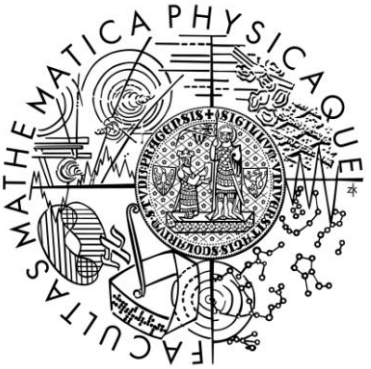
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Tuan Do Manh

Multi-platform Multiplayer RPG Game

Katedra softwaru a výuky informatiky

Vedoucí diplomové práce: Mgr. Jakub Gemrot

Studijní program: Informatika

Studijní obor: Softwarové systémy

Praha, 2015

Many thanks to Mgr. Jakub Gemrot and Bc. Otakar Nieder for their priceless advices and guidance.

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne ..... podpis

Název práce: Multi-platform Multiplayer RPG Game

Autor: Tuan Do Manh

Katedra / Ústav: Katedra softwaru a výuky informatiky

Vedoucí diplomové práce: Mgr. Jakub Gemrot, Katedra softwaru a výuky informatiky

Abstrakt: V rámci práce byla vytvořena multiplatformní hra, kterou je možné spustit na různých zařízeních se systémem Windows 8.1 a Windows Phone 8.1. Mělo se jednat o univerzální herního klienta spustitelném na různých zařízeních (PC, notebook, tablet, mobilní telefon). Hra měla spadat do žánru her na hrdiny (RPG) se zaměřením na akčně tahový boj. V rámci práce byl vytvořen vlastní 3D herní engine pro malé scény umožňující renderování objektů a animování postav. Vývoj herního enginu byl proveden pomocí rozhraní DirectX. V práci se podařilo napsat engine v .NET C# s použitím knihovny SharpDX. V rámci práce bylo také implementováno rozhraní pro cross-device komunikaci založené na bluetooth technologii pro komunikaci mezi klienty běžících na různých zařízeních.

Klíčová slova: multiplatformní aplikace, cross-device komunikace, 3D herní engine, DirectX, SharpDX

Title: Multi-platform Multiplayer RPG Game

Author: Tuan Do Manh

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jakub Gemrot, Department of Software and Computer Science Education

Abstract: A multi-platform game, which would be able to run on various devices with Windows 8.1 and Windows Phone 8.1 systems, was created in this work. It was supposed to be a universal game client executable on desktop PCs, notebooks, tablets or mobile phones. The game was supposed to be role-playing game (RPG) with focus on turn-based action combat. In this work, a 3D game engine was written which supports rendering simple scenes with objects and animated characters. The engine was developed using DirectX. The engine was written in .NET C# using SharpDX library. A cross-device communication framework based on bluetooth technology was implemented in this project as well. This communication framework allows two game clients running on two different devices to communicate with each other.

Keywords: multi-platform application, cross-device communication, 3D game engine, DirectX, SharpDX

Table of Contents

Preface

General Introduction

There have been significant development and progress made in the area of portable devices such as smartphones and tablets in the past few years. In 2013 there have been several millions smartphones sold all over the world. [make a reference to number of tablets sold]. With growing number of smartphone and tablet users grows also the demand for good and quality applications. It's quite common to own more than one device nowadays. People use smartphones for everyday tasks such as checking e-mails, reading news online, etc. At work they might be using notebooks or tablets which are getting more and more widespread. And at home they might be using an old good desktop PC. With so many different types of devices out there, it is only natural that we should have a uniform environment and uniform applications and programs which would be able to run on all the devices. There are number of applications that meet this requirement, but not as many as we would like. And that goes even more for games.

Lately, Microsoft has been trying really hard to develop an universal operating system which would be able to run on every popular type of device such as desktop PC, tablet or mobile phone. Realization of this idea came with introduction of Windows 8 operating system family. It consists of Windows 8, Window 8 RT and Windows Phone 8. Windows 8 is meant for desktop PCs and notebooks, Windows 8 RT for tablets and Windows Phone 8 for mobile phones. Even though Windows 8 systems targeted every type of device, they were not as unified and applications were not as universal as intended. An application written for one system was not executable on other systems. If you spent a month developing an application for Windows Phone 8, in the end you were not able to run it on a PC nor a tablet. Windows 8 RT did not help Windows 8 family get more popularity either, since it was strongly rejected by consumers and later by manufacturers as well. Tablet users were not happy with the fact that Windows 8 RT was based on ARM processors so one would only be able to use applications from Windows Store, but not classic

desktop programs. That was a big problem at the time because Windows Store lacked many applications that Android and iOS systems offered. Eventually, Microsoft was forced to accept failure and they stopped the development of Windows RT with the second line of their tablet device Surface 2. Instead, we will find Windows 8 on all Windows tablets these days.

With Windows 8.1 update Microsoft introduced the Universal Application model which allows developers to reuse the code written for Windows 8.1 device to be shared to Windows Phone 8.1 device and vice versa (hereinafter referred to as Windows device, Windows Phone device respectively). It is important to note, that an application written as Universal Application is still not 100% universal. There are certain things regarding adjustment to Windows device or Windows Phone device that need to be done, e.g. writing proper GUIs so they fit different screen sizes. This usually means writing two different codes for both small and large displays. Nevertheless, the Universal Application model brings a good and solid way to do things and it was well-received. In this work we will use the universal application model and try to develop an universal game client which players will be able run on every popular device with Windows 8.1 operating system.

DirectX is one of the two commonly used application programming interfaces (API), if you want to create a graphic application. The other one is OpenGL. In this project, we will choose DirectX developed by Microsoft, since we are targeting Windows devices. If you have any experience with DirectX, you know that DirectX applications can only be written in native C++. However, there are certain libraries that provide us with the ability to access DirectX API from .NET C#. Namely, the two well known libraries are SlimDX and SharpDX. Because SlimDX is no longer being developed, we will be using SharpDX library. Using SharpDX library we will try to write a little complex DirectX application, but in C# and not C++. We will be developing our own 3D game engine which will be able to render simple scenes with objects and animated characters. The character animation will be implemented using skeletal animation. We are certainly not trying to create a complete game engine such as Unity3D or Unreal Engine. We are merely trying to show that it is possible to create a nontrivial 3D graphic application written in .NET C#.

There are really a lot of games out there for every kind of device. PC games have been here for decades. And even though tablets and smartphones are relatively new in comparison to PCs, there is quite a number of great games. A lot of people like to kill some time during the day while they are commuting to work or waiting for someone or they just simply want to relax with some casual game such as Fruit Ninja, Temple Run or any puzzle game. Although all these games are enjoyable and very popular, they are still casual games and players might find them boring after a while. One might say there is no kind of progress that would keep players coming back to the game. Of course, there are games where players can progress - strategy and role-playing games (RPG) are an excellent example. Players are building their towns, realms or improving their heroes over time and sometimes players can compete with each other. The problem is that most of these games are designed so that they can be played online when there is internet connection. What if you spend every day an hour in underground train where there is no signal reception and no internet connection and you are tired of playing casual games? Maybe you would like to play a game that will offer interesting content and good progress system on a smartphone or a tablet. And when you get home where you have internet connection, you want to come to your desktop PC and start your game and continue playing right there where you stopped playing in the train. Or let's say you meet a friend somewhere and there is no internet connection and you would like to compete with each other to see who is better. Most of the games out there nowadays support multi-player only when you are online and the multi-player requires connection to a game server. Wouldn't it be great if you both could just pull out your phone or your tablet and play with each other? These are the problems that we want to address and we will attempt to create a game which you can play wherever you want, on whatever device you want. Your game data will be saved and uploaded to cloud once you come online and your saved data will synchronize across all of your devices. And you will be able to play the game with your friends anytime and anywhere, even if there is no internet connection.

In the end, I would like to discuss the process of making the whole game. Creating a new game from nothing is not a trivial task. Usually there is a whole team of developers involved since there are so many different areas that need to be covered. And in practice it is not possible to do with one-man team. Art team is responsible for tasks like drawing concepts, designing the game characters, the game world and objects in it. Next step would be modeling and animating characters and objects, essentially turning concepts into something real in the game world. Then there needs to be a team of programmers to write an engine, program the game itself and many other things. This is just a simple description of how the process might look. As many programmers dream about creating their own game, they mostly do not know where to start. They do not know what it all involves and what needs to be done. I was no different myself. So last but not least, with this paper I would like to provide those readers, who are thinking about starting to develop their games, but do not know where to start, with a short example of how to make your game.

Content Overview

At the start of this paper we will take a look at some existing games. We will point out some games which might be inspiration for our project or games on which we will show shortcomings to see where we want to be innovative.

After talking about existing games we will have some idea from where we started. In the chapter 2 we will be talking about designing the game. We will introduce game mechanics that will be present in our project and explain them.

When we are done explaining the content and mechanics, we will look at how to turn the ideas and mechanics from the previous chapter to something real. Chapter 3 is the technical documentation, where we talk about possible solutions to encountered problems and how is the engine, the communication framework and the game itself implemented.

Chapter 4 is the User's Guide. In this chapter we teach the players how to play the game. We tell them where they find what they need. "How to do a mission?" "Where can I overview my character statistics?"

The last chapter of this paper will focus on the whole process of making the game from the start. As I was participating on virtually all the stages of making the game, I would like to walk readers through the individual stages, so that readers have an idea of what the whole process involves.

Finally, at the end of this paper we conclude the work that has been done and evaluate our effort. We point out what might have been done better. Also we talk about possible future development.

1. Existing Software

In this first chapter we go over some existing games available today at the market. With these quick reviews, we would like to talk about out certain game mechanics that are interesting and relevant to our game. We highlight significant and characteristic game elements which will bring inspiration to our own game. On the other hand, we point out specific shortcomings in these games that we are going to address.

Before we start going through existing game titles, let us first shortly describe the type of game that we are trying to develop. We would like to create a role-playing game with focus on turn-based combat with action elements. The game should be able to run on various types of devices and it should offer a possibility for players to fight in multiplayer battles no matter on what devices they play. We would like to approach those players who love fantasy role-playing games, slowly progressing and improving their game characters and measuring their skills with friends.

