



**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**  
BRNO UNIVERSITY OF TECHNOLOGY



**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**  
**ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ**

FACULTY OF INFORMATION TECHNOLOGY  
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

# MULTIPLATFORM GAME DEVELOPMENT USING THE UNITY ENGINE

MULTIPLATFORM GAME DEVELOPMENT USING THE UNITY ENGINE

**DIPLOMOVÁ PRÁCE**

MASTER'S THESIS

**AUTOR PRÁCE**

AUTHOR

**Bc. ROMAN JAŠEK**

**VEDOUCÍ PRÁCE**

SUPERVISOR

**Ing. RUDOLF KAJAN**

BRNO 2014

## Abstrakt

Tato práce se zabývá možnostmi herního vývoje pro vícere platformy pomocí engine Unity 3D. Rozebírá různé aspekty multiplatformního vývoje na počítačích i mobilních zařízeních. Důraz je kladen na analýzu tvorby her současně pro více platform a problémy spojené s tímhle přístupem. Práce poskytuje možnosti řešení problémů, které vyvstanou při použití tohoto postupu, pomocí nástrojů, které poskytuje Unity 3D. Analýza se zabývá zejména na zvyšování výkonu her za použití metod dostupných na všech platformách vybraných pro testování. Tyhle vylepšení zahrnují způsoby jak snížit práci ve scéně a naopak zvýšit počet vykreslení za sekundu při zachování stejné vizuální kvality. Práce také nabízí pohled do minulosti tohoto odvětví a předpoklady o jeho příštím směřování.

## Abstract

This thesis deals with multiplatform game development using the Unity 3D engine. It approaches various tasks of development on both desktop and mobile devices. The focus of the thesis is to analyze the aspects of developing for more platforms at once and the challenges connected with this approach. It provides solutions of the stated issues using the tools provided by Unity 3D. The analysis focuses mainly on increasing the performance of games by means that are available on all of the platforms selected for testing. These improvements include the ways of decreasing the work within a scene and increasing the frame rate while keeping the same level of visual quality. The thesis also offers a look at this area and its historical and future development.

## Klíčová slova

Unity 3D, multiplatformní, hry, kombinování 3D objektů, zapékání osvětlení, optimalizace uživatelského rozhraní, Windows, Windows Phone, Windows RT, Android

## Keywords

Unity 3D, multiplatform, games, mesh combining, light baking, user interface optimizations, Windows, Windows Phone, Windows RT, Android

## Citace

Roman Jašek: Multiplatform Game Development Using the Unity Engine, diplomová práce, Brno, FIT VUT v Brně, 2014

# Multiplatform Game Development Using the Unity Engine

## Prohlášení

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením Ing. Rudolfa Kajana.

.....  
Roman Jašek  
May 28, 2014

## Poděkování

Rád by som poďakoval za odbornú pomoc môjmu vedúcemu, Ing. Rudolfovi Kajanovi.

© Roman Jašek, 2014.

*Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.*

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Game Engines</b>	<b>5</b>
2.1	Cross-platform independent Game Engines . . . . .	5
2.1.1	Unity . . . . .	7
<b>3</b>	<b>Multiplatform development</b>	<b>9</b>
<b>4</b>	<b>Developing on different platforms</b>	<b>11</b>
4.1	Platforms used . . . . .	11
4.2	Windows . . . . .	13
4.3	Windows Phone . . . . .	14
4.4	Windows RT . . . . .	14
4.5	Android . . . . .	15
4.6	In-game logs . . . . .	15
4.7	Platform-specific functions . . . . .	16
<b>5</b>	<b>Combining meshes</b>	<b>18</b>
5.1	Approaches to combining . . . . .	18
5.1.1	Combining speed . . . . .	19
5.1.2	Large amounts of vertices . . . . .	21
5.1.3	Multiple textures . . . . .	22
5.1.4	Other components of game objects . . . . .	24
5.2	Custom script . . . . .	25
<b>6</b>	<b>Baking lights</b>	<b>28</b>
6.1	Performance testing . . . . .	29
<b>7</b>	<b>Custom shaders</b>	<b>32</b>
<b>8</b>	<b>User interface</b>	<b>34</b>
8.1	GUI scripting . . . . .	34
8.2	GameObjects . . . . .	34
8.3	Performance testing . . . . .	36
<b>9</b>	<b>Tutorial</b>	<b>38</b>



<b>10 Demo scene</b>	<b>39</b>
10.1 Mesh combining tests . . . . .	40
10.2 Baking lights tests . . . . .	45
10.3 User interface tests . . . . .	47
10.4 All optimizations combined . . . . .	48
<b>11 Conclusion</b>	<b>50</b>
<b>A DVD contents</b>	<b>53</b>
<b>B Tutorial examples</b>	<b>54</b>

# Chapter 1

## Introduction

Video games have been here with us for a couple of decades. People started making games basically since the beginning of the computer era. At first, these were only some scientific projects used as either simulations or technology demonstrations. For example, a war game called simulating Hutspiel created in 1955. The video game industry was not taken seriously for a while as people thought of computer games just as toys for kids. In the beginning, small teams of the fans of technologies usually made games and the games were aimed at a specific hardware. Back then, the resources and code were highly specialized for the game with very small re-usability. Every game was basically a separate product.

When games became more significant they started to be produced for the general PC architecture. That way they were playable across a wide variety of computers, which helped to spread them even more. This helped to standardize the development process. Developers could target a wider audience with their titles. The gaming industry was growing fast and new titles started to be increasingly complex. Developers started to try to reuse parts of their software in multiple projects. Sequels to game titles were being created which shared many of the assets from the previous games. Game developers were able to build on top of their previous work. They also had access to reviews and could try to improve their titles.

Game engines came to existence this way. However, the line between a game and its engine is often blurry. Some engines make a reasonably clear distinction, while others make almost no attempt to separate the two [4]. At first they were based on the first-person shooter genre. Freespace is considered one of the first 3D engines. It was developed by Incentive Software and first used in their 1987's game Driller. This engine was an in-house work and was never licensed outside of the studio. In the 1990s id Software came with their successful engine id Tech 1 and allowed other studios to license it. This was the era of Doom-like games.

Since then a few decades passed and nowadays this industry is rivaling Hollywood with the income of \$67 billion in 2012 and still growing strong. There are huge titles being produced with budgets taking as much as hundreds of million dollars and teams of hundreds people working on them.

With the spread of games to the culture, many new devices appeared for gamers to use. These included gaming consoles as well as hand-held devices. With the occurrence of these technologies developers started to be focused on single platform development like in the first stages of game development. However, the companies felt the need to being able to focus on more platforms to cover a larger market than just focusing on a single platform development.

Later we can say that virtually all game engines contain a familiar set of core com-

ponents, including the rendering engine, the collision and physics engine, the animation system, the audio system, the game world object model, the artificial intelligence system, and so on.[4]

More recently, there has been a boom of smartphones. It brought the power of a handheld gaming device to hands of ordinary people via their phones. Now the game studios do not focus solely on people who are ready to pay for special devices for their gaming purposes. They also need to work with the desires of casual gamers, who play games just occasionally. Many different operating systems are running on today's smartphones. Moreover, the developers were hit by the question which platform to focus on. Which one of them brings them the highest revenue and the most users?

This was the situation a few years ago. Since then the game engine creators have gone forward to bring new game engines. To give developers tools to be able to build their games for multiple platforms. Platforms running on various kinds of devices are included.

Unfortunately, the multiplatform development is not supported sufficiently at the time when this thesis is being made (May 28, 2014). Unity is the game engine of choice for the purpose of this thesis. It currently has a still growing support for the smartphone devices. However, the support for these new devices is not yet on par with the PC and game console development possibilities.

Thus, this thesis makes an effort to elaborate on the current ways of multiplatform development using the Unity engine. It also tends to describe the tools and procedures needed to achieve seamless development of games for as many platforms supported as possible.

# Chapter 2

## Game Engines

We have gone through a brief history of video games and game engines. We can now focus on the present situation, which has a lot to do with game engines.

There are a lot of game engines out there nowadays with Wikipedia currently listing 220 different engines. Therefore, it is quite difficult to choose the right one for the task provided. There are highly specialized engines used for specific platforms and/or game genres, which are not suitable for the general usage.

### 2.1 Cross-platform independent Game Engines

For the purpose of this thesis, we need a game engine that is capable of development for different platforms. We want to emphasize that one of the goals of this thesis is to find solutions to problems that accompany multiplatform development. As we will see in the next chapters this is especially evident when developing for both the desktop and mobile systems. That is why this thesis will consider only game engines that support both PC and mobile platforms. In table 2.1 we can see a list of game engines that meets the said criteria. The platforms listed in red are available only in a paid version of the engine (these engines usually come in a free version with some limitations and one or more paid professional versions):

Engine	Desktop	Mobile	Consoles	Web
Antiryad Gx	Windows, Mac OS X, Linux	Android, iOS	PlayStation 3	-
Edgelib	Windows, Mac OS X, Linux	Android, iOS, Maemo, Sym- bian, Windows Mobile 6	-	-
Game Editor	Windows, Mac OS X, Linux	Android, iOS	-	-
GameKit	Windows, Mac OS X, Linux	Android, iOS	-	-

Gameplay	Windows, Mac OS X, Linux	Android, iOS, BlackBerry 10	-	-
Ignifuga	Windows, Mac OS X, Linux	Android, iOS	-	-
Jumpcore	Windows, Mac OS X, Linux	Android, iOS, WebOS	-	-
Matali Physics Engine	Windows, Mac OS X, Linux	Android, iOS, Windows Phone, Windows RT	Playstation Vita	Silverlight
MonoGame	Windows, Mac OS X, Linux	Android, iOS, Windows Phone, Windows RT	OUYA	-
Ogre 3D	Windows	Android, Win- dows Phone, Windows RT	-	-
OpenFL	Windows, Mac OS X, Linux	Android, iOS, BlackBerry	-	HTML 5, Flash
Orx	Windows, Mac OS X, Linux	Android, iOS	-	-
PlayN	Windows, Mac OS X, Linux	Android, iOS	-	HTML 5
Project Anarchy	Windows	Android, iOS	-	-
Proton SDK	Windows, Mac OS X, Linux	Android, iOS, WebOS, Black- berry Playbook	-	Flash
Simple Direct- Media Layer	Windows, Mac OS X, Linux	Android, iOS	-	-
Unity 3D	Windows, Mac OS X, Linux	Android, iOS, Windows Phone, BlackBerry, Windows RT	Xbox 360, Playstation 3, Nintendo Wii, Nintendo Wii U	Web Player, Google Na- tive Client

Table 2.1: Game engines - platform support

Most of these 17 engines have very good support for the desktop platforms, including Windows, Mac OS X and Linux. Here we can see that the people creating multiplatform game engines do not have only Windows in mind when developing for desktop as it used to be in the past as Windows is considered the usual gaming platform among PC gamers.

In terms of mobile development all the selected engines support both Android and iOS platforms. These are the most widespread mobile operating systems nowadays with Android leading the smartphone market and iOS coming in second place. Therefore, it is understandable that game engine creators want to support these mobile platforms. Windows Phone is next which again resembles the current market situation as this system is gaining traction. Windows Phone is now „the third largest OS across Europe with 10% - more than double its share compared with last year“[1] according to Kantar (one of the world’s largest insight, information and consultancy group). This system managed to take 3.6% of sales in the 3rd quarter of 2013 while growing the amount of devices sold by 156% year-on-year. This thesis considers these 3 mobile platforms as significant as the rest of the platforms are either already deprecated (Symbian, Maemo, Windows Mobile 6, WebOS, BlackBerry Playbook), or so far failed to catch up market attention (BlackBerry 10, Windows RT).

The rest of the table deals with the operating systems that are a welcome bonus but are not necessary for the purposes of this thesis.

However we can see that only a few of the engines have some kind of support for console development, usually supporting some of the 7th generation consoles (Xbox 360, Playstation 3 and Nintendo Wii) and some of them have support for handheld consoles (Playstation Vita, Playstation Portable, Nintendo DS). The only 8th generation console that is currently supported is the Nintendo Wii U supported by Unity. The rest of these consoles (Xbox One and Playstation 4) is missing as these are very new and so far the console development has been affected by high license fees. Consoles have been mostly a domain of large game developing companies. Nevertheless, the independent developing community is growing and the companies making these consoles realize that. Therefore, they are currently releasing news about the support for independent developers and we can assume that these developers will use the 8th generation consoles much more than the previous one. That is why we expect that the console support will grow in the following years even among the free game engines.

