGpeps
- https://www.youtube.com/watch?v=ztfhn42pRGA
- https://www.youtube.com/watch?v=a6LgepzXIjw
- http://forum.unity3d.com/threads/example-project-wheel-collider-car.12751
- http://docs.unity3d.com/ScriptReference/WheelCollider.html
- http://docs.unity3d.com/Manual/class-WheelCollider.html
- http://forum.unity3d.com/threads/car-is-pulling-to-the-right.57902/
• http://forum.unity3d.com/threads/way-to-focus-scene-view-on-selected-object.766/
• http://docs.unity3d.com/ScriptReference/TextureImporterSettings.html
• https://www.youtube.com/watch?v=2HYbICYhy_w
• http://docs.unity3d.com/ScriptReference/Mathf.InverseLerp.html
• http://forum.unity3d.com/threads/importing-object-with-2-uv-sets.128359/
• http://answers.unity3d.com/questions/268931/car-movement-basics.html
• https://www.youtube.com/watch?v=ZUAIlSsVwfg
• http://answers.unity3d.com/questions/544596/switching-multiple-cameras.html
• http://answers.unity3d.com/questions/257774/car-handbrake.html
• http://unity3d.com/learn/tutorials/modules/beginner/ui/ui-scroll-rect
• http://unity3d.com/learn/tutorials/modules/beginner/ui/rect-transform
• http://unity3d.com/learn/tutorials/modules/beginner/ui/canvas
• http://unity3d.com/learn/tutorials/modules/beginner/ui
• http://unity3d.com/learn/tutorials/modules/beginner/ui/text
• http://docs.unity3d.com/ScriptReference/GUIStyle.html
• http://docs.unity3d.com/Manual/gui-Controls.html
• http://docs.unity3d.com/Manual/gui-Basics.html
• http://docs.unity3d.com/Manual/class-GUIStyle.html

• http://docs.unity3d.com/ScriptReference/Transform-eulerAngles.html

.3 Learning resource list Horde3D

• http://www.horde3d.org/docs/html/_pipeline.html

• http://www.horde3d.org/docs/html/_api.html

.4 Learning resource list Bullet Physics

• http://bulletphysics.org/Bullet/BulletFull/classbtRaycastVehicle.html

• http://www.raywenderlich.com/53077/bullet-physics-tutorial-getting-started

• http://bulletphysics.org/mediawiki-1.5.8/index.php/Vehicles

• http://bulletphysics.org/mediawiki-1.5.8/index.php/Hello_World

• http://www.bulletphysics.org/mediawiki-1.5.8/index.php/Collision_Shapes

• http://bulletphysics.org/Bullet/BulletFull/classbtCompoundShape.html#details

• http://bulletphysics.org/Bullet/BulletFull/btRaycastVehicle_8cpp_source.html

• http://bulletphysics.org/mediawiki-1.5.8/index.php/Rigid_Bodies

• http://bulletphysics.org/mediawiki-1.5.8/index.php/Collision_Shapes

• http://bulletphysics.org/mediawiki-1.5.8/index.php/MotionStates
• http://bulletphysics.org/mediawiki-1.5.8/index.php/Coordinate_system

• http://bulletphysics.org/Bullet/BulletFull/classbtQuaternion.html#a8bd5d699377ba585749d325076616ff

• http://bulletphysics.org/mediawiki-1.5.8/index.php/File:Vehicle_wheelattach.jpg

• https://code.google.com/p/bullet/source/browse/trunk/Demos/VehicleDemo/VehicleDemo.cpp?r=2430
Bibliography


[22] Cryengine flow graph editor. [http://docs.cryengine.com/display/SDKDOC2/Flow+Graph+Editor](http://docs.cryengine.com/display/SDKDOC2/Flow+Graph+Editor).


[40] Matt Gilgenbach. The case against data-driven design.


[101] Jeff Ward. Data-driven is half the battle.