1.1. Pokémon

Pokémon [1] is a video game made for Nintendo [2] handheld consoles such as Gameboy, Nintendo DS, Nintendo 3DS. It is a product of Japanese company Nintendo. There are other Pokémon titles for other Nintendo consoles, but the main focus was always on handheld titles. Pokémon game first saw the light of the world in 1996 with its first two editions named Pokémon Red and Pokémon Blue. A year after Pokémon games were released in Japan, 10.4 million copies were sold there. The total combined sales of Red and Blue in the United States were 9.85 million by 1998. The Pokémon series continues till today and it has sold over 270 million units as of May 2015, giving it the distinction of being one of the best-selling video game series in history.

Pokémon is a role-playing game where players get to play as a teenager hero who is traveling the world, catching and training pokémons and saving the world in the spare time. Every few years Nintendo releases new generation of Pokémon

games and every time it brings some kind of improvements. Mostly they are graphic and cosmetic changes. And of course, every new generation introduces a new generation of pocket monsters. The story is roughly the same - the main hero gets a starting pokémon and then travels the world to discover new pokémons and fights an evil organization which strives for the world dominance.

The one thing that Pokémon is most known for is the turn-based combat system. This is where players spend most of their time in the game. Through the combat they train their pokémons, make them stronger and progress in the story. One has to admit that the turn based combat system is very complex and quite powerful and that a lot of interesting game mechanics can be found in it. Many factors and hidden things contribute to the actual fight. Even though it works really well and it is a lot of fun to play, there is one particular limit that Pokémon combat system will never overcome - the fact that it is based on plain turn-based mechanic. There is no way for a player to affect the outcome of the turn. Let us say that players A and B will use the same attack ten times in a row. The outcome of these ten turns will be more or less the same. If player B has a pokémon with better attributes, player A cannot do anything to improve his/her chances to win the turn. As you can see, the combat feels to be very static. There is no action in it. It is not possible for players to actually execute the attack to express how well the attacks were performed.

I do not in any way dare to criticize a grand piece that Pokémon games surely are. I only want to point out a simple fact that there are certain limits in the game. In any case, Pokémon combat system is great for tactical thinking and players also have to prove themselves in devising different strategies while putting together their teams to work well in different situations. These are the things that I will try to keep in mind while designing my game.

1.2. Blood and Glory

Blood and Glory [3] is an action role-playing game developed by Glu Mobile [4]. You might have seen this game on one of the Android or iPad tablets. Blood and Glory is a clear example of a new mobile game made specifically for tablets and smartphones.

As a representative of games for mobile platforms, it shares their characteristics. Word “simple” captures it well. You will not find any great story here nor great way to progress your character. In this game players are put in an arena to fight with an enemy non-player character (NPC). As you defeat your enemies, you face harder opponents. We mention Blood and Glory in this paper for its action elements in turn-based combat.

In comparison with Pokémon, where you had so many pokémons with so many different attacks, there are not many things you can do here in Blood and Glory. The fight has 2 phases. The first one is where enemy attacks and you have to defend yourself. You can either dodge, block or parry. After enemy tries to hit you a few times, he gets tired and that is where the second phase begins. During this phase you have an opening to counter strike. So basically you just have to avoid attacks and then strike your opponent. The nice thing about Blood and Glory combat is that when enemy NPC is attacking, it is shown as a slowed animation of enemy movement (for example a swing of a sword from left to right), and you have to react accordingly. In short, the developers introduced an interesting real-time action elements in turn-based combat system.

Another thing worth mentioning is the way you control your character with touch commands in the game. All the commands that you do in the game is tapping on the screen or swiping with your finger. It is really important to have an easy and intuitive way to control your game character on the touch device because many games fail to deliver this. Programming character control on a touch device is different than doing it on PC where you have keyboard and mouse. Many developers try to create character control on a touch device as they would do it on PC and it just does not work well. Blood and Glory has solved this really well. The touch input is responsive and intuitive to use.

Even though Blood and Glory is not as complex game as Pokémon, players could spend hours, days and weeks in the arena dodging and striking enemy NPC gladiators. All it took was bringing little action into a turn-based system.

1.3. Dragon Mania Legends

Dragon Mania Legends [5] came from workshop of one of the great mobile games developer Gameloft [6] in the first half of 2015 and immediately became a hit. It is a game similar to Pokémon, but players breed and raise dragons instead of pocket monsters. It has a little bit of strategy game flavour mixed in. Besides raising and training dragons, players have to take care of an island that was given to them. There is also a nice single player campaign with a short story that will keep players busy for a while.

The combat system in Dragon Mania is also turn-based like we saw at Pokémon games. It is nowhere near Pokémon's combat mechanics, but there is a new interesting element introduced in Dragon Mania. Unlike in Pokémon, players can affect how well they perform an attack. Two teams of 3 dragons stand against each other in one fight of Dragon Mania. As we said before, the fight has turns. In each turn player selects a dragon's attack and a target and then they execute the attack. The execution lies in stopping the moving indicator in the allowed slim interval. If player stops the indicator in a very slim green area, they get bonus and deal extra damage. If they land on a little wider yellow area, they deal normal damage. And if they even miss the yellow area, they miss the attack completely. Players have to pay attention and focus on well performing the attacks. Also in comparison with Blood and Glory, players have to think tactically, because each of their dragon is of different type and has different type of attacks. Some types are better used in various situations, so they have to play smart. We could say that the combat system in Dragon Mania is a combination of tactical gameplay of Pokémon and a little action like in Blood and Glory. Although, we must say, that there is not as much action as in Blood and Glory.

One of the great things about Dragon Mania is that it is multiplatform. You can run it on PCs, tablets or smartphones. With a great expansion of mobile devices, it is important to target as many type of devices as possible, thus expanding the player base. For players that play Dragon Mania on more than one device, their game data is saved to their account on the game server. So while they are working on PC, they can peek in the game for a little while, do some things and then later they can log in on smartphone and continue where they left off. What Dragon Mania really lacks is multiplayer gaming. Even though it is an online game, you cannot challenge another player for a friendly duel in any way. In addition, you have to be connected to the internet to be able to play. So you can forget about feeding your dragons while commuting to work or casual quick fight with NPCs.

1.4. Ninshu Arts

For the purposes of my project and hereinafter in the paper, my game will be called Ninshu Arts project.

We were talking about Pokémon game series and its powerful turn-based combat system with a lot of possibilities and requirement for tactical thinking, but lack of action elements in the game. Then we went through Blood and Glory where there was a lot of action in the turn-based combat but little complexity and not many things to do. At last, there was Dragon Mania Legends where we saw little from both worlds, but not exceeding in any of the two.

We have talked about interesting game elements and strong points of the previously mentioned games. That is where I seek inspiration and I will try to incorporate similar interesting things into the game. We also talked about shortcomings of those three games which I will try to address. My game will have turn-based combat with non-trivial combat technique system, complex enough to make things interesting. Also I will implement action system to my turn-based combat to make players alert, responsive and mindful of the game. Also with a complex combat technique system, players will also have to think tactically.

Last but not the least, my game will be multiplatform and will target every type of device running Windows 8.1 operating system. Finally unlike Dragon Mania Legends, players will have the opportunity to fight with each other through bluetooth communication framework. In other words, we will connect the players playing on different devices and bring them together with the cross-device communication framework. The last two features are not likely to be found in any other games at the market these days. To have a multiplatform game running on different types of devices with the ability to communicate cross-device is a rarity.

Note: Hereinafter we will refer to Windows 8.1 and Windows Phone 8.1 as Win8 and WP8 respectively.

2. Designing the Game

In this chapter we look into the game designing phase. I introduce all important game mechanics that make my game stand out with respect to the things that we pointed out in the previous chapter. After the readers finish this chapter, they should have a good idea of what features the game will have and what players will be able to do in the game.

2.1. Game Concept

Ninshu Arts is a role playing game. At the start the players create their own character called *Shinobi*[1] in the world of the game. The goal of the game is to progress your character through *Missions* and *Hunting* assignments (both will be explained later in section 2.3. Progressing Your Character) and improving your battle skills to duel your fellow shinobis in multi-player battles. The strength of a shinobi is determined by his attributes (stats) and by combat techniques that he can use. Combat techniques are called *Jutsus*[2]. As players progress in game, they obtain and learn new jutsus to have more tools to use in combat.

The readers of Japanese manga[3] series *Naruto*[4] are no strangers to the terms "shinobi" and "jutsu". It is no coincidence that we are using the same terms, since the Ninshu Arts world is inspired by the story of Naruto. We could describe it as a fantasy world with a little taste of Asian culture. There are several ninja villages in it where shinobis live. Shinobis go on missions to improve their village status. Villages might be in peace, compete or be at war with each other, so it is a primary objective for a shinobi to grow stronger and stronger.

---

[1] *Shinobi* or Ninja is a term for a covert agent in feudal Japan.

[2] *Jutsu* is a word from Japanese which translated means "technique", "method" or "spell".

[3] *Manga* are comics created in Japan or by Japanese creators conforming to a style developed in Japan in the late 19th century.

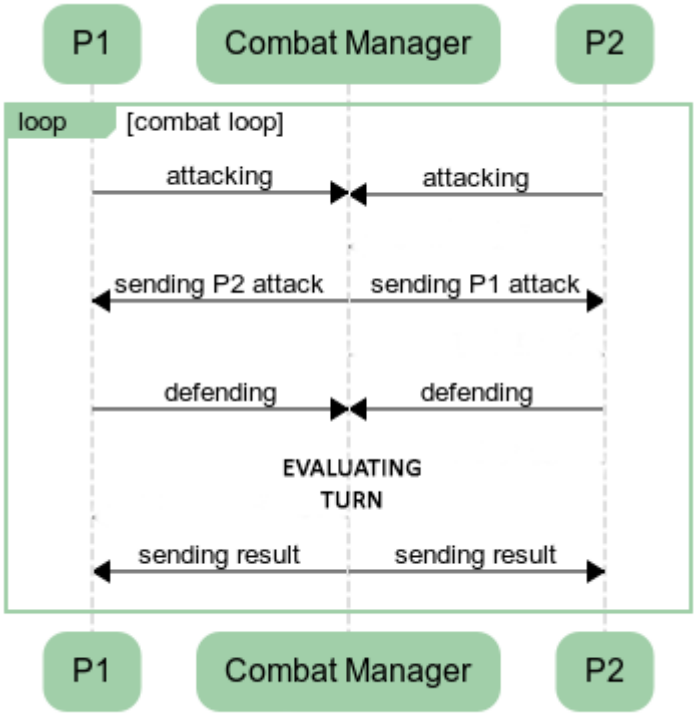
[4] *Naruto* is a Japanese manga series written and illustrated by Masashi Kishimoto.