The last segment - web development is also on the rise with HTML 5 gaining better support among web browsers and the appearance of Google Chrome as a platform based built off of a web browser. Moreover, even Javascript is getting faster with better implementations than it used to have in the past. That is why it is good to see that some of the engines already have the support for web development included.

### 2.1.1 Unity

After going through the platforms, we want to elaborate more on why this thesis chose Unity as the implementation engine. The support for the various platforms is definitely a welcome feature of Unity. The engine was first released in 2005. Originally it started on Mac OS but as it gained popularity it spread to all of these platforms. The current version was released in December 2013 as version 4.3.2.

Until recently, Unity has been mostly used for 3D games but in the last version, it gained support for 2D game development as well and thus giving us an opportunity to provide examples of both 2D and 3D games. On the contrary to this there were some 2D game engines listed in the table 2.1 - Game Editor, Ignifuga, Jumpcore, Orx, and PlayN.

Another part of the decision is the current support for the rendering engines among the game engines. The market of smartphone devices and tablets is currently dominated by Android and iOS. Both of these systems use OpenGL ES for rendering. Therefore, it

is natural that most of that are oriented on multiplatform development support Android and iOS. However, Microsoft uses Direct3D in its operating systems and this is true for Windows Phone and Windows RT as well. These platforms are currently gaining market share and this thesis wants to focus on them as well. Therefore, it is great that Unity already provides support for these systems.

The last and the most important reason why we chose Unity is its support for game developers. The development cycle is fast and the developers of Unity listen to the community and try to implement the desired functionality. Unity also provides its own asset store, which contains over 4,500 models, over 1,200 textures and materials and more than 600 audio effects. These are available at various prices ranging from free to hundreds of dollars. This is especially great for the independent programmers as it allows them to develop without the need for various artists. There are also over 300 completed projects available for purchase, which demonstrate how to work with Unity and can be bought and modified by other developers to learn from them. In addition, the community around Unity is huge. The Unity forums have already about 400,000 registered members who posted more than 1,400,000 posts and are a vital source of information. Ogre is the engine that gets the closest to these numbers with a little over 30,000 members and around 430,000 posts.

These are the reasons for us to choose Unity - a great support for the multiple platforms, support for both 2D and 3D game development and a large community of people already working with the engine and providing information and assets needed for game development.

## Chapter 3

# Multipatform development

There are various approaches to multipatform development. We can generally divide it into two types:

- interpretation or pre-compilation of code
- individual compilation for each platform

In terms of game development, the first option is usually used in HTML 5 games. Web browsers interpret JavaScript and it does not need to be compiled for different architectures.

The second approach is the more common in gaming. Most of the game engines support C++ as the programming language. Moreover, C++ is a compiled language, so the engines need to be able to do this for all the platforms they support.

Nevertheless, these are not the only differences that developers face when decide to develop for multiple platforms. We already mentioned the different rendering engines. However, even if the engine is capable of solving the different handling of rendering it will always suffer from the different hardware that it runs on. The desktop platforms have always had the upper hand in terms of hardware equipment. The mobile systems need smaller parts and they need to be much more power-efficient because they run on batteries. When we develop for mobile devices we cannot count on all the features that are naturally available on desktop. However, there are practices that solve this issue while keeping support for multipatform development.

The most simple of these practices is called the lowest common denominator. As the name suggests we will use just those parts of the engine that are common for all the targeted platforms. Therefore, if we encounter a feature that is not available at one of the platforms we do not use it.

Another option is to use different code for different applications. We can do this in two ways:

- separate codebases
- common core, separate specifics

The first way is the old approach that required developers to create a separate version of a program for every single platform. As we have already stated we want to avoid this, so we will not consider this option anymore. Nevertheless, the second option is much more reasonable. The main idea is that we create a core that implements the functionality, which is common for all the platforms. On top of this core we implement the functions that are



platformspecific. This is the way of development that is usually used nowadays. As there are many differences between platforms and the engine is not capable of handling them all. We will take a close look at these in the next chapter.

# Chapter 4

## Developing on different platforms

We are finally getting to the challenges that are present in development with Unity. First, we will go through the differences in development among the platforms selected for testing. Then we will take a look at the multiplatform development issues and ways that we can deal with these.

### 4.1 Platforms used

For testing purposes, we chose these 4 platforms:

- Windows
- Android
- Windows Phone 8
- Windows RT

We will therefore test the solutions provided by this thesis on the platforms listed above. We chose Windows because it is the most widespread desktop platform and the one mostly used by gamers. Therefore, it makes sense that developers who intend to develop for multiple platforms would use Windows as one of them. Similar to this Android is the platform used on most of the mobile devices and is thus the platform of choice for many mobile developers. Another difference is that Unity uses different graphic API for each of these platforms. While Windows is well known for the advances in graphics using Direct3D, Android uses OpenGL. These can also produce various differences when dealing with multiplatform development.

The last two platforms - Windows Phone 8 and Windows RT - were chosen because of their novice status in the Unity family of supported platforms. These platforms were added just in 2013 and are therefore among the newest additions to Unity. Windows Phone and Windows RT are however interesting even from the graphics API point of view. They are a part of the Microsoft ecosystem of operating systems so they run on Direct3D. Unfortunately, the platforms are not yet properly united. This brings the aspect of different versions of Direct3D running on different systems. We will discuss this more in chapter 7. Then we can say that we have three different approaches to drawing graphics on the screen. However, why did we choose Windows RT? The WinRT environment can run on the same system as Windows 8. Therefore, it is quite interesting to see the differences when our

game runs on the same hardware just using different APIs. We will talk about this more when we investigate the performance of different platforms.

Before jumping on to the comparison of all the abilities of different platforms let us take a short look at what hardware we used for testing purposes.

For testing the Windows and Windows RT version of the application, we used a laptop computer. The specifications of the computer are the following:

- CPU: Intel Core i5-2520M, 2.5 GHz
- GPU: Intel HD Graphics 3000
- RAM: 8 GB
- Resolution: 1366 x 768 pixels

You can see that it is not exactly a gaming machine as it only contains an integrated graphics card. On the other hand it is not, a low end PC neither.

To test the Android platform we decided to use two different devices with chipsets from two different manufacturers to see the difference. We tested Android on a Nexus 4 phone and a Nexus 7 tablet. Both of these phones were announced in the fall of 2012, so they are not the newest devices. Both of them are however in the higher-end spectrum of the Android devices.

The full specifications of Nexus 4 are:

- CPU: Qualcomm S4 Pro APQ8064, 4 x 1,5 GHz
- GPU: Adreno 320
- RAM: 2 GB
- Resolution: 1280 x 768 pixels

Whereas the Nexus 7 tablet comes with the following specifications:

- CPU: nVidia Tegra 3 T30L, 4 x 1,2 GHz
- GPU: ULP GeForce
- RAM: 1 GB
- Resolution: 1280 x 800 pixels

We can see that Nexus 4 is the device with faster processor and higher RAM capacity. However, as we will see later, the main difference is between the chipsets themselves. The architecture of nVidia Tegra chips is different than the chips by Qualcomm, so we can expect some variations when we will compare these two devices. To distinct between Nexus 4 and Nexus 7 in tables we will use the abbreviations N4 and N7 respectively.

The phone used for Windows Phone tests is a Nokia Lumia 920. The specifications are:

- CPU: Qualcomm S4 Plus MSM8960, 2 x 1,5 GHz
- GPU: Adreno 225
- RAM: 1 GB

- Resolution: 1280 x 768 pixels

As can be seen from the specifications, this phone is not one of the high-end devices in terms of mobile devices. However, the Windows Phone phones usually do not tend to compete in the hardware specifications and focus more on software optimizations. Therefore, when we take a closer look at the world of Windows Phone devices we actually find out that this phone is one of those with higher hardware specifications in this limited spectrum of the mobile world. When it was released in the fall of 2012 it was meant as one of the high-end Windows Phones. Therefore, we can say that the testing phone is on the higher end of the specifications spectrum in terms of Windows Phones. However, compared to the Nexus 4's specifications, we can say that this device lacks in raw power when facing a high-end Android device.

After going through the platforms used in this thesis and the hardware used for testing these platforms we can look at how to develop for said systems. We could not find a comprehensive list of instructions about how to develop for all of these platforms. There were some tutorials available for separate platforms and some papers. We will therefore try to write down my findings about how to get a Unity game running on these platforms and what tools can be used to develop for them.

## 4.2 Windows

We naturally develop games on a desktop system, in my case it is Windows. We use Visual Studio as my IDE of choice and my preferred language is C#. This is convenient as Unity supports C# and we can open the scripts in Visual Studio directly from the Unity environment. The deployment to Windows is also rather easy. We just need to build the application and get an executable file and a folder with data needed to run the game.

Unfortunately, there is currently no free way to debug a running Unity game from within Visual Studio. The usual way of debugging a game running on Windows is to use MonoDevelop. This is a cross-platform IDE, designed for .Net languages. When we load a project into Monodevelop and start debugging, it opens the Unity environment with this project. Then if we launch the game in Unity editor, we can use the debugger of Monodevelop. We can use breakpoints, step through the code and see the values in variables very much like in Visual Studio. Unity displays the log messages in the console in Unity environment.

A paid Visual Studio extension called UnityVS is however available. It allows us to debug from within Visual Studio. It has a 14-day trial version so we decided to try it. It installs as a Unity package and then needs to be added to the project. It adds a new menu option „UnityVS“ and we can open the project in Visual Studio using this menu. Then we can debug the game that runs within the Unity editor from Visual Studio. The log messages are displayed right in Visual Studio in its error messages. This extension costs \$99 perpetual for independent developers. The people behind UnityVS also state that they are working on an education license for students and teachers. Therefore, we think this extension could be considered for future development if we prefer Visual Studio as our environment.

### 4.3 Windows Phone

The first of the mobile platforms that we use is Windows Phone. To compare this with the Windows Phone platform there are a few requirements needed in order to get the application running on a device. We need to have the following on top of the Unity installation:

- Windows Phone SDK 8.0 or higher
- Windows 8.0 or higher
- Windows Phone developer account

Until recently the only option to get a developer account for Windows Phone one had to pay a yearly fee. However, Microsoft now offers a free way to get a developer license. By signing up for an App studio account at <http://appstudio.windowsphone.com/> we are allowed to developer unlock 1 Windows Phone device and load up to 2 custom applications on it. This way we can develop and test games on our Windows Phone for free. The paid option is to get a regular Windows Phone account and be able to unlock three devices with up to 10 applications being side-loaded on each of those phones. This currently costs \$19 annually.

Once we have these installed, we can start to develop for this platform as well. We design our game in Unity just as we do for all the platforms. Then we have two options of launching the game. We can press the „Build & Run“ button in our Unity environment, which launches the game on a device. But if we want to be able to debug a game we can use Visual Studio to do so. We first need to build our game in Unity; it creates a Visual Studio solution with a project named as the Unity project. Nevertheless, we also have to add another project to be able to debug the C# scripts. This project is called AssemblyCsharp and is located in the Unity project folder. Once we add this project to the solution, we can see that all the C# scripts are within the Assets folder. Then we can debug them just as we are used to from other C# programs. Either we can deploy to an emulator or directly to a device (we need a developer unlocked device for this).

There is one more difference between debugging a Windows and Windows Phone application. The log does not display within the Unity editor. Therefore, if we are using logging we need an external tool to take a look at the output of the log. The program that allows us to do this is called Windows Phone Power Tools. We can use it to browse through the developer applications' local storage on our Windows Phone device. However, we do not have an access to the log while the game is running as the file is locked by the application.

### 4.4 Windows RT

We would like to specify the use of this platform - we do not have a Windows RT device available now. Luckily, Microsoft provides a very useful simulator for the Windows 8 platform. Therefore, we are using the simulator for the time being for testing Windows Store version of the Unity game. It is not the precisely the RT version of Windows as we have an x86 processor and RT runs on ARM. However, it is still a Windows Store application with its limitations. When we think of Windows RT, assume it is the same platform on slightly less powerful hardware and with the requirement that we use a remote debugger that should cover most of our architecture differences [7].