2.2. Combat System

The combat system is the core and most important part of the game. This is where players will spend most of their time. All the collecting and learning new jutsus and training is only to get stronger to be able to maximize their combat skills and capabilities. Whether players are on an NPCs on hunting assignments or in a multiplayer battle arena, they will be fighting using our combat system. Let us introduce the key features of the combat system now.

2.2.1. Turn-based Combat with Action Elements

Battles in Ninshu Arts are conducted in a turn-based manner. Each turn consists of an *attack phase* and a *defense phase*. At the start of a new turn, players select their jutsu that they want to use. Then the attack phase begins. Players have to perform the attack moves during the attack phase. . Each jutsu can have one or more hits. When players execute their jutsu, they need to do the attack to the right or the left direction for the particular hit. These swipes to the left or right will form a combo which then later the defender will have to counter. The defense phase begins after the attack jutsu execution. The defender has to counter an incoming attack combo in this phase. We can imagine this as if attacker sent several shots coming to us from the left or right side and we have to counter them. The defender needs to swipe to the direction that a shot is coming from to counter it. If the defender counters a hit, (s)he will be inflicted with reduced damage, and if not, (s)he will receive full damage. By watching the enemy movement, the defender will be able to predict from which direction the attacks will be coming from. So players will have to perform the actual attacks and then they will have to observe the enemy to be able to react properly to defend themselves. After the defense phase the turn is evaluated and both players will receive calculated damage based on their defense action.



Picture 1.: Combat turn model

It is important to note that during the attack phase both shinobis are attacking at the same time and they are defending at the same time during the defense phase as well. We can imagine this as both of them are performing a jutsu, then they shoot a combo at the other at the same time and when an incoming shots approach, they have to counter them, again at the same time.

In addition, we propose one more feature in order to make executing attack and defense phase a little more interesting. We want to make sure that the execution part will not be too trivial. We require that in particular the swipes of attack or defense combo are performed at a certain time, more or less. Each jutsu has times configured in its definition so that swipes must be performed either exactly at those defined times or in a allowed time interval. If a swipe was made within the allowed time margin, it is considered a hit, otherwise we consider it a miss. Only hit swipes will be included in damage calculations, missed swipes will be ignored. On top of that, the defender will have to perform defense swipes right at those times that the attacker performed attack swipes. We could visualize this feature as a panel with a moving indicator and marked time intervals. And players need to do the swipes when

the indicator is within a marked interval. With this feature we want to make players pay more attention to a battle and bring a little more complexity to the combat.

2.2.2. Combat Techniques - Jutsus

Combat techniques or jutsus, as we called them, are tools for shinobis to use in battles. Each jutsu is different and unique in its own way. Players will start with some basic jutsus and they will obtain new jutsus as they progress in the game. New jutsus mean new options in combat.

Each jutsu is based on one of the basic elements - Fire, Wind, Lightning, Earth and Water. We will explain each element in detail later in section 2.2.3. Elements. If you recall, we mentioned earlier that a jutsu can have one or more hits. Specifically, it can have up to three hits. In raw numbers, more hits usually mean that you can deal more damage.

Every jutsu is divided into two types - offensive and defensive. *Offensive jutsus* usually have more hits and the hits deal more damage than hits of defensive jutsus. On the other hand, defensive jutsus have a special support combat effect depending on the jutsu's element (more on combat effects later). Combat effects allow us to manipulate the enemy in different ways. As defensive jutsus and their combat effects might be quite powerful, it takes more time until they can be used again. In other words, defensive jutsus have bigger cooldown.

Jutsu's *cooldown* tells us in how many turns can we use the jutsu again. Not only that defensive jutsus have bigger cooldown than offensive jutsus, but the stronger a jutsu is the longer its cooldown will be. That said, it is essential to plan your moves wisely, because you do not want to use your super jutsu that deals a lot of damage when you are affected by enemy jutsu and your damage is rapidly reduced, since you will not be able to use that powerful jutsu again for a long time.

2.2.3. Basic Elements

There are five basic elements in the world of Ninshu - Fire, Wind, Lightning, Earth and Water. Every shinobi is born with affinity for one element, which will be their primary element. That means training and using techniques of this primary element will come most natural to them. In time as shinobis progress, they will learn to control and use other elements, though they will have to put in more effort to training them. It is important for shinobis to master more elements because each element has its special properties and different combat effect.

Let us now establish the meaning of terms "buff" and "debuff". We say that a shinobi is *buffed*, if (s)he is affected by a positive combat effect. On the other hand, the shinobi that is affected by a negative combat effect is called *debuffed*. A positive combat effect will be designated with a term *buff*, whereas negative combat effect *debuff*.

Fire is the element of power. Its special effect is called *Scorch*. If shinobi uses a defensive jutsu with the scorch effect, when (s)he is hit by an incoming attack, (s)he retaliates against the attacker and deals them some damage in return. The more attacker hits the more (s)he is retaliated. Defensive fire techniques are used when player wants to deal extra damage to his/her opponent or player expects that in the current turn the opponent will hit hard.

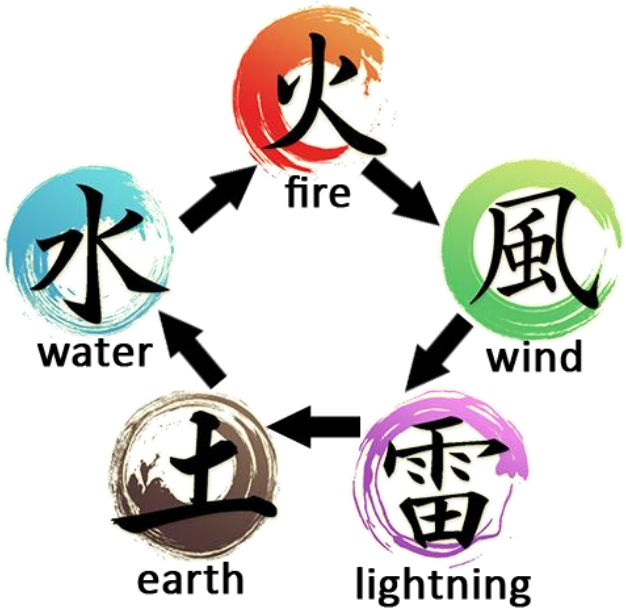
Wind is the element of swiftness. It can be used to affect the opponents so they deal less damage in the next few turns. This combat effect is called *Daze* effect. If shinobi uses a jutsu with daze effect, enemy shinobi will be dazed and have his/her damage reduced. Players should use defensive wind jutsus when they want to make sure that the opponent will be unable to hit hard for a short time.

Lightning is the element of unpredictableness. Its combat effect is called *Electrify* and it affects the enemy cooldowns. When shinobi uses a defensive lightning jutsu, the enemy shinobi is then electrified and his/her cooldown is increased. Defensive lightning jutsus are perfect for delaying the opponent's big jutsus. They are great, if you want to buy some time.

Earth is the element of sturdiness. When shinobi uses a defensive jutsu of earth element, (s)he gains *Absorb* effect. With this buff (s)he will absorb a part of the incoming damage. Players should use defensive earth techniques when they expect to be hit hard and they want to reduce the incoming damage right away.

Water is the element of recovery. Defensive water jutsus have *Heal* effect. When shinobi uses a technique with heal effect, (s)he recovers some health at the end of the turn. Note that absorb and heal effects may appear to be similar, but absorb only blocks incoming damage. After the turn you cannot end up with more health than in the previous turn using a jutsu with absorb effect, whereas it is possible, if you used a jutsu with heal effect. That is the reason why earth techniques will absorb more damage than water techniques will heal, if we consider earth and water techniques of the same level.

Now that we have introduced all five basic elements, it is important that we talk about elemental strengths and weaknesses. Every element is strong against a different element and weak against another element. Strengths and weaknesses of each element against others can be seen here in the following elemental wheel:



Picture 2. Elemental Wheel

As you can see from the picture, fire is strong against wind, but weak against water. Wind is strong against lightning, but weak against fire etc. What it means is that if you use one jutsu against opponent's jutsu which is based on element stronger against your jutsu's element, your jutsu will lose against your opponent's jutsu. As a result, your jutsu's damage will be reduced, while your opponent's jutsu damage will be increased.

Note: some dependencies in the elemental wheel might not make sense on the first sight. That is because our elemental wheel is adopted from Naruto manga series which is originally derived from Chinese horoscope elements. Chinese "cosmic elements", as they call it, are fire, metal, wood, earth and water. Fire destroys metal, because fire melts metal. Metal destroys wood, symbolizing metal chopping down trees. Wood destroys earth, since roots splits earth. Earth destroys water, meaning water is absorbed and lost in earth. Masashi Kishimoto, author of Naruto, then replaced metal element with wind element and wood element with lightning element, probably because wind and lightning are more suited when we talk about combat. And the strengths and weaknesses were kept as they were. This is how the elemental wheel came to be.

2.3. Game Character - Shinobi

A shinobi, as we called the characters of Ninshu Arts world, is a player's representation in the game world. One of the primary goals of every role-playing game is to improve your character, make it better and stronger. Strength and capabilities of your shinobi is described by your shinobi's attributes.

2.3.1. Shinobi's Stats

Attributes are the basic building blocks for a character's combat ability. These attributes are generally called *stats* in gaming terminology, as an abbreviation for "statistics". Here in the paper we will stick to the term stats and we will use it normally. There are five main stats in Ninshu Arts - stamina, speed, attack, defense and resistance. Each of them is important and each describes a shinobi's characteristics.

Stamina attribute determines how many health points a shinobi has. It is important for players to improve this stat, because more health means player's shinobi will last longer in battle.

Speed attribute says how fast your shinobi is compared to your opponent's shinobi. If your shinobi is faster than your opponent's, you will have more time to perform a jutsu combo and more time to defend against enemy attacks. However, if your shinobi is slower than opponent's shinobi, you will have less time to perform attack and defense actions.

Attack stat directly corresponds to shinobi's attack power. The bigger attack stat a shinobi has the bigger his/her attack will be. This stat is used in formulas to compute final damage that a shinobi deals.

Defense stat increases shinobi's defense capabilities. More defense means that a shinobi will take less damage. This stat is used in formulas to compute final damage that a shinobi receives.

Resistance stat is important for countering incoming attacks. High resistance attribute will reduce more damage on a good counter. The more resistance your shinobi has, the more damage can (s)he block when (s)he successfully counters a shot.

2.3.2. Experience and Levels

In Ninshu Arts we would like to propose a slightly different way to interpret experience and levels than other games. Players earn *experience* points through various actions in game. When they reach certain amount of experience points, they level up. Traditionally, in most games *levels* tell us how strong a player is. The higher player's level is, the better (s)he is.

From the previous description of experience and level model we can see that it is virtually impossible for a player with a low level to defeat a player on a high level. In reality it means that if a player spent longer time in game, (s)he defeats all those players who spent less time. This traditional model has one flaw - it does not

take player's skill into account. A player can be really skilled and good in combat, but (s)he will never be able to defeat a player on a higher level, even if that player is not very skillful.

Here in Ninshu Arts we propose a little different approach. Levels will only indicate how long the player has played the game. Higher level means that a player has already spent more time in game. However it will not give him/her a dominance in combat. Combat system will only take player's stats into account and those will be a primary factor for computing damage output. Experience points will be gained normally for most of game actions. Players will be rewarded for completing missions, hunting assignments and fighting other players in Player vs Player battles (PvP). When players reach a certain amount of experience points, they will gain a new level. Of course we want to reward those players that have been playing more. Time and effort put into game must be rewarded. So players with higher level will have advantage over those players with lower level, but only advantage. It is still possible for a lower player to defeat a higher player, if (s)he is better in tactical thinking or more skilled in combat. We want players to stay alert and not think that higher level will provide them the total dominance in game.

2.3.3. Progressing and Improving

There are two ways to progress and improve your character in Ninshu Arts - active and passive way. We want to provide players with option to play actively if they have time and want to. And we also want to give players a way to keep up with active players and improve their shinobi's stats even if they do not have time to play actively all the time. It goes without saying that active progressing will be more rewarding, but passive progressing will not be much worse.

Missions will be a passive way to progress your character. Players will be able to send their shinobi on various missions that take a certain amount of time to complete. During this time the shinobi is away and players will not be able to perform any other actions. When the shinobi is finished, (s)he will come back to village and will be rewarded with stats and possibly get a bonus reward. There will be several levels of missions. At the start, players will only be able to go on easy

missions with minor rewards that take less time to complete. As players progress and are more experienced, they will get access to more challenging missions with a prospect to better rewards.

Besides missions, players will be able to send their shinobis on *Hunting assignments*. Shinobis will be tasked to hunt down the enemies of the village and defeat them. On this hunting assignments players will fight NPC shinobis. Like missions, players will start with easy hunting assignments where players will have to fight low level NPCs. Later, as they get stronger, they will be able to fight stronger NPCs with higher levels, higher stats and better jutsus. After an NPC is defeated, players will gain rewards for completing the assignment.

2.4. Multiplatformity

One of our primary objective is to create a multiplatform game. We have to keep that in mind in all the stages of the game development (for the development stages see subchapter 5.1.). It goes without saying that players will have dissimilar gaming experience while playing on a PC and on a tablet. On a desktop PCs and notebooks, players can use keyboards and mice. In the gaming world, this is how it has always been done. Players have the most control and precision while playing on PCs or notebooks.

On the other hand, touch devices are fairly new in comparison to desktop PCs. They have their own perks, since controlling things with touch might feel a little more natural and intuitive. On the other hand, touch input is definitely less precise than mouse and in addition you are covering your view with your hand. Note that in last years there was a huge expansion of hybrid devices (sometimes called 2 in 1 devices). It could be a notebook with a touchscreen monitor or a tablet with attachable keyboard or a notebook that can be flipped in various ways. So this kind of devices may have capabilities from both worlds. Also we should point out that not all touch devices are alike. Some support multi-touch when the screen of the device captures the movement of more fingers. And some devices have larger screen and some only small screen (typically smartphones or small 7 inch tablets).

Since playing games on various types of devices is different, we have to come up with a way to bring the players a comfortable, intuitive and responsive way to control the game. This especially needs to be reflected in combat because battling is going to be the main fun feature in Ninshu Arts. When we think about all the ways to control different devices, we can only go as far as the least capable device. We want to target as many devices as possible which means we have to consider supporting only the most basic input capabilities. Namely, we can only consider touch input with single finger (no multi-touch). Smartphones and tablets usually do not have keyboards, so that is out of question. As for PCs and notebooks that do not support touch input, we can easily simulate the behaviour with mouse. That is why we limited the touch input to single finger earlier.

With that in mind, we ought to talk about controlling the game. Operations on game screens and navigating through screens is typically done with mouse clicks and touch tap gestures. Let us have a look at controlling a game character in battles now. Imagine a battle scene with two characters in it. One character represents your shinobi which you are able to control and the other character is your opponent's. Enemy character is either controlled by a human opponent or an NPC unit. Now if you recall, one of the other objectives of this project is to create an engine capable of rendering and animating characters. We would like to have an elegant way to tell our character to do the right punch or left kick which would then be shown on the screen with animations of our character doing the fighting moves.

If we were programming this as a PC game, we would simply take the user input from a keyboard. Bind several keys to certain actions and animate the character according to keys that were pressed. But as we said earlier, we want to design a system that relies only on touch and mouse input. One of the first ideas how to deal with this problem would be displaying a few buttons on the touch screen and simulate the behavior as described with keyboard. This is a possible way to do it, indeed. There are mobile games that work on this principle, role-playing game series Dungeon Hunter [7] from Gameloft is a good example. Sadly, this might not be the best solution for us, because we have to consider even small screens on smartphones

and having a lot of buttons on the screen might not look good and tapping small buttons might not be comfortable.

Now what if we could just control the character with swiping a finger over a touch screen. If we think about it, we could just require direction command from the player. Swiping in a particular direction tells the game what it needs. On a device without a touchscreen we could just take mouse input instead. Clicking and dragging a mouse in a direction is in fact the same as swiping a finger. Of course, if we decide to use this system, we will have to make a few compromises. Specifically, we cannot distinguish a command for a kick or a punch. One way how to cope with this issue is not to distinguish kicks and punches and only require a direction information.

Even if we do not distinguish kicks and punches and take only direction information into account, it would not look very action if characters were doing only punches and simple moves. We need to incorporate both punches and kicks animations into the combo to make a game look action. We could organize the animations so that punches are used at the beginning and the middle of attack combos and kicks will be used as a finishing blows of the combos. This way we will keep the nice action feeling. With these compromises we hope to have a intuitive and comfortable way for players to control their characters in battles and still have a challenging and action combat to play.

2.5. Multi-player

As we mentioned previously in the paper, games for mobile devices do not support multi-player much. And if they do, it is mostly only the option to play online. Games for mobile devices and even for PCs do not support offline multi-player. It is not possible for 2 players to come together and have a multi-player battle with no network or internet connection available. This is where Ninshu Arts come in. We design Ninshu Arts with emphasis on multi-player battles and PvP. What better reasons would players have for spending time and effort improving and training their shinobi than to measure their strengths and skill against other players?

2.5.1. Cross-device Multiplayer

Back in 1990s when Gameboy Color was a hit, players were able to come together, connect their Gameboys with a Game Link Cable [8] and have battles. Nowadays, communication between devices are all based on wireless technologies. It is only logical to be using wireless communication, especially if we are developing a multiplatform game that is supposed to run on different types of devices, since having different types of cables to connect various devices would be highly impractical. In fact, it is possible to connect different types of devices, usually using USB, mini USB or micro USB cables and their variations, but there are no available APIs that we could base our communication framework on. There are two wireless technologies that would suit our purposes - Wi-Fi and Bluetooth. These two technologies are very common these days and present in virtually all devices.

Originally, *Wi-Fi* does not support peer-to-peer (P2P) connection. Wi-Fi is mainly seen in Personal Area Networks (PAN) where it connects devices to a local network and provides access to internet using a Wi-Fi router. This is not exactly what we want, since we want players to be able to play with each other even if they are not connected to a local network. With introduction of *Wi-Fi Direct* technology [9] new doors opened for us. Wi-Fi Direct offers a possibility to have a P2P connection between two devices which have Wi-Fi adapters. This P2P kind of connection was only possible through Bluetooth technology. Even though Wi-Fi Direct seems very promising and offers new possibilities to us, we will not be using it for Ninshu Arts project. Sadly, it is still relatively a new technology and not all devices support Wi-Fi Direct. Namely, WP8 operating system does not have any working API to access Wi-Fi Direct. However, Win8 operating system is capable of working with Wi-Fi Direct.

Our other option is to use *Bluetooth* [10]. Bluetooth is not a new technology[5]. It has been in development for more than ten years now and it is quite a common capability of nowadays devices. We will be using Bluetooth Rfcomm which is a Windows Runtime API available in both Win8 and WP8 operating systems. This is

[5] Bluetooth 1.0 was released in 1999. First mobile phone with Bluetooth comes to market in 2000. For more information please read [11].

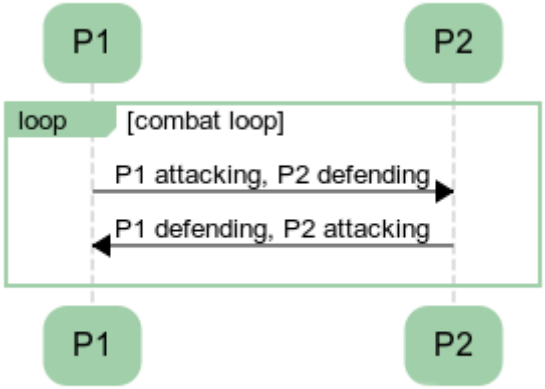
actually perfect for the purposes of the project, since Bluetooth Rfcomm API is accessible from all kinds of devices running Win8. Furthermore, virtually all smartphones have a Bluetooth adapter, most of tablets and notebooks have a Bluetooth and some PCs that come with Bluetooth present on their motherboards. And if not, buying a new Bluetooth adapter that is plugged into a USB port is not a big deal, because they are really cheap. Since combat in Ninshu Arts is turn-based, we will only need to send messages from one device to another. Bluetooth will handle this task without any problems.