Except for Unity, we are going to need these to be able to develop for Windows RT:

- Windows 8.0 or higher
- Windows Store developer account

We would like to clarify the use of another developer account. This was true until November 2013. However now Microsoft merged the Windows Phone and Windows Store developer accounts. Therefore, these are basically the same and we can use either the free option or the paid \$19 account to develop for Windows RT alongside Windows Phone.

The debugging is very similar to the Windows Phone. We again need to build our game in Unity and open the produced solution in Visual Studio. Then we add the Assembly-Csharp project and we are ready to debug. We can launch the application on the local machine, in the simulator or on a remote device.

In terms of logging, it is easier to access the log file then the process used on Windows Phone. We can find the log file produced by the game in "C:\Users\<USER>\AppData\Local\Packages\<PKGID>\TempState\UnityPlayer.log". The file is not locked for reading while the game is running so we can open it and see the logging information on the go.

## 4.5 Android

The last platform that we selected for testing is Android. This is a very different platform to the Microsoft's ones. It is a Linux based system developed by Google. To test on this system we will need another set of tools:

- Java Development Kit
- Android SDK
- ADB USB driver for debugging on a device <sup>1</sup>
- Enable USB debugging on the device

Android collects the debug output of applications in a series of circular buffers. This is also the case for Unity's log. We will use a tool from the Android SDK to access the log on an Android device. It is called ADB shell and is located in the platform-tools folder in the SDK. We will use the utility logcat within the ADB shell. The call from a command prompt is as follows:

```
$ adb shell
$ logcat
```

Once we launch logcat we will see the live output from the log buffers including the output from our application.

## 4.6 In-game logs

There is one more way of going through the logs that we would like to mention. It is called Log Viewer and is a Unity Asset. It was created by dreammakersgroup and we can find it in the Unity Asset Store. This free tool allows us to see the logs directly in our game. We

---

<sup>1</sup><http://developer.android.com/tools/extras/oem-usb.html>

just need install the asset to our Unity project, create an empty game object in Unity, and attach the InGameLog script to this object. Once we run our application with the script included, we activate it with a circular gesture. It then shows us the live feed from our logs as seen in figure 4.1.

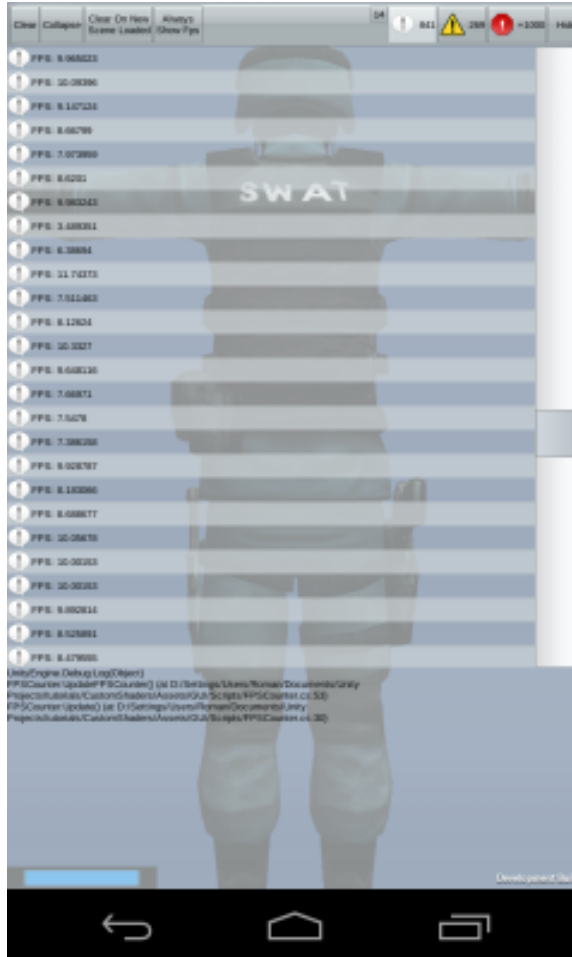


Figure 4.1: Game log displayed on an Android device

This way we can debug a game on a device even without connecting it to our development computer.

## 4.7 Platform-specific functions

In the previous chapter, we stated that we want to be able to use some platform-specific functions. These include various sensors and controlling devices used throughout within the game. Mobile applications usually run within a sandbox. This way the apps have only certain ways to access the local storage. Moreover, these ways vary a lot among platforms. Another example of this are the different sensors that are usually available on mobile devices, including motion sensors (accelerator, gyroscope), GPS locators or even more exotic like thermometers. Some games can utilize these sensors for game control. However, there is not a single way to work with these sensors across different platforms. Each system allows a different way of handling input from these devices. That is where

platform dependent compilation comes in place. Unity supports platform defines that we can use to make sure that a part of code executes only on a specific platform. These include `UNITY_WIN`, `UNITY_ANDROID`, `UNITY_PS3`, `UNITY_WP8` and so on for all the different platforms. We can utilize them by stating an if macro followed by a platform define. Anything that lies within the `C#` if block will be executed only on the selected platform.



## Chapter 5

# Combining meshes

Rendering is an essential part of a game engine. We will now look at some ways of rendering optimization in Unity.

The biggest performance killer is the amount of times that the graphics API has to draw something on the screen. In addition, this is crucial especially on the mobile devices as they have very limited power compared to the PC.

To reduce the number of draw calls we want to combine the meshes together. We can do this for the meshes that share the same material and do not move independently. These can be some generic objects within our level that are not connected but share the same model and material.

One way to do this is to create the combined meshes directly in a modelling software. This however has the disadvantage of always drawing the whole combined mesh. Unity Pro includes a feature called occlusion culling - this allows for filtering out objects that are not visible by the camera and leaving them out of the draw call. So if we want to take advantage of this feature we do not want to combine meshes within a modelling software.

Unity has a way to batch these meshes on its own. To use this with static meshes that are not going to move in the game we can just set the static flag for these meshes and Unity will combine them together. We can batch meshes of any size if they are marked as static. Unity can use this feature even for batching moving objects. However, there are certain limitations. Unity does this only for objects with limited vertex count of 900. This amount even decreases if the shader used with the object uses more attributes like multiple UV values or tangent.

A different approach to this is to manually combine the meshes that we want and leave Unity's automatic batching alone. We can do this by using a script. Unity's `Mesh` class contains a method called `CombineMeshes()`. It can be used to create a new mesh based on the meshes provided in an array as a parameter.

### 5.1 Approaches to combining

There are some examples of how to use the `CombineMeshes()` and they were considered for the needs of this thesis. We will look at the ways they deal with various aspects of this optimization technique and analyze them.

In general, the scripts work in a similar way. We create a couple of objects that we intend to combine and we create a parent object that encapsulates all of these. Then we attach our script to the parent object. This way we have control over which objects we want

to combine. This can be helpful for example when used with occlusion culling, when we can combine just small areas where the similar objects appear. If we combined objects far away from each other, we would effectively defy occlusion culling, as the new larger mesh would be drawn as a whole even though some parts of it would not be visible. Therefore, it is needed to think about which object groups we want to combine together and which should stay separated.

We chose two scripts for further investigation of different approaches to mesh combining. They operate in different ways and therefore we considered both of them for use within this thesis. The scripts can be found in the Unity Asset Store by the names:

- Simple Mesh Combine<sup>1</sup>
- Draw Call Minimizer<sup>2</sup>

The first difference to note in terms of these scripts is the different way of combining the meshes. The Simple Mesh Combine script uses the *CombineMeshes()* method so it relies on the Unity's built-in mechanism of dealing with this problem. Draw Call Minimizer however uses a different technique. It combines the meshes manually without using the *CombineMeshes()* method. Both ways of combining meshes have their advantages as well as disadvantages. We will now investigate the speed of combining meshes in both distinctive ways and the aspects that they take into account.

### 5.1.1 Combining speed

While going through the various scripts we generally found two approaches to combining the meshes. This is also true for the scripts that we are analyzing. The approaches are as follow:

- Combining in the editor
- Combining on runtime

The first way is to do the combining before we actually run the game. This way we do not need to do any work while the game is running so it is quick to load. However, the disadvantage is that when we want to add some objects to the combined mesh we need to launch the script again and create the mesh. This is therefore similar to the solution described before when creating meshes in an external modeling software. The only difference is that we do this within the Unity environment so we do not need to use any additional tools. Still, anytime we add or remove an object we need to think if it is a part of a combined mesh and launch the script again. The Simple Mesh Combine script works this way.

On the other hand, the second approach is seamless. The script is run at the beginning of the game so it is run every single time. This however produces overhead at the beginning of the runtime of a game.

While the speed of combining meshes while the game is not running is not really that important (as it does not bother the developer and it does not need to be done every single time), the speed of combining meshes in runtime is important to us. We carried out the tests to determine whether it is viable to combine the meshes at the beginning of each run of

---

<sup>1</sup> <https://www.assetstore.unity3d.com/en#!/content/8748>

<sup>2</sup> <https://www.assetstore.unity3d.com/en#!/content/2859>

the game or whether this idea has to be abandoned. However if the Simple Mesh Combine script was run just from the editor without the game actually running it would provide it with an unfair advantage compared to the script that actually runs while all the other game aspects are being loaded. That is why we slightly modified the Simple Mesh Combine script to run at the beginning of the game, so both scripts were running at the same point. Thus we can see a direct comparison of the two approaches to mesh combining - using the *CombineMeshes()* method and using a custom way of putting the meshes together.

Tests were run on many instances of a single model of a barrel with 578 vertices which can be seen in figure 5.1. One of the uses of mesh combining is to combine many props in a scene so this model is a good example of that use.



Figure 5.1: Model used for testing the mesh combining scripts

We ran the scripts also with different models with different vertex counts. The differences between the amount of meshes used with the same vertex amount (i. e. 10 meshes with 1000 vertices and 100 meshes with 100 vertices) were not significant. Therefore, we could base the comparison the comparison of two scripts just upon using one model and still be conclusive.

The tests were run using the PC discussed in section 4.1 within a scene compiled for the Windows platform. The results of the measurements can be seen in table 5.1.

Meshes	Vertices	Simple Mesh Combine [ms]	Draw Call Minimizer [ms]
2	1156	2	562
10	5780	3	565
20	11560	5	570
30	17340	6	574
40	23120	8	578
50	28900	12	604
60	34680	13	608
70	40460	14	621
80	46240	16	623
90	52020	19	631
100	57800	20	633
110	63580	21	643
120	69360	35	803

Table 5.1: Performance comparison of mesh combining scripts

As can be seen from the table 5.1 the times for the Simple Mesh Combine script which uses the *CombineMeshes()* method are much lower than the custom approach of the Draw Call Minimizer. We can also see the significant bump in the times of both scripts when going from 63580 vertices to 69360. We will discuss the reason for this in section 5.1.2.

In terms of the Simple Mesh Combine script, we can see that the time of combining increases pretty much in unison with the amount of vertices being combined. It is not exactly a linear increase but it is quite close to being so.

On the other hand, we can see that the Draw Call Minimizer has a large overhead that clutters its performance as a whole. The difference between combining 1156 vertices and 63580 vertices is not very high in terms of the absolute runtime of the script. Even the lowest amount of vertices takes a significant time.

It is also worth noting that the scripts are run sequentially. So if we wanted to combine ten groups of meshes each consisting of 1156 vertices it would take up much more time then combining one group containing 11560 vertices. This also needs to be taken into consideration when thinking about which meshes we want to combine together and which not.

So far, we have discussed the speed of the two chosen scripts, each using a different approach to mesh combining. The Draw Call Minimizer which uses a custom approach to mesh combining showed up to be the worse option in terms of speed. Nevertheless, we still chose it for the comparison. Let us take a look at why it was considered to be used in this thesis.

### 5.1.2 Large amounts of vertices

As we could see in table 5.1 there was a sudden speed decrease when going from combining 63850 vertices to 69360 vertices. This is caused by a limitation of the graphic cards. The usual way of drawing a mesh works in the way that they receive a 16 bits describing its vertices. However, we can save only 65535 vertices to 16 bits. That is why the amount of vertices per mesh is limited to 65535, or by the *CombineMeshes()* method to 65000.

In addition, here is the first mayor difference between the ways that the two investigated scripts work. The Simple Mesh Combine script simply takes all the meshes, puts them in an array and calls the *CombineMeshes()* method onto this array. Therefore, it cannot deal with combining meshes that are larger than the said amount. If it tries to do so it does not crash the game, but the result will not be what we would expect. You can see the result of the script trying to combine the 69360 vertices in figure 5.2.

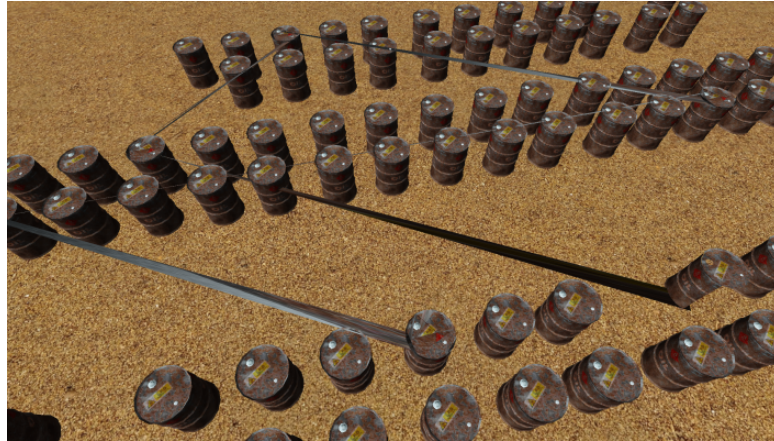


Figure 5.2: The result of using *CombineMeshes()* method with more than 65000 vertices

This is where the Draw Call Minimizer comes in handy. It checks if the total amount of vertices exceeds 60000 and if it does, it creates multiple combined meshes as the result. Therefore, it does not necessarily create only one combined mesh in the end but unlike the Simple Mesh Combine script it can deal with even a larger amount of vertices and therefore larger amount of meshes to combine. This is one of the reasons why we considered the Draw Call Minimizer to be used in this thesis.

### 5.1.3 Multiple textures

Dealing with more than one texture is another issue that the selected scripts solve in a different way.

As mentioned before, the Simple Mesh Combine script uses the standard *CombineMeshes()* method. This method does not deal with multiple textures. What this means is that it takes all the meshes that are about to be combined and applies the first texture found to all of them. Therefore, by using this standard method it is impossible to combine the meshes correctly using different textures. We can see the results of trying to use the Simple Mesh Combine script for combining meshes with various textures in figure 5.3.

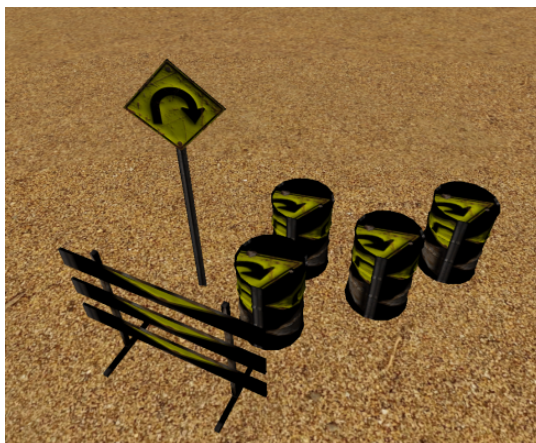


Figure 5.3: Trying to combine meshes with different textures using the Simple Mesh Combine script

An approach that we can use to combine meshes with different textures is the use of a texture atlas. A texture atlas is a large texture that multiple smaller textures used by various objects. When we use this technique, we need to apply the same material on all the objects covered by this texture atlas. They distinguish between the single textures within the atlas by using UV coordinates, which map to different coordinates within the texture atlas. This way multiple materials can use the parts of the same texture atlas without the need to have a separate texture.

The Draw Call Minimizer uses this approach. This script deals with everything in its own way, which is true also for dealing with textures. It combines all the textures of all the meshes into one texture atlas. This way it is possible to use this script even for combining completely different meshes. We can see the results of such a combination in figure 5.4. However, this way of combining textures has some drawbacks as well. One thing to consider when opting for combining textures with this script is that it creates only one new material. This material will take all the different properties of the materials used in the original meshes. Nevertheless, it will use only one shader, the shader of the first mesh from the original set of meshes. This way if we want to combine meshes that use materials with different shaders we downgrade the variety to only a single material.



Figure 5.4: Combining meshes with different textures using the Draw Call Minimizer script



There is also another thing to take into account when trying to combine textures by the Draw Call Minimizer. The textures have limited maximum sizes. While the desktop platforms and their OpenGL and Direct3D versions can handle textures with up to 16384 x 16384 pixels, the mobile devices are quite limited in this area. The Windows Phone devices are limited by their version of the Direct3D API to 4096. Some devices running Android are even limited to only withstand textures with the dimensions of 1024 x 1024. That is why it is not the best idea to just combine all the textures into one large texture atlas. They need to be either rescaled to fit the desired platform or just left in their original form. However, the Draw Call Minimizer does not deal with this problem. If it created the textures within the editor and we would be able to shrink them and work with them, this would not be such a big issue. However, as it works in the runtime of the game we have no way of directly influencing the size of the final texture. Therefore, this feature manages to crash the application when trying to combine more objects with different textures on mobile devices. This behavior is of course undesirable and is thus a major drawback of this script when considering it for multiplatform development.

#### 5.1.4 Other components of game objects

Another distinction between the two scripts is in how they deal with various components of the meshes. We have already discussed their way of manipulating with textures of the meshes.

As an example of such a component, let us look at colliders. While we have stated that the combining of meshes is often used for props and objects that are in the game just for aesthetics, they can also have some functions. Colliding with these objects may be beneficial for the gameplay as they can serve to block some areas or as a hiding place from enemy's bullets. Therefore, we would like to be able to use colliders with the final mesh created by combining smaller ones.

After combining the original meshes, the Draw Call Minimizer destroys the original ones. This way we lose all the additional properties and components that they may contain. In order to gain a collider with the newly created mesh we can modify the script and create a MeshCollider object with the new mesh as its property. However, a MeshCollider is less efficient than any of the primitive colliders such as a sphere, a capsule or box. So for example if we had a few objects with sphere colliders, which fit our requirements for collision detection with the objects' geometry, we would suddenly end up with a mesh collider encapsulating all the objects and being much slower than the original approach. On top of this, we would also lose all the other components like scripts or audio sources that come with the original objects.

The Simple Mesh Combine script on the other hand does not destroy the original objects after combining their meshes. It just disables the mesh renderers used for drawing the original meshes and leaves the meshes as they are. Therefore, it does not need to deal with other components of the original game objects. These components stay where they were before the combining of meshes started. All the original colliders, audio sources, scripts and others will work just as expected, while saving draw calls by disabling the mesh renderers and drawing just the newly created combined mesh.

## 5.2 Custom script

We have gone through the approaches of two very different scripts, which illustrate different ways of dealing with mesh combining. After the analysis of their major features we figured out that we would like to take the best aspects of both scripts. Therefore, we created a custom script that takes advantage of the positive features within the issues mentioned in previous sections. It also tries to avoid as many of the negatives as possible.

We call the new script Mesh Combinator and it can be found in the demo on the DVD attached in this thesis. Let us now look at what approaches we chose for the script and what it can actually do.

The first thing that we talked about was when the scripts execute. One of the scripts chose to do the entire job within the editor while the other ran after launching the game. We wanted to take advantage of both of these approaches. That is why Mesh Combinator supports both scenarios. We can set it to run automatically once the game starts and combine the meshes in the runtime. This is a good way to use while developing a game, as it does not bother the developer with extra steps and just takes a bit of time at the very beginning of the runtime. However, we also decided to support the other approach. This way once the group of objects, which we mean to combine, is finalized, we can just hit the Combine button and the script will combine the meshes right within the editor. When we run it afterwards it does not attempt to combine the meshes anymore and thus does not slow the loading down.

The next aspect to compare is the performance of our new script. We did the same tests with the Mesh Combinator as we did with Draw Call Minimizer and Simple Mesh Combine script. We can see the results in table 5.2.

Meshes	Vertices	Time [ms]
2	1156	5
10	5780	5
20	11560	6
30	17340	9
40	23120	11
50	28900	13
60	34680	15
70	40460	17
80	46240	18
90	52020	19
100	57800	21
110	63580	22
120	69360	24

Table 5.2: Performance of Mesh Combinator



To compare all three of the scripts we can look at graph 5.5.

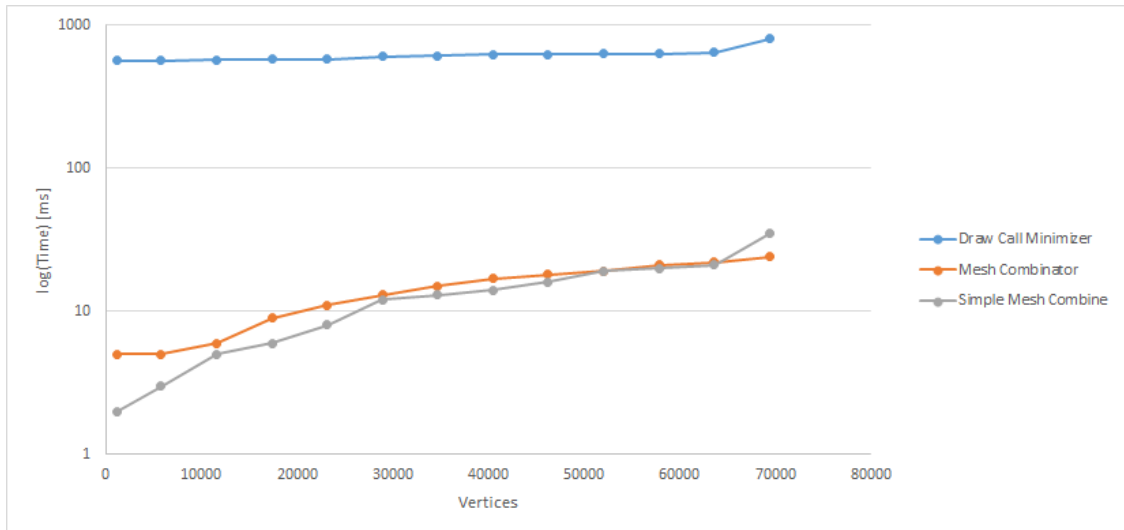


Figure 5.5: Graph comparing mesh combining scripts

We can see that Mesh Combinator is a lot faster than the Draw Call Minimizer. However, it is slower than the Simple Mesh Combine script, which comes as a winner out of this comparison. To achieve this kind of speed we used the standard `CombineMeshes()` method to combine our meshes instead of the custom approach used in the Draw Call Minimizer.

How does our custom script deal with a large amount of vertices? Very similar to the Draw Call Minimizer. It takes just as many vertices as fit within the 65000 vertex limit per mesh. Therefore, if we get to combine meshes with a total vertex count exceeding this limit we create more than one combined mesh at the end. We cannot extend the maximum amount of vertices, as this is a limitation of the GPU. We could reduce the polygon count in the final mesh, but intend to keep the quality of meshes that the developers chose to put in their scene and not mess up their creations. That way we chose the approach that combines as many meshes into one combined mesh. Only if it cannot fit into the limit, it creates another one.

In terms of dealing with multiple textures, we chose a different approach than the analyzed scripts. As mentioned before we tend not to lose any information, which is in the original meshes. If developers want to use texture atlases there are various solutions that allow them to create atlases outside of Unity and use them for meshes. We do not go against this approach and fully support it. However, we do not want to create artificial texture atlases and get to the point of crashing the game on mobile devices as the Draw Call Minimizer does.

The approach that we chose for Mesh Combinator is that we group the meshes based on their materials. If all the meshes share a single material they end up in one group and are combined into large meshes just by the sheer number of vertices that they are made of. However if we use meshes with different materials we first divide them into groups by their material. Then we go through the separate groups and apply the same approach of combining the meshes together by 65000 vertices. This way we can apply the Mesh Combinator to groups that contain various objects with different materials and textures. While it does not combine all the different objects together, it still combines as many as it can. This is also true for different models. If more models share the same material and

are in the group with the Mesh Combinator attached, the script combines them together. We find this approach as a nice compromise between combining everything and combining strictly just objects with the same material. When we use the script on a group of objects with one material, we get as good a result as with the Draw Call Minimizer. If used on a group of differently textured objects we receive more meshes in the end, but we keep all the materials intact, still using the shaders that the developer intended for them.

The last difference that we analyzed in the original scripts was their approach to other components within the combined meshes. For our script, we chose the approach of Simple Mesh Combine script. Therefore, after combining the meshes we disable all the mesh renderers of the combined meshes. We also allow the developer to undo the combining, when they launch the script from within the editor and they want to change something they can press the Undo Combine button and the combined mesh is destroyed, while the renderers of the original meshes are allowed again.