For the record, we would like to mention Unity3D Engine which is one of the two most favourite multiplatform game engines used to develop games (the other being Unreal Engine 4). At the time of writing this paper, Unity3D only supports Bluetooth communication for Android and iOS devices which means that developers are only able to create multi-player games that communicate over Bluetooth from phone to phone or tablet to tablet, basicly touch devices. There is no way to implement Bluetooth communication in Unity3D for PCs and notebooks. In other words, Ninshu Arts might really be one of a kind.

2.5.2. Turn-based Combat Model for Multi-player

There were some choices that have to be made for the turn-based combat model to work well in multiplayer battles. Namely we have to come up with a flexible turn-based model for combat so it would work properly with a communication framework. We will have a *Combat Manager* unit that takes care of managing the combat, calculating damage and other effects, storing data about shinobis in battle etc. Since we are working with a turn-based system, we will essentially be sending messages from players to the Combat Manager and it will evaluate actions and send results back to players, again through messages.

Our primary objective while designing how the combat turns will look like was to make the combat as much action as possible. To achieve that, players have to react to each others actions. One way to do this would be having one player attack while other player defends in one turn. One player would perform an attack and the other would have to perform appropriate defense action to counter the incoming attack. Then the turn would end, new turn would begin and the roles would switch. The other player would become attacker and the first player defender etc. The described model is shown on the following picture:



Picture 3.: Combat turn model

Even though most people would find this turn-based model as most intuitive, it is not suited to be expanded to multi PvP battles where more than 2 players participate. And we have to take that into account, because multi PvP battles are on list of things to be implemented in the future (feature 3. in the future development list, see subchapter 5.4.).

With that in mind, we have to adjust the previous model a little so the combat is easily extendable for more than 2 players. As described earlier in the section 2.2.1., we proposed a turn-based model where players attack at the same time and defend at the same time. Basically both players perform attack action at the same time, they send messages with details of the performed attack action to the Combat Manager unit, which sends the attack actions of the opponents to players. Players will then need to react properly and counter the incoming attacks with a well executed defense action. Defend actions are then again sent to Combat Manager in a form of messages.

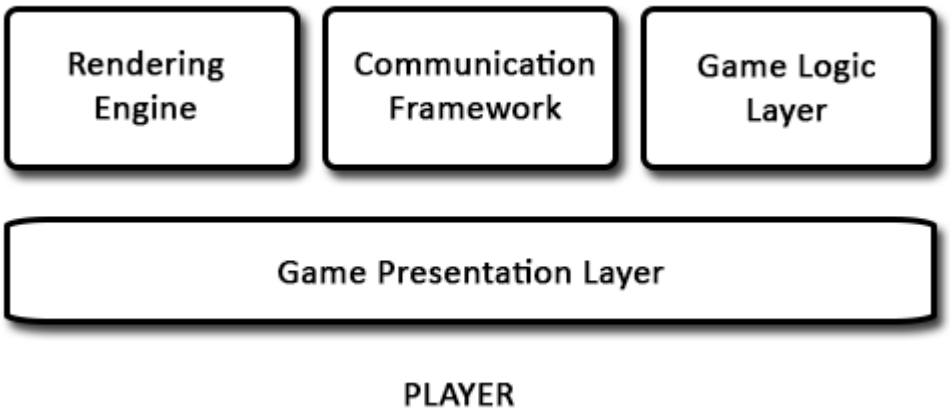
Combat Manager evaluates attack and defend actions, creating the result of the last turn and sends the turn result to both players (for better understanding see the picture 1. in section 2.2.1.). As you can see, this proposed model is more flexible than the first described model. With this model, we can expand PvP battles to multi PvP battles to offer players an option to play 2v2 or 3v3 battles. We will stick with this model and implement it in our game.

3. Technical Documentation

In this chapter we will discuss the process of implementing the game itself. The game consists of a few main building blocks:

- game rendering engine (described in subchapter 3.1.)
- communication framework (described in subchapter 3.2.)
- game logic layer (described in subchapter 3.3.)
- game presentation layer (described in subchapter 3.4.)

The rendering engine is a DirectX [24] application that is supposed to bring players into a combat scene where the battles take place and show players what is actually happening in the battle. The communication framework is responsible for exchanging data across game clients running on various devices. The game logic layer is where we implement how the game is supposed to behave, the game features and mechanics etc. The game presentation layer is what players will see on screens of their devices. Thanks to the presentation layer players can interact with the game. It is what connects players with the game. In following subchapters we discuss each of the building blocks in detail.



Picture 4.: Game building blocks

3.1. Writing the Rendering Engine

The Game Rendering Engine is a unit responsible for rendering a 3D scene and animating characters. It is used as a core unit for battles since it provides players with a look into a battle scene and shows them all that is happening in the fight. In essence, the rendering engine is a DirectX application. As DirectX natively supports only C++, you cannot develop a 3D graphic application in any other programming language than C++, such as .NET C#. Fortunately for us, there is a way how to access DirectX API from .NET C# and it is called SharpDX. As we mentioned at the beginning of this paper, one of our primary challenges is to try and develop non-trivial DirectX application written in a different language than C++. To sum up, in this chapter we describe the process of writing a DirectX 3D graphic application to render a 3D scene and animating characters in .NET C#.

A Direct3D Application typically consists of three stages:

- *Initialization*: This is a stage where we create a Direct3D device and resources. We create a swap chain object here with proper swap chain settings.
- *Render loop*: In this stage, we execute our rendering commands and logic. This is the most important part of the application since everything that a user sees on the screen is programmed here.
- *Finalization*: We clean up and free any resources here. The finalization stage is quite simple. After the render loop exits, we clean up any resources that were created and dispose of the device and swap chain.

3.1.1. SharpDX

SharpDX [13] is a free and active open-source project that delivers a full-featured Managed DirectX API including support for Direct3D9, Direct3D10.1, Direct3D11, Direct3D11.1, Direct2D1, Direct2D1.1 and others. SharpDX supports all Windows operating systems and .NET framework, including development for the

latest Win8 Metro[6] applications and WP8 applications. The support for Win8 applications is most important for our project because we have to keep in mind that we are developing a universal multi-platform app, which targets all types of devices, not a regular desktop application.

3.1.2. Direct3D Introduction

Direct3D is a part of the larger DirectX API comprised of many APIs that sits between applications and the graphics hardware drivers. For the purposes of this project, we will be working with Direct3D 11. Let us talk about some components that Direct3D consists of. In the remainder of the section 3.1.2., we will have a brief introduction to programming with DirectX.

Device

The main role of the device is to enumerate the capabilities of the display adapter(s) and to create resources. Applications typically have a single device instantiated. There has to be at least one device to use the features of Direct3D. Unlike previous versions of Direct3D, in Direct3D 11 the device is thread-safe, which means that resources can be created from any thread.

With Direct3D 11, Microsoft introduced Direct3D *feature levels* to manage the differences between video cards. The feature levels define a matrix of Direct3D features that are mandatory or optional for to implement in order to meet the requirements for a specific feature level. This is an important aspect for us because we have to consider a lot of different devices with various graphic processor units (GPU). We will come back to feature levels later (little research was done regarding this matter, see subchapter 5.3.).

[6] *Metro application* is an old way of calling Windows 8/8.1 applications. Microsoft now uses *Windows Store application* which is a program different than regular desktop programs and is able to run only on Windows 8/8.1 or Windows Phone 8/8.1 operating systems.

Device Context

All rendering functions are encapsulated in the device context. These include setting the pipeline state and generating rendering commands with resources created on the device. Two types of device context exist in Direct3D 11, the immediate context and deferred context. These implement immediate and deferred rendering respectively[7].

The *immediate context* provides access to data on the GPU and the ability to execute or playback command lists immediately. Each device has a single immediate context and only one thread may access the context at the same time. All commands that are to be executed eventually must pass through the immediate context. The same rendering methods as for immediate context are available on a *deferred context*. Nevertheless, the commands are added to a queue (called a command list) for later execution.

Swap Chains

A swap chain helps us with the creation of one or more back buffers. These buffers store rendered data before presenting to an output display. The swap chain takes care of the low-level presentation of this data.

Swap chains are part of the *DirectX Graphics Infrastructure* (DXGI) API. It is responsible for enumerating graphics adapters, display modes, defining buffer formats, sharing resources between processes, and finally presenting rendered frames to an output device (via the swap chain).

[7] Immediate and deferred rendering are two ways of rendering supported by Direct3D 11. For more information, please read an article on MSDN [29].

Resource Views

Resources must first have views before they can be used within a stage of the pipeline. These views describe to the pipeline stages what format to expect the resource in and what region of the resource to access. For example, type of resource views include:

- Depth Stencil View (DSV)
- Render Target View (RTV)
- Shader Resource View (SRV)
- Unordered Access View (UAV)

Buffers

A buffer resource provides structured and unstructured data to stages of the graphics pipeline. The usage is defined by how and where it is bound to the pipeline. Type of buffer resources include:

- Vertex buffer
- Index buffer
- Constant buffer
- Unordered access buffer

Shaders and High Level Shader Language

The graphics pipeline consists of fixed function and programmable stages. The programmable stages are called *shaders* which are small programs written in *High Level Shader Language* (HLSL). The HLSL is implemented with a series of shader models, each building upon the previous version. Each shader model version supports a set of shader profiles, which represent the target pipeline stage to compile a shader. Direct3D 11 introduces Shader Model 5 (SM5). And for example, shader profile `ps_5_0` indicated a shader is for use in the pixel shader stage and requires SM5. Although it is best to use the newest shader model, it is not always possible. In our case, we will not always be able to use SM5 (for more information see the table in section 5.3.). Specifically, we will have to make some compromises when it goes

to WP8 devices, since they do not support feature level `11_0` which means they do not support SM5 either. More on this problem later.

Graphics Pipeline

The graphics pipeline is comprised of nine distinct stages that are generally used to create 2D raster representations of 3D scenes. That means taking the 3D model and turning it into what we see on the display. Four of these stages are fixed function and the remaining five programmable stages - shaders. As our engine is supposed to provide basic functionality, we will only be needing Vertex Shader (VS) and Pixel Shader (PS). In addition, it is a good practice to keep the number of stages to a minimum to ensure faster rendering. Among other stages, there are Hull Shader, Tessellator, Domain Shader, Geometry Shader[8].