Now we have a custom script and that can run all the different platforms without problems with texture combining and issues with too many vertices.

## Chapter 6

# Baking lights

Unity Pro supports real-time shadows. The free version of Unity currently has support for shadows from only one directional light per scene and no other kinds of lights.

On top of this Unity does not support for shadows on Windows Phone. A driver issue of the Adreno 225 GPU unit causes this problem. As this GPU is present in most of today's Windows Phone 8 devices the Unity team disabled shadows for this platform altogether. They however promised that they would re-enable it once Qualcomm (the manufacturer of the GPU) fixes the driver.

Qualcomm however is not the only company with such issues. The Tegra line of processors from nVidia face a similar problem. The GPUs on Tegra 2 and Tegra 3 do not contain a depth buffer. As Unity uses the depth buffer to draw the real-time shadows, these do not work on the devices with Tegra processors, no matter which platform they run. This is the main reason why we chose the Nexus 4 as a testing device besides the Nexus 7. When we tried Nexus 7, we could not get any shadows computed on the runtime and therefore there would be no testing available for Android. With Windows Phone, the testing of shadows is not possible, as the whole platform is cut off from the real-time shadows. However we wanted to test our examples on Android, as this is possible. Therefore we reached out for another device, this time with a processor by Qualcomm, which can handle drawing shadows on the go.

Luckily, there are also other ways to provide shadows in a scene. In order to get static shadows we can use light baking. This technique can precompute the lighting attributes of the surface and store them in a texture. It saves the processing necessary in the runtime, as we do not have to compute lighting attributes once the lighting is baked. This is another way to speed up the rendering process.

Unity calls this feature lightmapping and offers its basic version for free. It integrates the Beast lightmapper owned by Autodesk (formerly a product of Illuminate Labs), which is a set of powerful tools for creating realistic lighting in games<sup>[3]</sup>.

Only the static objects can have their shadows baked in the scene. So in order to use an object for light baking we need to declare it „Ligtmap Static“. We can set this option in the top-right corner of the Inspector window. Then we need to turn on the shadows for the lights we want to use for light baking. Finally, we open the Lightmapping window. There are various settings including quality and resolution in the Lightmapping window. For our example, we set the mode to „Single lightmaps“ in the bake tab, set ambient occlusion to 0.4 and the resolution to 16 texels per world unit. Then we bake the lights into the scene. This way we get the shadows baked into the scene and we can use them on all the platforms in the same way. We can see the result running on a Windows Phone device in figure 6.1.



Figure 6.1: Baked lights on a Window Phone device

Unity Pro provides even more functionality. It allows us to bake Global Illumination of the scene as well as the Sky Light. These features are not available in the free version.

## 6.1 Performance testing

After describing the light baking functionality in Unity, we can look at how long it takes to bake a scene and how the settings influence baking of lights.

To test the performance we use the model of barrel 5.1 with 578 vertices, which we used for testing of mesh combining scripts. When we started with baking lights using this particular model, we encountered an issue. The shadow casted from the model onto the underlying terrain was alright. However, a problem appeared on the barrel itself. In figure 6.2 we can see the outcome of the original try.

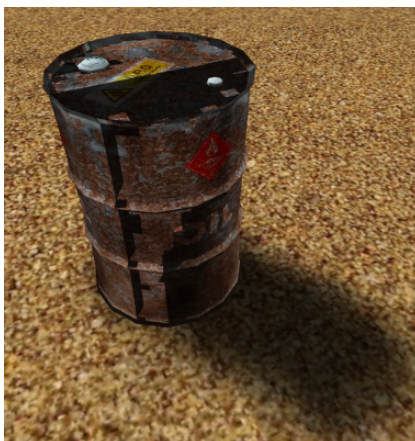


Figure 6.2: Issue encountered when baking lights

The barrel got a large dark shadow casted on its top and partly on its side. Unity did not provide us with any error or warning message so we did not know what was the reason for this unexpected shadow. After doing some research on the issues connected to light baking in Unity, we figured out that the problem is with lightmap UVs of our model. Luckily, Unity can unwrap your mesh for us to generate lightmap UVs [9]. This can be done

when we go to the import settings of the model and select „Generate Lightmap UVs“. Afterwards, when we used the UVs generated by Unity, the model issue disappeared.

Another obstacle that we met was the size of the lightmaps. The default maximum resolution of textures is 1024, as we examined in section 5.1.3 when we dealt with textures in mesh combining. This is also true for lightmaps. However, the terrain, which we use as the floor of our scene is set to the resolution of 2000 x 2000 pixels by default. We can take a look at how the 1024 x 1024 lightmap maps onto this size of the terrain in figure 6.3.



Figure 6.3: Lightmap on terrain with 2000 x 2000 pixels

We can set Unity to work with larger textures and lightmaps. Nevertheless, as we described, there may occur problems when targeting mobile platforms and using larger textures. Therefore we did not want to chose this approach. However, if we tried to bake our lights with the 1024 x 1024 lightmap textures, the barrel did not cover even a single texel to cast a shadow. This is why we went for the other option - shrinking the terrain.

When we switched the terrain into 200 x 200 units, we got the result from figure 6.4.

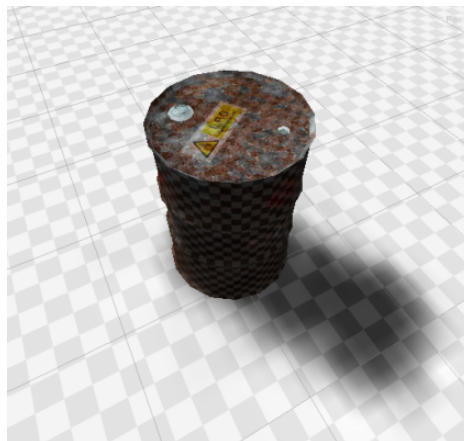


Figure 6.4: Lightmap on terrain with 200 x 200 pixels

This is much more usable. But once we solved a problem by shrinking the underlying terrain, we cannot forget that not all the games are played on small maps. If a game needs more space we advise to either use multiple smaller terrains to get proper lightmaps or use custom game objects as the floor for our level.

However, after we discussed the issues that we faced when trying to test the light baking, we can finally take a look at the results that we achieved in table 10.2.

Meshes	Vertices	Resolution	Time [mm:ss]
10	5780	10	1:04
10	5780	20	1:08
10	5780	50	1:10
10	5780	100	1:24
10	5780	200	2:09
10	5780	400	4:40
10	5780	600	4:49
10	5780	800	5:10
10	5780	1000	7:23
100	57800	10	1:13
100	57800	20	1:22
100	57800	50	1:46
100	57800	100	3:04
100	57800	200	9:14
100	57800	400	20:14
100	57800	600	27:02
100	57800	800	43:11
100	57800	1000	51:35

Table 6.1: Baking lights with different settings

When we look at the table, we can tell that the times for 10 meshes are quite reasonable. Even baking the shadows at the resolution of 1000 texels, it raised only by about a multitude of 7. However when we add more vertices to the scene we can see that the influence of resolution on the time of baking is also significant. While keeping the resolution down to about 50, the time was not changing that drastically. However it started rising up after 100 and went all the way to over 50 minutes when the resolution reached 1000 texels.

There is also the issue with Beast consuming all the performance it can get. When launched, the CPU usage raises all the way to 100% and stays there until the bake is over. Therefore it is not advisable to launch high resolution light baking on large scenes when we plan to do some other work at the same time. That is also why we would like to suggest using lower resolution bakes while in development in order to see the general layout of the shadows in the scene. Only when the scene is ready, we can perform a full out light baking and wait for the outcome for a few hours.

## Chapter 7

# Custom shaders

Unity has a set of built-in shaders. It also allows developers to use their own shaders. We can use Shaderlab to create custom shaders. It allows us to write both vertex and fragment shaders in Unity.

We would like to look at the differences between platforms in terms of shader programming. There are several rendering platforms available in Unity. For Windows Unity defaults to Direct3D, for Mac OS it uses OpenGL and so on. We can specify the platforms that apply to our shader. For example if we are creating a shader that requires some functionality of Direct3D 11 we do not want to execute this shader on a Mac so we can then specify that the shader will only be used on Windows by using this pragma:

```
#pragma only_renderers d3d11.
```

Android and iOS support the OpenGL ES rendering. Unity supports OpenGL ES in version 3.0, which is fully compatible with OpenGL 4.3. This way we can take advantage of a recent version of this rendering interface as it came out in 2012. OpenGL ES 3.0 is the current version that is used on the mobile devices.

On the other hand, devices running either on Windows Phone or Windows RT platforms support Direct3D rendering. They use version 11 of Direct3D with feature set 9\_3. The feature level includes a set of tools within the specified Direct3D version. It is aimed at a specific hardware. Developers can this way use the newest version of Direct3D with a limited functionality. However, a shader written in an older feature set can use the performance improvements of the newer runtime on hardware that supports the same version of Direct3D. Nevertheless, we need to say that this feature set is limited compared to the current version 11\_1 which is available for Windows on desktop.

We can see the comparison of these feature sets in table [7.1](#).

Feature	Feature level 11_1	Feature level 9_3
Shader Model	5.0	2.0
Geometry Shader	Yes	No
Stream Out	Yes	No
DirectCompute/ ComputerShader	Yes	N/A
Hull and Domain Shaders	Yes	No
Texture Resource Arrays	Yes	No
Cubemap Resource Arrays	Yes	No
BC4/BC5 Compression	Yes	No
BC6H/BC7 Compression	Yes	No
Alpha-to-coverage	Yes	No
10-bit XR High Color Format	Yes	N/A
Logic Operations(Output Merger)	Yes	No
Target-independent rasterization	Yes	No
Multiple render Target(MRT) with Forced-SampleCount 1	Yes	No
UAV slots	64	N/A
Max Texture Dimension	16384	4096
Max Cubemap Dimension	16384	4096
Max Volume Extent	2048	256
Max Texture Repeat	16384	8192
Max Primitive Count	$2^{32} - 1$	1048575
Max Input Slots	32	16
Simultaneous Render Targets	8	4

Table 7.1: Direct3D Feature level comparison [2]

Let us look at one example of the feature level 9\_3 limitations:

Semantics are required on all variables passed between shader stages. The example below creates an error:[6]

```
struct vertOut{
float4 pos: SV_POSITION;
float4 scrPos; //ERROR! no semantic
};
```

The fix is easy:

```
struct vertOut{
float4 pos: SV_POSITION;
float4 scrPos: TEXCOORD0; // <-- FIX! add semantic.
};
\label{verb:ShaderError}
```

So when developing for more platforms we need to consider this. If we want to target devices running either Windows Phone or Windows RT platform we need to resource for the feature set supported by them.



## Chapter 8

# User interface

The graphical user interface (GUI) can be a lot trickier to program for multiple platforms than the 3D environment. There are many different screen resolutions to cover.

### 8.1 GUI scripting

The first way to create a GUI in Unity is by using a script to draw the controls. We do this in the *OnGUI()* function. The controls are positioned on absolute position on the screen. This could be a problem if we want to target different screens with different resolutions. To make these controls respond to the screen resolution we can use the `Screen.width` and `Screen.height` properties to determine the resolution. For example, the code below draws a button in the top right corner of the screen no matter what the screen resolution is.

```
void OnGUI()  
{  
GUI.Button(new Rect(Screen.width - 150, 0, 150, 50), "Top right button");  
}
```

When we look at the mobile devices, we can also see that they work in different orientations. Specifically we have the landscape orientation, which is similar to the PC screen - the horizontal resolution is higher than the vertical one. However, we also have a portrait orientation. The games access this area in different ways. For most of the games, it does not make much sense to run in both landscape and portrait. The user interfaces are then optimized for just one layout. Still if a game decides to go for the portrait mode on a mobile device it cannot benefit from it on a PC screen.

That is why we need to be able to react to screen orientation changes. We can do this by using the `Screen.orientation` property in Unity. We can do this by saving the previous orientation and checking if the orientation changed in the *OnGUI()* function. If it changes, we can redraw the GUI according to the new orientation.

We can use GUI scripting for menus and GUI heavy scenarios. It is however not very efficient in terms of performance. Therefore, we do not advise the use of GUI scripting for in-game menus.

### 8.2 GameObjects

Another way to display user interface is to use GameObjects created for this purpose. For creating on-screen GUI, Unity contains `GUIText` and `GUITexture` GameObjects.

GUILayout is a simple text field that allows us to display simple text data. We can modify its basic properties like color or font.

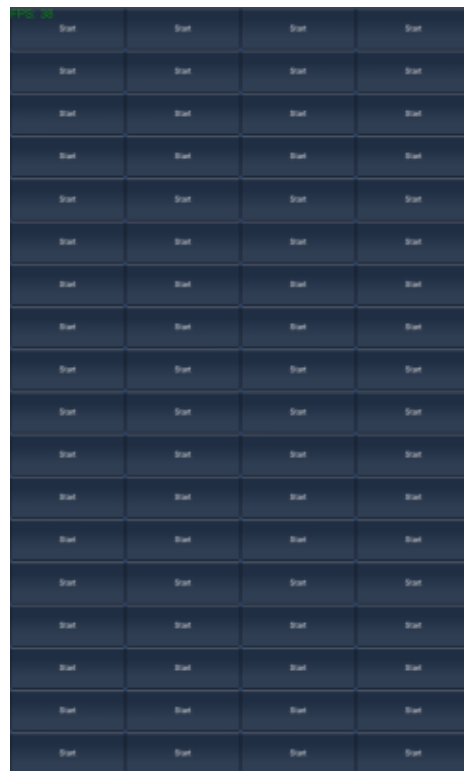
We can use GUILayout to create more controls. It displays a texture on the screen. We can also assign it a script that reacts on various events, for example entering or leaving of mouse over the object. This way we can manipulate the object using different textures.

Both GUILayout and GUILayout are positioned relatively onto to screen. So we do not need to care about the specific resolution and they will be placed where we intended even if the screen resolution or orientation changes.

Another advantage of using the GameObjects is their performance. We carried out a simple test, placing 72 buttons on the screen using both methods as we can see in the figures 8.1a and 8.1b.



(a) GUILayout



(b) GUI script

Figure 8.1: GUI buttons on a Windows Phone device

### 8.3 Performance testing

When we went through the basic differences between the two approaches to creating user interface we tested them out on various platforms. First we took a scene filled with buttons, 4 columns by 18 rows and tried to compare the performance of game objects to the GUI scripts. We can see the results in table 8.1.

Platform	Game objects [FPS]	GUI scripting [FPS]
Windows	580	294
Android (N4)	60	45
Android (N7)	60	58
Windows Phone	60	40
Windows RT	60	30

Table 8.1: Comparison of GUI scripting and game objects in testing scene with 72 buttons

It is worth noting that the FPS caps at 60 for all the mobile platforms for Unity. Therefore, we will consider 60 as the top rate. It stayed over 60 on desktop for both approaches. However when we tried to run it on any of the mobile devices it was very different. Game objects kept the 60 frames rate, but when the buttons were drawn using a script the rate dropped lower on every single platform.

This is why we decided to investigate more on the GUI scripting. We tried different amounts of buttons on the mobile platforms. The results that we achieved are in table 8.2.

Buttons	Android (N4) [FPS]	Android (N7) [FPS]	Windows Phone [FPS]	Windows RT [FPS]
10	60	60	60	60
20	60	60	60	60
30	60	60	60	60
40	60	60	60	60
50	56	60	49	45
60	52	60	44	38
70	47	60	42	33
80	43	57	36	28
90	41	52	33	24
100	38	48	29	19

Table 8.2: Testing different amounts of buttons using GUI scripting

In contrast to the mobile platforms we also tried to increase the amount of buttons on Windows. However we found out that it can outperform all the mobile platforms in terms of drawing buttons using GUI scripting. At 500 buttons it still ran at 60 FPS, while lowering the frame rate to 51 at 600 buttons. It is even more interesting when we realize

that the Windows and Windows RT versions were tested on the same hardware. We can see that the optimization of Unity on Windows is much better than on Windows RT.

However, we can see that both of the approaches to creating the user interface differ. We can create simple user interface elements with game objects and use the scripting for some more advanced controls. Therefore they are usually used for different purposes. The usual way to use the scripts is to create the main menu, pause menu and other menus that are visible while the actual game is not running. On the other hand we use the `GUITexture` and `GUIText` for creating an in-game menu. This way we get tools for easily creating the complex menus when the game is not running and we use the less performance sensitive game objects when we need to save as much computing power for the game as we can.

## Chapter 9

# Tutorial

This thesis describes some ways of dealing with multiplatform development. However it keeps to be a formal thesis and not really written for the general public. That is why a tutorial was introduced to share these discoveries with all the developers that might find them useful. The purpose of the tutorial is similar to this thesis. It is supposed to inform people about how to develop for multiple platforms at once. But on top of it it tries to do so in an easy-to-understand way.

The tutorial can be found at <http://unitytips.wordpress.com> and is in a form of a blog. This seems to be a fitting structure for people to be able to look through the various tips and find what they need. There are 3 kinds of articles posted on the blog:

- general
- quick tip
- deep dive

The general category serves for announcements or things that are not directly related to the topic of the tutorial. Currently you can only find a welcome note in this category but it is prepared for further needs if they arise.

The next category is the most usual one. Quick tips are what offers people an easy insight into some challenges and their solutions when they deal with multiplatform development. These are filled with images and step by step tutorials on how to work out a solution to the topic in question. The image tutorials are a nice addition to the formal nature of this thesis. As it was not desirable to fill the thesis with too many images this approach was chosen.

The deep dive category is here for the people who want to learn more about the solutions provided in quick tips. If an interesting discovery was made during the time when an issue was studied it is described here. So most users that just want to get something working can go through the quick tips. But those who want to find out why the issue was there in the first place or why this kind of solution worked can go for a deep dive and learn more about it.

All the mayor issues from this thesis are or will be described within the tutorials. You can also find examples of the currently published tutorials in attachment **B**. The tutorials are an easy way to build up on this thesis even after it is published. There was also a need for a simple solution to share the new discoveries with people rather than having to publish everything within a paper.

# Chapter 10

## Demo scene

As a part of this thesis we created a demo scene in Unity. Its purpose is to demonstrate the techniques described in the previous chapters and test them in a real-world scenario.

We call the scene Desert Mission as it takes place in a desert city. As the author of this thesis is not an artist and cannot create representative models we decided to use some assets from the Unity Asset Store to make it look good. The list of the used assets follows:

- Arabic City<sup>1</sup>
- Industrial Barrel Pack<sup>2</sup>
- Palm Trees Pack<sup>3</sup>
- Angry Bots<sup>4</sup>

To start with the description of our scene we will take a look at its complete layout in figure 10.1.



Figure 10.1: Demo scene

---

<sup>1</sup><https://www.assetstore.unity3d.com/en/#!/content/10410>

<sup>2</sup><https://www.assetstore.unity3d.com/en/#!/content/12452>

<sup>3</sup><https://www.assetstore.unity3d.com/en/#!/content/13540>

<sup>4</sup><https://www.assetstore.unity3d.com/en/#!/content/12175>

As we can see, the Arabic City can be seen pretty much everywhere in the scene. All the buildings, walls and most of the props come from this pack. The Industrial Barrel Pack is used to demonstrate the work with different textures on the same models. The barrels from this pack serve as props in the scene, fit the arabic environment and blend in with the other props pretty well. Palm Trees Pack contains different models of palm trees, which however use the same materials. Therefore, we can demonstrate that the Mesh Combinator script can combine these meshes together even though they have different geometry. These three packs are used to create the environment for our game.

The last pack used is the Angry Bots demo provided by Unity. We do not use the whole pack as it is unnecessary for our purposes. We use this pack for creating a gameplay within our scene. We decided to use the prefabs for player and two enemies - a mechanical spider and a mech walking on two legs. These are all already configured for the use within a game so we did not need to do much to modify their behavior. Whenever the player comes within a certain radius from a spider it starts its auto destructive sequence and starts to chase the player. Therefore, this is a kamikaze unit that does a lot of damage on impact, but is quite easily destroyed. On the other hand the walking mech is a tougher enemy, which shoots homing missiles at the player whenever he comes within its range. It has also a higher amount of health points so it is more difficult to kill the unit. The player is equipped with an automatic sub-machine gun and can thus shoot the enemies from a distance, rather than approaching them for a close combat.

After describing the assets used in the scene, let us take a look at the gameplay designed for the demo. The player starts in the middle of the map and there are no enemies to begin with. The enemies get spawned regularly every few 10 seconds at 6 predefined spots. When we spawn a new enemy a red dot flashes on the minimap located in the upper-right corner of the screen at the spot where the enemy was spawned. The player can then go to the specified location and try to kill the enemy. If he or she succeeds he gets 2 points for killing a spider and 10 points for killing a walking mech. The current score can be seen in the upper-left part of the screen. If the player is hit either by the spider's explosion or the mech's rocket he or she loses some hit points. These are displayed as a series of hearts in the upper-left corner of the screen. The game ends when all the hit points are depleted. Then a game over screen appears with the score achieved in this game play. The user can then choose to start a new game or to exit the game.

Controls of the game are kept from the Angry Bots asset. When we play the game is on a desktop platform, it uses either WASD or arrow keys to move the player around and mouse to rotate and shoot. On mobile devices the controls are provided in the form of two virtual joysticks placed in the bottom part of the screen. The left one serves for moving the player and the right one to shoot the weapon in a desired direction.

## 10.1 Mesh combining tests

We have described the scene, its components and the gameplay intended for demonstration. Now we can finally look at the different aspects that were worked within the scene to demonstrate the outcomes of various optimization techniques mentioned in this thesis. We will start with the use of our Mesh Combinator script.

When designing the scene, we took the mesh combining into account as it is a large part of this thesis. Therefore, there are a lot of groups of props spread all over the scene to see how our script performs in an actual scene rather than some testing scenarios. For a quick repetition of what the script actually combines together we can say that it goes through

the objects within a group to which it is attached. It divides the meshes based on their materials and then combines the meshes with the same materials together. Therefore, it handles groups containing objects with one material very well, while still giving us a certain advantage with groups using multiple materials.

We would also like to note that the shadows were turned off during testing the mesh combining. Moreover, the objects were combined within the other groups, which do not belong to the group that we are currently testing. Both of these steps were taken in order to reduce the unnecessary draw calls that do not directly influence our testing cases.

To examine the combining on a group consisting solely of one type of meshes we used the part of the scene which can be seen in figure 10.2. When the script was not used, this view took 55 draw calls. If we used Mesh Combinator the draw call count lowered to 30. Therefore, by running our script, we saved 25 draw calls when combining a group of 26 barrels and sped up our game. We can see the combined mesh created by our script in figure 10.3. When measured, the combining took 9 milliseconds if run at the beginning of the game.



Figure 10.2: Group of the same objects in demo scene



Figure 10.3: Combined mesh created using the group from figure 10.2



The next group of objects is a group of palm trees and is a specific one. It is made of different models. However, the models all share the same materials. All the models are made of two different meshes each of them using a different material. Thus, we have two different materials within the group and the script produces at least two meshes at the end. If there are two many vertices it may produce even a larger amount, but this is true for any group. We can see the group of palm trees in figure 10.4. The draw call count in figure 10.4 is 27 and this is the number after combining the meshes. Before using the Mesh Combinator, we would see 51 draw calls in this scene. Again, we managed to save 24 draw calls simply by running our script. We can see the 2 large meshes, created by combining the group, in figures 10.5a and 10.5b. For this group of meshes it takes 14 milliseconds to finish the combining when run within the game's runtime.