3.1.3. Rendering Framework

The rendering framework is a set of classes which will make it easier for us to work with complex scenes. This framework will take care of initializing our Direct3D device, swap chain and render targets. It will also provide appropriate methods and events for managing Direct3D resource lifecycle in our Direct3D application.

In our project, we implemented three distinct libraries which our rendering framework uses. They are `Common`, `Common.WinRT` and `Common.WP`. As their names might suggest, `Common.WinRT` library contains classes needed for a Win8 part of the application to work, while `Common.WP` contains classes for WP8 part. In fact, they contain exactly the same classes, but our rendering framework demands that there are two separate libraries. The reason for this is that SharpDX has different set of DLL files for Win8 applications and WP8 applications. Basicly, it means that our `Common.WinRT` library uses SharpDX DLL files for Win8 applications, while `Common.WP` library uses DLL files designed for WP8 applications. Our Windows Universal Application then includes the library that is needs at the moment.

[8] For more detail on other stages of the graphics pipeline and basics on programming for DirectX we refer readers to the book Direct3D Rendering Cookbook [14], chapter 1.

One might wonder why Common library is there. Especially when Common.WP and Common.WinRT libraries cover all types of devices that run Win8 OS. There are two simple reasons. First, the Common library contains common classes used by both Common.WinRT and Common.WP. And second, it contains classes which can be used to develop a desktop version of Direct3D application. Thus, Common.WinRT and Common.WP libraries only include classes relevant for Windows Store applications, while Common library has classes that are usable even in desktop applications. We designed the libraries this way to leave us with the possibility to implement the desktop client of the game in the future.

We will not go into much detail for every source file here. Most of the code in the rendering framework classes are straightforward commands with no deep meaning. For example, commands to create devices, initialize swap chains etc. We will only describe a larger picture so that the reader has the whole idea of how it all fits together since this is an uninteresting part of this work[9].

The three key elements of our rendering framework are following:

- *Device manager*: This is a class that manages the lifecycle of the Direct3D device and device context.
- *Direct3D application*: This is a set of classes that manage the swap chain and render targets, along with other common-size dependent resources (such as depth/stencil buffer and the viewport setup). We descend from one of these to create our render loop.
- *Renderer*: This is a class that implements a renderer for a single element of the scene. We create instances of these within our Direct3D application class.

[9] For more info on initialization commands and their detailed description we refer readers to Direct3D Rendering Cookbook [14], chapter 1 and 2; and the Code Documentation on the DVD attachment of this paper and the source code of this project.

Device Manager

We have `DeviceManager.cs` file in `Common` library. This class corresponds to the device manager element of the rendering framework. It takes care of creating our Direct3D device and context within its `Initialize` function.

Direct3D Application

The `D3DApplicationBase.cs` base class provides appropriate methods that can be overridden to participate within the rendering process and Direct3D resource management. Specific Direct3D application classes can then descent from the base class and implement the abstract method `Run()` in order to provide a render loop. Specifically, we have `D3DApplicationWinRT.cs` and `D3DApplicationWP.cs` classes descending our base class that implement a Direct3D application for Win8 application and WP8 application respectively. The source code of both mentioned files is short and fairly self-explanatory on the first look. If we wanted to write an application for a desktop client, we would create another `D3DApplicationDesktop.cs` class to implement things for a desktop app.

Renderer Classes

Let us start on interesting things here. Renderer classes provide us with ability to actually render something. All rendered objects, at their simplest form, are made up of one or more primitives: points, lines, or triangles which are made up of vertices. In Ninshu Arts project we need to create two renderer classes:

- *Quad renderer*: This renderer is used to render quads consisting of two triangles.
- *Mesh renderer*: This renderer is responsible for rendering meshes loaded from compiled mesh objects[10] (CMO). It also includes methods providing mesh animation.

[10] Compiled Mesh Object (CMO) is a file format used by Visual Studio graphics content pipeline, as explained in Direct3D Rendering Cookbook [14], chapter 3.

There is a `RenderBase.cs` file which contains abstract class for all renderer classes. We will analyze `QuadRenderer.cs` and `MeshRenderer.cs` classes corresponding to the two mentioned renderer classes later.

3.1.4. Implementing the Rendering Engine

Shaders

Let us have a look at shaders first. We will start with our HLSL shader code now. As we mentioned earlier, we will only need vertex shader and pixel shader for the purposes of our project. There are three HLSL shader code files in our solution and they are `Common.hlsl`, `VS.hlsl` and `BlinnPhongPS.hlsl`.

`Common.hlsl` file contains declarations of structures describing input for vertex shader and pixel shader; and constant buffers, using which we pass data to shaders. In `VS.hlsl`, there is a code for our vertex shader. Lastly, `BlinnPhongPS.hlsl` is a file containing our pixel shader.

Our vertex shader is fairly simple, so we will not be breaking down the code here. It has `VSMain` method which simply transforms UV coordinates and vertex positions into world space. Vertex shader file also includes `SkinVertex` method which we will analyze later when we explain character animation and vertex skinning. Our pixel shader is nothing complicated either. In its `PSMain` method it simply implements texture sampling and computes lighting using *Blinn-Phong light model*[11]. For those readers, who are interested in the code, please see the source files.

[11] *Blinn-Phong light model* is a modification to the Phong reflection model. For more information, see the book Moderní počítačová grafika [25], chapter 10.5.

Constant Buffers

There are several constant buffers we are using in our Direct3D application. First of them is the *Per Object* constant buffer. As the name suggests, it contains data common for all rendered objects. Namely, we are talking about `WorldViewProjection` matrix, `World` matrix needed for lighting calculation in world space and `InverseWorld` matrix used for transformation of normal vectors into world space.

Then there is the *Per Frame* constant buffer which is updated once per frame. This buffer includes the position of camera and lighting settings. We only use directional light in our scene, so we need to remember direction and color of light.

Next we have the *Per Material* constant buffer that holds the material configuration for the currently rendered object. The configuration contains ambient, diffuse, specular and emissive component of the material. Also it includes UV transformation matrix.

Lastly, there is the *Per Armature* constant buffer which holds skin matrices for the bones of animated character. Currently in our project, Per Armature buffer is set to contain maximum of 80 bones. Normally the number would go much higher, but we had to make some compromises in order to make our Rendering Engine to work on WP8 devices.

One might wonder why do we have this many constant buffers. Why not to just use one buffer and send all the data to shaders through that one buffer. The answer is simple - efficiency. We only want to update the data in shaders as least often as possible to minimize the traffic. It is essential to think of optimization where we can because we have to keep in mind that some smartphone devices do not have powerful CPUs and GPUs. It is also a good practice to group things that belong together and send them at the same time. For example, while rendering objects, we would keep the objects with the same material together and render them at the same time simply because this way we do not have to resend the material data in Per Material constant buffer.

In `ConstantBuffers.cs` file we will find classes representing each described constant buffer. We created these classes to make it easier to save the data to buffers and work with them. Here is the snippet from the code:

```
public struct DirectionalLight
{
     public SharpDX.Color4 Color;
     public SharpDX.Vector3 Direction;
     float _padding0;
}
```

And here is the corresponding declaration of constant buffer in HLSL code, Common.hlsl file:

```
struct DirectionalLight
{
    float4 Color;
    float3 Direction;
};

cbuffer PerFrame: register (b1)
{
    DirectionalLight Light;
    float3 CameraPosition;
};
```

Quad Renderer

Our quad renderer is implemented in `QuadRenderer.cs` class. It inherits from `RendererBase` abstract class, so it has to implement `CreateDeviceDependentResources()` method where we create needed resources for our quad renderer. And then it has to implement `DoRender()` method to actually render the quad.

The `CreateDeviceDependentResources()` method does the following. First, it retrieves the instance of the Direct3D device. Then it loads the texture for the quad, if it has any:

```
string file = … ; // path to the texture file
LoadTexture.LoadFromFile(DeviceManager,file,out textureView);
```

Note the `LoadFromFile()` method from `LoadTexture` class. `LoadTexture` is a static class implementing a few methods to load textures in

different formats. There are methods to load textures from bitmaps as well as DDS. We will not go into the detail here. For those interested, please see the source code.

After the texture is loaded we create a sampler state object which controls behavior of texture sampler. Then we simply create an array and fill it with vertices:

```
vertices = new[] {
new Vertex(new Vector3(-4f, 4f, 4f), Vector3.UnitZ),
     new Vertex(new Vector3(-4f, 4f, -4f), Vector3.UnitZ),
     new Vertex(new Vector3(-4f, 0f, -4f), Vector3.UnitZ),
     new Vertex(new Vector3(-4f, 0f, 4f), Vector3.UnitZ),
};
```

For example, here we have an array with four vertices. Each vertex is represented by the Vertex.cs class which simply describes the structure of the vertex that we are sending into a vertex buffer. There are many constructors for Vertex class. Here in the snippet we used a constructor with position and normal vector parameters.

And finally when all is prepared, we create a vertex buffer out of vertices array, vertex buffer binding object and index buffer:

```
vertexBuffer = ToDispose(Buffer.Create(device,
BindFlags.VertexBuffer, vertices));
quadBinding = new VertexBufferBinding(vertexBuffer,
Utilities.SizeOf<Vertex>(), 0);


  // v0    v1
  // |-----|
  // | \ A |
  // | B \ |
  // |-----|
  // v3    v2
indexBuffer = ToDispose(Buffer.Create(device,
BindFlags.IndexBuffer, new uint[] {
                       0, 1, 2, // A
                       2, 3, 0  // B
                 }));
```

You can see the use of `SharpDX.Component.ToDispose()`[12].

[12] `ToDispose` method allows us to create an object instance without having to explicitly dispose of the instance; any objects registered within the `ToDispose` method will be automatically released upon disposal of our SharpDX, as explained in [14], chapter 2.

Component instance.

And with this we are done with `CreateDeviceDependentResources()` method.