Figure 10.4: Group of different objects with the same textures in demo scene

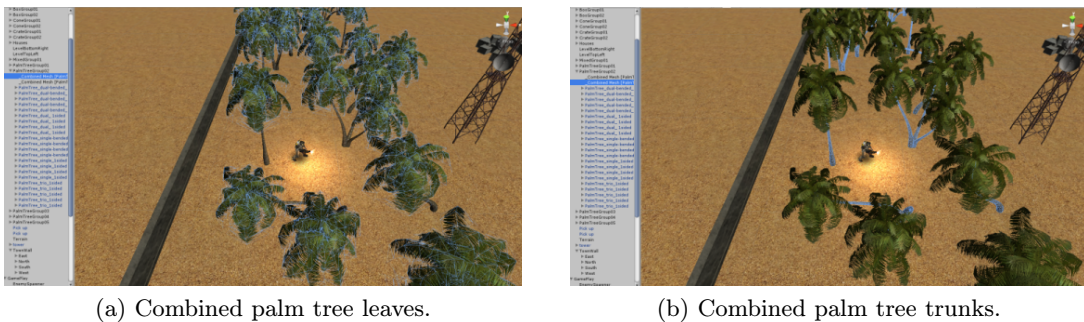


Figure 10.5: Combined meshes created using the group from figure 10.4

The last group to examine is a mixed group of objects. We can see it in figure 10.6. It is comprised of various objects with various materials, geometry and vertex counts. This illustrates the most usual way of using props within a game. There are usually different props placed close to each other to make the scene look realistic. When we take a look at the performance of our script, the draw call count went from 81 down to 35 after combining the meshes. It did not go all the way to one mesh as the Optimized Mesh Combiner would,

but on the other hand it kept all the materials and other components of the meshes intact, while still managing to reduce the draw calls significantly. We can see the various meshes created by combining this mixed group of objects in figures 10.7a, 10.7b, 10.7c and 10.7d. The time it takes to use the Mesh Combinator on this group of objects is 11 milliseconds.



Figure 10.6: Group of different objects in demo scene



(a) Combined crates.



(b) Combined barrels.



(c) Combined boxes.



(d) Combined tires.

Figure 10.7: Combined meshes created using the group from figure 10.6

We have shown our Mesh Combinator script used in three different scenarios. The results show that it performs the best when used for groups of objects that share the same material. However, it still manages to save draw calls even for mixed groups that contain

at least two objects, which share the same material. Therefore, we can declare that the script successfully optimizes our game, making it playable even on mobile devices.

As the last part of this section we would like to take a look at the performance of our script on different platforms. The meshes used for combining have a total count of 200345 vertices and belong to 21 different groups. We think this can provide a real-life example of the use of our script. The results of testing its speed when executed at the runtime of our game are in table 10.1.

<b>Platform</b>	<b>Time [ms]</b>
Windows	92
Android (N4)	380
Android (N7)	516
Windows Phone	473
Windows RT	1112

Table 10.1: Speed of the Mesh Combinator script on various platforms

Before analyzing the numbers we need to note that we encountered a strange behavior of the script on Android devices. The first group of objects, which got combined (no matter which group it was) took much longer than the rest. We tried to run the script several times, but we always encountered this issue. To be more specific, it took approximately 189 milliseconds to combine the first group of objects on Nexus 4 and 213 milliseconds on Nexus 7, while the rest of the combines stayed within the area of 1 - 70 milliseconds. It might have been caused by some other work that the devices do while starting the game, but it was always present. The other platforms did not suffer from this and they behaved as expected. We again experienced the difference between optimization of Windows and Windows RT, when the same script run on the same hardware took 92 milliseconds on Windows and more than 12 times that when run within a Windows RT game. The rest of the devices basically follow their hardware specifications, with the Nexus 4 coming out on top with the most powerful processor, followed by Nokia Lumia 920 and Nexus 7.



## 10.2 Baking lights tests

The next issue to discuss has to do with shadows. After testing the light baking on some artificial scenes, we can finally see its performance within a real-world game scenario. The static objects in our game have a total of 313575 vertices. This is a larger scene compared to the testing ones so let us see how the Beast performs with baking lights within such a large scene.

Resolution	Time [hh:mm:ss]
10	00:04:43
20	00:07:49
50	00:28:57
100	01:03:12
200	01:58:01
400	04:10:53
500	05:12:00

Table 10.2: Baking lights with different settings

As we can see, the time raises pretty much linear. However, the higher resolution we use, the more we need to wait, so we would not recommend to use light baking with high resolution when testing the game. This should be left for the final stages of game development, as it is very time consuming.

To compare our scene when dynamic lights are used as opposed to light baking, we can look at the sets of figures 10.9 and 10.9

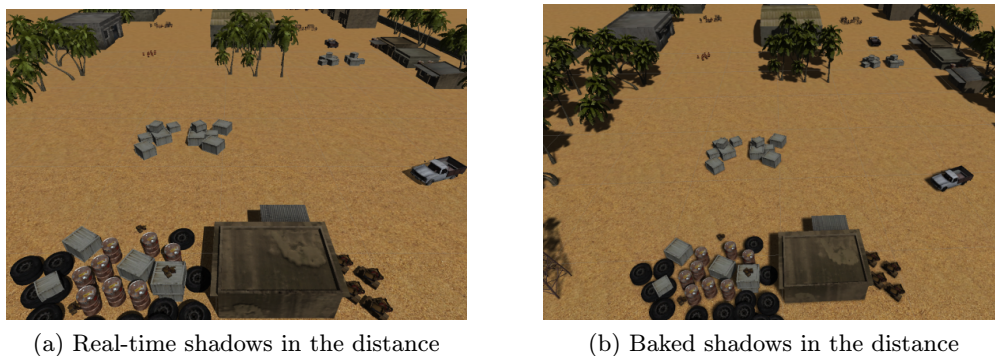


Figure 10.8: Comparison of shadow techniques in the distance

The first set of figures shows shadows which are far away from the camera. We can however tell that Unity does not display dynamic shadows this far away. This is an optimization technique so that it does not have to compute far away shadows in large scenes. Therefore we can only see a few of them, for example the building in the front or the car on the right side. Nevertheless, the baked shadows are casted onto the texture, so we can see the baked shadows no matter how far they are.

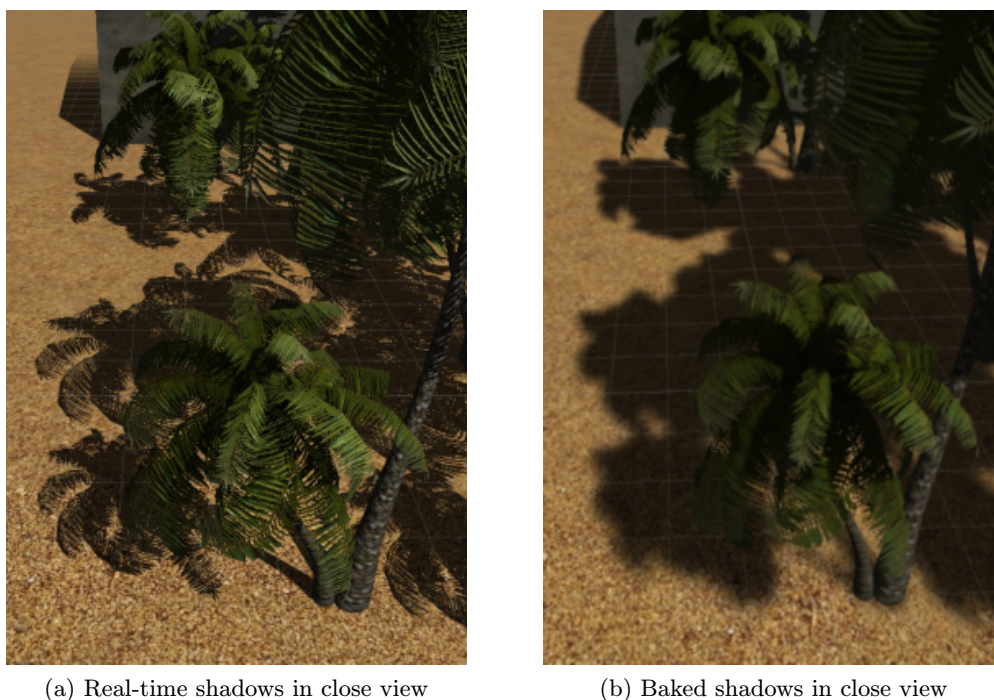


Figure 10.9: Comparison of shadow techniques in close view

When we however compare the shadows from up close. We can see that there are much more details visible in figure 10.9a, which uses the real-time shadows. Here we can see the details that we sacrifice by using the baking lights technique. However, the shadows in figure 10.9b still look good enough, especially if we focus on mobile development.

We can now take a look at the advantage of light baking. The performance improvement. We will test the different parts of the scene, which we tested in section 10.1. However this time we will test the frame rate that we achieve when using real-time shadows compared to using only light mapping. We cannot test this neither on the Windows Phone device, nor on Nexus 7, as these devices do not support real-time shadows. Therefore we will consider only Windows, Android on Nexus 4 and Windows RT for this series of tests. In this case we decided not to use mesh combining on any of the grouped objects, as we wanted to explore only the performance gained by baking the lights. We will explore the combination of both optimizations later on in section 10.4. We can see the results in table 10.3.

Part of the scene	Shadows	Windows [FPS]	Android (N4) [FPS]	Windows RT [FPS]
Barrels 10.2	real-time	85	42	25
Barrels 10.2	baked	203	60	44
Palm trees 10.4	real-time	52	30	20
Palm trees 10.4	baked	60	50	37
Mixed 10.6	real-time	54	30	17
Mixed 10.6	baked	60	52	35

Table 10.3: Influence of baking lights on testing scenes; the numbers in the first column refer to the corresponding figures from section 10.1

## 10.3 User interface tests

When we designed the demo, we wanted to show the different optimizations that we talked about in the previous chapters. One of them is optimizing the graphical user interface (GUI). As we noted, we do not need to optimize the GUI in the main menu, as when this menu is displayed, the game is not running. Therefore we will focus on the in-game menu. These are the labels, numbers and other parts that are displayed throughout the game and need to be optimized in order not to take away the performance needed for our game.

The in-game user interface in the demo application consists of three parts:

- HP counter
- Score counter
- Mini map

HP counter is here to display the hit points of the player. It is made of a number of heart symbols to represent the amount of hit points that our player has. It resides in the upper-left corner of the screen and we can see an example of it in figure 10.10.

The second control is used to count the score, as the name implies. Score counter is located below the HP counter, so it takes another part of the upper-left corner of the screen. It is a text field, we can again see it in figure 10.10.

Mini map is the last part of our testing GUI. It displays the layout of the level as well the spawning positions of enemies. The spawning positions are displayed as flashing red dots as can be seen in figure 10.10. The mini map is located in the upper-right corner of the screen.



Figure 10.10: In-game GUI from the demo scene

In chapter 8 we wrote about two ways of dealing with user interface. We can use either scripts and draw controls within the *OnGUI()* function or we can use game objects to display our information.

We created the three aforementioned controls using both approaches. All of the controls use some kind of a script to make them work. HP counter uses it to subtract health points when the player is hurt, score counter receives a message anytime an enemy dies to add the score and the mini map gets notified when an enemy gets spawned to flash the red dot at according position. These scripts are the same for both versions of the GUI. We used one more optimization technique when designing the function of the mini map. We decided to

use coroutines. A coroutine is a function that can suspend its execution until a yield action has completed[8].

The main difference between the two approaches is how they draw the controls. One of the methods of creating the GUI is based on `GUITexture` and `GUIText` game objects. It uses `GUIText` for the score counter and `GUITexture` for the rest of the controls. The other approach uses the controls drawn within the `OnGUI()` function by using `Labels` for HP counter and score counter and the `DrawTexture()` function for mini map. Now let us take a look at how these two approaches stack up to each other on different platforms, which we can see in table 10.4.

Platform	Game objects [FPS]	GUI scripting [FPS]
Windows	762	680
Android (N4)	60	60
Android (N7)	60	60
Windows Phone	60	60
Windows RT	60	60

Table 10.4: Comparison of GUI scripting and game objects in demo scene

All the platforms handled the GUI flawlessly, no matter whether we displayed it using a script or game objects. The GUI is however pretty simple, in fact it only consists of 12 or 13 objects (depending on whether the enemy spawn point is displayed or not). Therefore if we take another look at table 8.2, we can see that all the platforms performed well with this amount of user interface elements. Nevertheless, we would still advise to use game objects for in-game menus, as if the menus grow in complexity, we would not have to face performance issues. We were able to recreate the menu using both techniques and it still looks the same.

## 10.4 All optimizations combined

After going through the single optimization techniques we can finally take a look at how they affect our demo scene when they are all combined together. For testing we again use the three parts of our scene, which we got familiar with in sections 10.1 and 10.2. The results of the thorough testing can be seen in table 10.5.

Platform	Part of the scene	Optimizations	Draw calls	FPS
Windows	Barrels 10.2	none	214	88
Windows	Barrels 10.2	all	32	217
Windows	Palm trees 10.4	none	231	52
Windows	Palm trees 10.4	all	30	158
Windows	Mixed 10.6	none	320	54
Windows	Mixed 10.6	all	37	177
Android (N4)	Barrels 10.2	none	202	42
Android (N4)	Barrels 10.2	all	33	60

Android (N4)	Palm trees 10.4	none	230	30
Android (N4)	Palm trees 10.4	all	27	52
Android (N4)	Mixed 10.6	none	315	30
Android (N4)	Mixed 10.6	all	37	57
Android (N7)	Barrels 10.2	none	55	35
Android (N7)	Barrels 10.2	all	33	35
Android (N7)	Palm trees 10.4	none	51	20
Android (N7)	Palm trees 10.4	all	28	20
Android (N7)	Mixed 10.6	none	105	23
Android (N7)	Mixed 10.6	all	37	24
Windows Phone	Barrels 10.2	none	55	33
Windows Phone	Barrels 10.2	all	33	40
Windows Phone	Palm trees 10.4	none	50	25
Windows Phone	Palm trees 10.4	all	27	27
Windows Phone	Mixed 10.6	none	105	30
Windows Phone	Mixed 10.6	all	36	35
Windows RT	Barrels 10.2	none	214	25
Windows RT	Barrels 10.2	all	32	51
Windows RT	Palm trees 10.4	none	231	20
Windows RT	Palm trees 10.4	all	30	43
Windows RT	Mixed 10.6	none	320	17
Windows RT	Mixed 10.6	all	37	45

Table 10.5: Comparison of performance improvement with all optimization methods; the numbers in the second column refer to the corresponding figures from section 10.1

As we can see, the optimization methods worked on all of the platforms. We experienced different amounts of draw calls when not using any of the optimization techniques than what we saw in section 10.1. This was caused by the use of real-time shadows. We can tell that the shadows add a lot of draw counts. For example the part of the scene with mixed group of objects raised its draw call count from 81 to 320 on Windows. This is a major raise. We did not see such high numbers when either our Windows Phone device or the Nexus 7 were used. This is obviously caused by the lack of dynamic shadows on these platforms. Therefore they kept very similar numbers to the original tests. The slight differences between the platforms is mostly caused by their resolutions, as the devices with lower resolutions display a smaller part of the level at once and thus the part contains less objects.

In terms of the frame rate, we saw improvements on all the devices except for the Nexus 7. Tegra 3, its processing unit, did not take advantage of the less amount of draw calls. However, all the other devices on all the different platforms managed to get a better result in terms of FPS. We can therefore say that the techniques that we decided to use work to create better performing games on all the 4 platforms tested within this thesis. When combined they provide a real advantage to the developer when targeting multiple platforms.



# Chapter 11

## Conclusion

This thesis offers a look into working with a game engine while targeting multiple platforms.

First we went through different engines that are currently available on the market. We explored the engines in terms of the platforms that they support. We took a closer look at Unity, the game engine selected for the purpose of this thesis. The analysis why we chose this particular engine came at the end of the first part of the thesis.

The next part deals with the multiplatform development itself. It shows us different approaches to this topic and briefly analyzes them.

Then we looked closer at development with Unity. After exploring the platforms available for Unity we chose 4 of them for testing purposes for this thesis - Windows, Android, Windows Phone and Windows RT. Afterwards we explored the development options for these platforms. While doing so, we discovered that all the different platforms approach developing and debugging in various ways. Therefore we described all the processes needed for developers to use Unity on all the chosen platforms. The approach to this is important as anyone who wants to follow up with Unity development needs to use the processes described in this part.

In the next chapters we analyzed various techniques used for optimization as well as differences between the selected platforms.

We first took a look at how we can reduce the draw calls in scenes with a lot of repeating objects. We picked the method of combining meshes. After discovering that there are multiple implementations of mesh combining for Unity, which we found on the Internet, we went on to analyze two distinctive ways of completing this task. We listed all the pros and cons of these two approaches and said why we did not decide to use either of them. Then we created a custom script, which combines the pros of both of the analyzed scripts. In the end, we compared the performance of both the analyzed scripts and our custom script in some testing scenarios.

After combining meshes to reduce the draw call count we looked at shadows. We provided an analysis of the light baking method and its performance. This method was used later on to reduce the frames in our demonstration scene.

The third part of our analysis briefly dealt with graphic APIs on different platforms. We described the APIs used on the tested platforms and mainly the difference between the versions of Direct 3D used on the desktop Windows system and Windows Phone used on mobile devices.

The next part dealt with optimizing the user interface. We analyzed two different approaches of creating the graphical user interface, compared their performance and provided some advices on how they could be used effectively.

We also created a tutorial as part of this thesis. We described its form in the next chapter and provided some examples of tutorials in attachment **B**, which can be already found online.

The last part of the thesis dealt with our demo scene. We created a game which is designed to demonstrate the techniques described in the previous chapters. We used our demo scene to measure the solutions that we analyzed before in a real-world scenario. We provided different examples for each of the methods and tested the performance of our solutions. We could see that the techniques described within this thesis helped us to optimize our game on all the platforms that we used for testing.

Last but not least, we wrote a paper summarizing the contents of this thesis and submitted it to the EEICT competition [5]. This paper submitted to the competition got published as part of the Proceedings of the EEICT conference and placed first in the Graphics and Multimedia category within the competition.

The results of the tests show that we succeeded in optimizing games for running on different platforms. These findings can be used by Unity developers who focus on multiplatform development and struggle with this area. Moreover, the attended competition proved that this thesis has some scientific impact as well.

# Bibliography

- [1] Kantar worldpanel comtech smartphone os barometer 06 01 14.  
[http://uk.kantar.com/media/615033/kantar\\_worldpanel\\_comtech\\_smartphone\\_os\\_barometer\\_06\\_01\\_14.pdf](http://uk.kantar.com/media/615033/kantar_worldpanel_comtech_smartphone_os_barometer_06_01_14.pdf).
- [2] DirectX developers - directx in windows phone. [http://developer.nokia.com/Community/Wiki/DirectX\\_Developers\\_-\\_DirectX\\_in\\_Windows\\_Phone](http://developer.nokia.com/Community/Wiki/DirectX_Developers_-_DirectX_in_Windows_Phone), 2013.
- [3] Autodesk. Realistic lighting for realistic games. <http://gameware.autodesk.com/beast>, 2014.
- [4] Jason Gregory. *Game Engine Architecture*. A K Peters/CRC Press, 2009. ISBN 978-1-56881-413-1.
- [5] Roman Jašek. Multiplatform game development using the unity engine. In *Proceedings of the 20th Conference STUDENT EEICT 2014*. FEEC VUT v Brně, 2014.
- [6] Marker Metro team Jaime Rodriguez, Keith Patton. Porting tips for windows phone with unity. <http://aka.ms/unityWPTips>.
- [7] MarkerMetro team Jaime Rodriguez, Keith Patton. Getting started on windows store with unity. <http://aka.ms/unityWinStoreStart>.
- [8] Chico Queiroz Matt Smith. *Unity 4.x Cookbook*. Packt Publishing, 2013. ISBN 978-1-84969-042-3.
- [9] Unity Technologies. Lightmapping uvs.  
<https://docs.unity3d.com/Documentation/Manual/LightmappingUV.html>, 2010.

# Appendix A

## DVD contents

The DVD attached to this thesis contains the following:

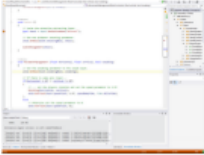
- Bin - folder containing the builds of demo scene for 4 platforms
- Thesis - folder containing the text of the thesis and the source files for building it
- Source - folder containing the source files of the demo scene
- Video - folder containing a short video presenting the main functionality implemented as a part of the thesis

## Appendix B

# Tutorial examples



Now we are all set to run our game on a Windows Phone device and debug it using Visual Studio. We just need to press the Run button and Visual Studio takes care of deploying our game to the phone and attaching the debugger. Then we can use breakpoints, watches, modify the variables on the run and all the neat features of debugging with Visual Studio.



As you can see debugging on Windows Phone needs some setting up but once we are done with the initial setup it is easy to debug games using Visual Studio. The logs on the other hand are quite difficult to use on Windows Phone. It's impossible to see logs while the game is running. They are accessible only via Windows Phone Power Tools, a program that needs to be installed separately. Once installed it can connect to a device and see all the developer apps and their isolated storages. There we can find our Unity log file and open it using any text editor straight from the phone. But it is locked while the app is running so we can only open the logs and see what was going on retrospectively.



Figure B.1: Tutorial - QUICK TIP - Debugging on Windows Phone

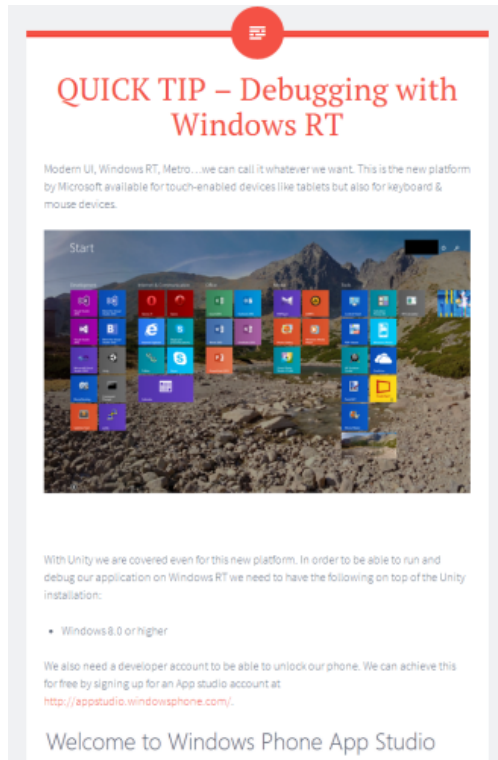


Figure B.2: Tutorial - QUICK TIP - Debugging on Windows RT

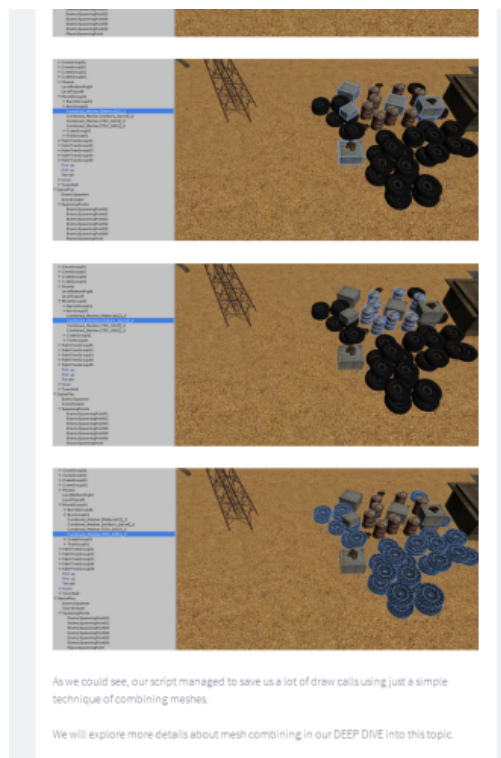


Figure B.3: Tutorial - QUICK TIP - Combining meshes

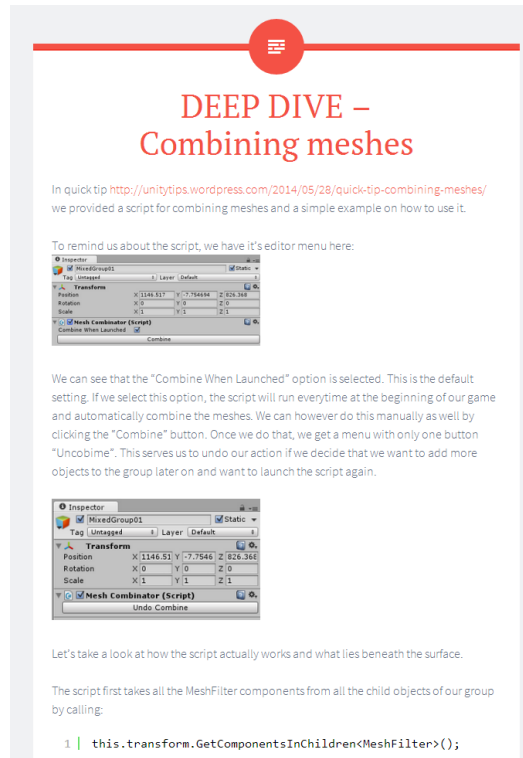


Figure B.4: Tutorial - DEEP DIVE - Combining meshes