Our quad renderer also needs to implement `DoRender()` method. It is quite simple. Here is how it is done:

```
context.PixelShader.SetShaderResource(0, textureView);
context.PixelShader.SetSampler(0, samplerState);
context.InputAssembler.SetVertexBuffers(0, quadBinding);
context.InputAssembler.SetIndexBuffer(indexBuffer,
Format.R32_UInt, 0);
context.InputAssembler.PrimitiveTopology =
SharpDX.Direct3D.PrimitiveTopology.TriangleList;

// Draw the 6 vertices (2 triangles) using the vertex indices
context.DrawIndexed(6, 0, 0);
```

As you can see, it is nothing complicated. To render vertices saved in a vertex buffer you only need to set shader resources and sampler for pixel shader. Then you pass vertices from buffer to input assembler, set index buffer. Before drawing anything, we need to set the primitive topology to triangle list since we need to draw triangles. And eventually, we call `DrawIndexed()` method on the context which draws triangles using the vertex indices saved in index buffer.

Meshes

Now we will take a look at how we will get a model/mesh that artist provided us with into our rendering engine. Usually modeler would hand a model to us in a Autodesk FBX format [26] (or some other format like DAE or OBJ; FBX is widely used though). Visual Studio graphics content pipeline is able to compile an FBX model into a compiled mesh object (.CMO) file which we will be working with.

Another question is how we will represent the data from CMO file in our rendering engine. For this purpose we create a `Mesh.cs` class in Common library. It will represent a single mesh within a CMO file. `Mesh` class also implements methods for extracting and parsing the data from a CMO file. We will not analyze the exact

structure of the CMO file here[13]. It is just a technical thing with no deep meaning. Basically, the Load() method has a BinaryReader and it reads how many materials there are in a CMO file and then it reads the exact count of materials and then it repeats the same process for other things (vertices, indices, bones, etc.). `Mesh` object contains submeshes from which the mesh is comprised of, materials for each submesh, list of vertices and vertex indices, and finally animation data such as list of bones, their names and keyframes of the animation.

Character Animation Basics

There are several ways to animate a character. A current standard technique for character animation is called *skeletal animation*. We will only explain the basics of skeletal animation here[14].

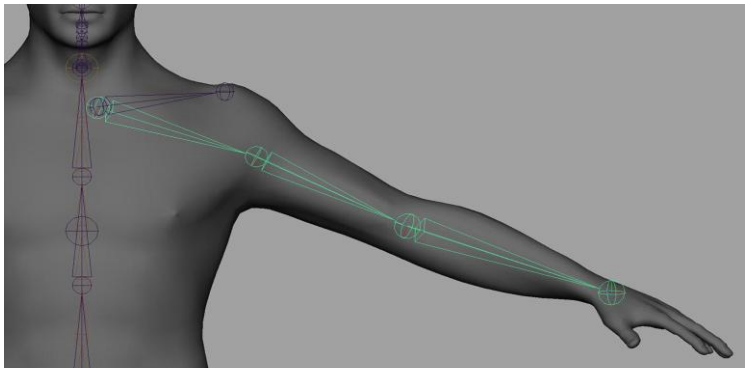
The key component of *vertex skinning* or *skinning* is the hierarchy of pose and movement that is produced by a bone structure or skeleton within a mesh (also known as an *armature*). As basic anatomy teaches us, a skeleton provides a mechanism for transmitting muscular forces. It is a collection of bones, each connected to another.

We apply the same concept to the armature of a mesh. We have a root bone, and each subsequent bone is parented by the root bone or another bone that ultimately resolves its parentage to this root bone. In this manner, if we move the root bone, the whole body moves with it. But for example, if we move the shoulder, then only the arm moves with it.

---

[13] Readers can see the source code of Mesh class for the details about the internal structure of a CMO file or visit DirectX Tool Kit homepage [16] where they can find a CMO file parser written in C++.

[14] Skeletal animation is not a new technique. There are a lot of books and articles explaining how it works in detail. For more information on skeletal animation we refer readers to [17], chapter 25.



Picture 5.: Connected bones

The initial position or pose is referred to as the *bind pose* or *rest pose*, and it represents the default starting transformation of each bone at the time the mesh was bound to the armature or rigged. Simply having an armature in place is not enough for it to apply forces upon the skin. We must bind the mesh to the armature and specify how each of the bones will influence the vertices. By applying weights, known as *bone-weights* or *blend-weights*, to the vertex of each of the influencing bones, our armature will be able to influence the placement of vertices.

Our implementation supports up to four bone influences per vertex. This is the maximum number supported by the CMO file format produced by Visual Studio. As already mentioned, the key component of skinning is the hierarchy of transformations that is produced from the bones of an armature. These transformations or skin matrices are implemented using the 4x4 affine transformation matrix. Nevertheless, instead of transforming from the local object space to world space as we have done previously, we will be applying these transformations in bone space.

The transformation to bone space involves calculating the current translation and rotation for each bone as well as its scale against its parent bone. To bring the transform into bone space, we apply the inverse bind pose matrix of the bone. The result skin matrix for each bone is what we are storing in the `PerArmature` constant buffer, and it is the matrix that we refer to within the vertex shader.

Inside the vertex shader we blend the bone influences together based upon their weight. With the armature's skin matrices in place, we can apply the vertex skinning within the vertex shader (you can see this in the code of `VS.hlsl` file).

```
float4x4 skinTransform = Bones[bones.x] * weights.x +
Bones[bones.y] * weights.y +
Bones[bones.z] * weights.z +
Bones[bones.w] * weights.w;
// Apply skinning to vertex and normal
position = mul(position, skinTransform);
normal = mul(normal, (float3x3)skinTransform);
```

For each of the bone influences, the skin transform matrix is retrieved and multiplied by the bone-weight. These four matrices are then added together to determine the final skin transform for this vertex. If the first bone has a weight of zero, we are skipping this process. This is important because we are using this vertex shader for meshes whether or not they have any bones.

Mesh Renderer

We will analyze the mesh renderer now. It is represented in `MeshRenderer.cs` class. Just as `QuadRenderer` class, it descends from `RendererBase` class which means it needs to override `CreateDeviceDependentResources()` method and `DoRender()` method.

These are the properties that deserve pointing out:

```
// Create a timer
Stopwatch clock = new Stopwatch();

// The currently active animation (allows nulls)
public Mesh.Animation? CurrentAnimation { get; set; }

// Play once or loop the animation?
public bool PlayOnce { get; set; }
```

The `clock` object provides us with stopwatch use to control the animation. `CurrentAnimation` remembers the animation that is currently being played. And `PlayOnce` boolean variable tells us if we want the animation to repeat or play once.

The `CreateDeviceDependentResources()` method works in a similar fashion as in QuadRenderer - it initialized resources for our mesh renderer. The code

is straightforward. It takes Mesh object corresponding to the current instance of the MeshRenderer and initializes vertex buffer and index buffer. After that it loads textures, if materials have any. Finally, we create a sampler state.

Now let us have a look at `DoRender()` method which not only renders a mesh but also animates it, if animation data are present. If they are, we first retrieve local transformation matrix for each bone. After that we load the key-frames of an animation. Animation data in key-frames contain a transformation matrix for particular bone which is identified by its index and time when the transformation should be done:

```
public struct Keyframe : IDataSerializable
{
     public uint BoneIndex;
     public float Time;
     public Matrix Transform;
};
```

We iterate these frames up to the current frame time and replace the existing bone's local transform with that of the key-frame:

```
for (i = 0; i < CurrentAnimation.Value.Keyframes.Count; i++)
{
// Retrieve current key-frame
     var frame = CurrentAnimation.Value.Keyframes[i];

     // If the current frame is not in the future
     if (frame.Time <= time)
     {
                // Keep track of last key-frame for bone
           lastKeyForBones[frame.BoneIndex] = frame;
           // Retrieve transform from current key-frame
           bones[frame.BoneIndex] = frame.Transform;
           }
          // Frame is in the future, check if we should
interpolate
     else
     {
// Interpolate a bone’s key frame
...
     }
}
```

We then interpolate each bone's current key-frame with its next future key-frame based on the amount of time that has lapsed between the two frames. To produce a smoother animation without artefacts introduced from the linear interpolation of rotation matrices, we decompose our transformation matrices for the

two key-frames into its translation, rotation, and scale components. The translation is stored within `Vector3`, the uniform scale within a `float` variable, and finally the rotation component is now stored within a *quaternion*:

```
Vector3 t1, t2;   // Translation
Quaternion q1, q2;// Rotation
float s1, s2;     // Scale
// Decompose the previous key-frame's transform
prevFrame.Transform.DecomposeUniformScale(out s1,
out q1, out t1);
// Decompose the current key-frame's transform
frame.Transform.DecomposeUniformScale(out  s2,  out  q2,  out
t2);
```

We *linearly interpolate* (Lerp) between the two scale floats (`s1` and `s2`) and the two translation vectors (`t1` and `t2`). Then we perform *spherical linear interpolation* (Slerp) between the two quaternions (`q1` and `q2`). The matrix is then reconstructed from the interpolated scale, rotation, and translation. And to ensure that we end up with the correct result, we multiply the matrices in the following order: first apply scale, then rotate, and finally translate.

```
// Perform interpolation and reconstitute matrix
bones[frame.BoneIndex] =
 Matrix.Scaling(MathUtil.Lerp(s1, s2, amount)) *
 Matrix.RotationQuaternion(Quaternion.Slerp(q1,  q2,  amount))
*
 Matrix.Translation(Vector3.Lerp(t1, t2, amount));
```

The rest of the `DoRender()` method here is quite straight forward and similar to the `QuadRenderer.DoRender()`. We update the constant buffers, group the sub-meshes by material to render them together and call the `context.DrawIndexed()`.

Putting it all together

Now we have all the needed ingredients. We just have to put it all together. We are going to do it in `D3DApp.cs` class which inherits from `D3DApplicationWinRT/D3DApplicationWP` classes. There are two important methods. They are `CreateDeviceDependentResources()` and `Render()` methods. This is how our `CreateDeviceDependentResources()` method works:

Firstly we need to load the vertex shader file:

```
#if WINDOWS_APP
                StorageFile sampleFile = await
StorageFile.GetFileFromApplicationUriAsync(new
Uri(@"ms-appx:///Shaders/VS_5_0.fxo"));
#endif
#if WINDOWS_PHONE_APP
                StorageFile sampleFile = await
StorageFile.GetFileFromApplicationUriAsync(new
Uri(@"ms-appx:///Shaders/VS_4_0_level_9_3.fxo"));
#endif
```

Note the `#if WINDOWS_APP` and `#if WINDOWS_PHONE_APP` directives. These tell the compiler to compile the code between `#if` and `#endif`, if the condition is met. Specifically, the first directive is satisfied when we are compiling the code for Win8 app, the other directive is for WP8 applications. This is an elegant way to have multiple versions of code snippet in one source file.

After we load vertex shader file, we convert it to shader bytecode which is then used to create a vertex shader object. Then we need to create a vertex layout. Vertex layout defines a structure of a vertex that is sent to vertex shader, so that vertex shader knows how to interpret the received data. Vertex layout might look as follows:

```
vertexLayout = ToDispose(new InputLayout(device,

bytecode.GetPart(ShaderBytecodePart.InputSignatureBlob).Data,
      new[]
      {
                new       InputElement("SV_Position",       0,
Format.R32G32B32_Float,
0, 0),
                new          InputElement("NORMAL",         0,
Format.R32G32B32_Float, 12, 0),
                new          InputElement("COLOR",          0,
Format.R8G8B8A8_UNorm, 24, 0),
                new         InputElement("TEXCOORD",        0,
Format.R32G32_Float, 28, 0),
           #if WINDOWS_APP
                new
InputElement("BLENDINDICES",0,Format.R32G32B32A32_UInt,
36, 0),
#endif
#if WINDOWS_PHONE_APP
                new             InputElement("BLENDINDICES",0,
Format.R32G32B32A32_Float,
36, 0),
#endif
                new       InputElement("BLENDWEIGHT",       0,
Format.R32G32B32A32_Float,
52, 0),
      }));
```

When we are done with vertex shader, we load the pixel shader file and we create pixel shader object from the loaded shader bytecode in a similar way. After that we create constant buffer objects. Then we could create and configure a depth buffer to discard pixels that are further than current pixel. Lastly we create renderers and initialize them:

```
bottomQuadRenderer = ToDispose(new QuadRenderer(1,
CubeMapFace.Bottom));
bottomQuadRenderer.Initialize(this);
frontQuadRenderer = ToDispose(new QuadRenderer(1,
CubeMapFace.Front));
frontQuadRenderer.Initialize(this);
backQuadRenderer = ToDispose(new QuadRenderer(1,
CubeMapFace.Back));
backQuadRenderer.Initialize(this);
```

As for the `Render()` method of `D3DApp` class, it goes quite straight forward. At the start of the method we call commands to set depth buffer and render target view. After that we compute MVP matrix, get the camera position and directional light settings. With these information we can fill the corresponding constant buffers. After that we iterate through all the objects in the scene, each of which has its own renderer object, and simply call `Render()` on the renderer.

It is important to note that before we iterate through objects in the scene and call their `Render()` methods, we need to group the objects with the same material together. It is to optimize the rendering process, so that we do not have to resend the data in Per Material constant buffer.

3.1.5. Changes Regarding Transitioning to Windows Store App

Integrating Direct3D Application to XAML

Now we have to stop for a second and think for a while how we are going to connect our rendering engine to the game presentation layer. We have several options of how to do the game presentation layer for a Windows Universal application. For example, it is possible to write the graphical user interface (GUI) in JavaScript and HTML. However, we are going with Extensible Application Markup Language (XAML) [15].

We will render to an XAML `SwapChainPanel`. This panel allows us to efficiently render using Direct2D/Direct3D within an XAML Windows Store app. By integrating Direct3D into XAML we are able to use XAML to create flexible and dynamic UIs for our DirectX application. The `SwapChainPanel` XAML element is new to Windows 8.1.

In order to be able to integrate our DirectX to XAML, we have to create a new `D3DAppSwapChainPanelTarget` class which inherits from our `D3DApplicationWinRT` or `D3DApplicationWP`, depending on whether we are in `Common.WinRT` or `Common.WP` library. Our new class accepts a `SwapChainPanel` instance, attaches a handler to its `SizeChanged` and `CompositionScaleChanged` events, and retrieves a reference to the `ISwapChainPanelNative` interface.

It is important to note that the `SwapChainPanel` XAML control descends from `Windows.UI.Xaml.Controls.Grid` and therefore, supports layouts for child controls and can be added as a child to other controls. We will be using this fact when we implement GUI for battles, but more on that later.

By retrieving the `ISwapChainPanelNative` interface from the `SwapChainPanel` instance, we have connected our new swap chain to the panel through the `ISwapChainPanelNative.SwapChain` property (natively this is done through the `ISwapChainPanelNative.SetSwapChain` method). The panel then takes care of associating the swap chain with the appropriate area on the screen[15].

Loading and Compiling Resources Asynchronously

Within Windows Store applications, it is desirable to keep your application as responsive as possible at all times. Instead of showing a static splash screen for the

[15] Other stuff regarding transitioning to Windows Store application is quite insignificant and not important for us at the moment. If the reader is interested in more details or other ways of integrating Direct3D application to XAML, we refer the reader to Direct3D Rendering Cookbook [14], Chapter 11.

duration of compiling shaders and loading resources. We will accomplish this by initializing our renderers and resources using the `async/await` keywords.

Within the `D3DApp` class, we update the signature of `CreateDeviceDependentResources()` method to include the async keyword as shown in the following snippet:

```
protected async override void
CreateDeviceDependentResources(DeviceManager deviceManager)
{
...
}
```

Now when we compile our shaders, load our meshes or initialize our renderer instances, we can do something like the following snippet:

```
// Compile shader, the event caller will continue executing
using (var bytecode = await
HLSLCompiler.CompileFromFileAsync(@"Shaders\VS.hlsl",
"VSMain", "vs_5_0"))
{ ... }

// Load mesh
var meshes = await Mesh.LoadFromFileAsync("Character.cmo");

// Other CPU-bound work
await Task.Run(() =>
{
... (e.g. initialize renderers)
});
```

This not an essential thing, it is just something we should do to improve user experience for players. It will definitely make the game look more responsive. When we combine this with a simple XAML animation, we get a dynamic loading screen rather than unappealing static splash screen.

3.1.6. Compromises for Multi-platformity

Shader Model Restrictions

There are a few things we have to consider regarding the multi-platformity feature of our project. Firstly, we have to address the problem that WP8 operating system does not support feature level `11_0`. Normally desktop PCs, notebooks and tablets running on Win8 OS have no problem with this since Win8 OS supports the

newest feature levels of DirectX API, including feature level `11_1`. Unfortunately, WP8 OS graphic capabilities are a little bit limited. WP8 devices can only access feature level `9_3`.

What exactly does it mean for us? The most significant adjustment we have to make concerns shader models. DirectX feature level 9_3 only allows us to use shader model 2 (SM2) which is a great jump back compared to SM5. SM2 for WP8 operating system provides us with shader profile `vs_4_0_level_9_3` for vertex shaders and `ps_4_0_level_9_3` for pixel shaders. The restriction that affects us the most is that we can only use 256 constant float registers in vertex shaders. One constant float register can hold a single `float4` component.

Let us think about how we are sending bone matrices through a constant buffer to vertex shader. We represent a bone matrix with float4x4 component (4x4 float matrix). If we only have 256 registers, we would only be able to store 64 bone matrices in a vertex shader at a time. The real number of stored bones would be even smaller because we must not forget that we have to store other data in vertex shader apart from bone matrices (materials, light settings, etc.). So we would get to storing about 43 bones which might not be enough in some cases. Most models come with 60 bones and more. And note that with SM5 we would be able to store maximum of 1024 bones. So you can see that the difference between SM5 and SM2 might be significant. Of course we would not be using all 1024 bones in our models. It is also recommended to keep the bone count to minimum to reduce the memory bandwidth.

So how are we going to solve the problem? There are several options. For example, one would be CPU skinning. That means compute the bone transformations on CPU instead of GPU. One would simply do the bone transformation on vertex position before sending vertex into a vertex shader. I personally tested this method and results were not good at all. Even on a Intel Core i7-4770 processor (4 cores, 3.4-3.9GHz) with nVidia GeForce GTX 980 graphic card the engine would run with 24 frame per second (FPS) at most. You can imagine the results on a low power smartphone.

Another way to address this problem would be not using float4x4 matrix, but using three float4 vectors instead. If we think about how the transformation matrix looks like, we notice that the last row is always zeros and one at the end. So we can save some space by sending three float4 vectors which are three rows of the original float4x4 matrix and send them separately. We then assemble the original transformation matrix in a vertex shader and use it for bone transformation. If we use this method, we get maximum of 80 bones which is sufficient for most cases.

Compiling Shaders

As we already mentioned many times, we have to take into account even the low budget smartphones with low performance CPUs and GPUs. One of the issues while testing the engine was a long loading time into a 3D scene on some WP8 devices. To optimize and reduce the loading time as much as possible we might consider precompiling our shaders and then only load them in the initialization stage.

Originally, we had `HLSLCompiler.cs` class and `HLSLFileIncludeHandler.cs` class which provide us with the ability to load a shader file and compile it to a shader bytecode. But for optimization purposes we precompiled our shaders using `fxc.exe` utility which is available in DirectX SDK, for example. Now, the engine only has to load the precompiled vertex shader and pixel shader and use them.

3.2. Implementing the Communication Framework

The communication framework is responsible for any device to device communication. As we already mentioned before, one of the primary objective of this project is to implement a communication framework capable of providing the ability to exchange messages between two devices of different types, e.g. desktop PC to tablet, notebook to phone, etc.

For the purposes of this project we picked bluetooth technology since bluetooth is widely present on virtually all smartphones and tablet devices. Most of notebooks nowadays have bluetooth adapter. And some desktop PCs come with

bluetooth adapter integrated on motherboard; and if not, purchasing a USB bluetooth adapter is cheap and easy.

We remind the readers that we picked bluetooth over wi-fi technology because WP8 operating system does not support Wi-Fi Direct communication, as we pointed out in subchapter 2.4. It was bluetooth API and Wi-Fi Direct API that offered a possibility to have a P2P communication which is exactly what we wanted - ability to exchange messages with no need for wi-fi router and local network, or connection to internet.

Although, let us keep in mind that we should design our communication framework in a way so that it would be easy to extend. For example, if Microsoft would provide us with an Wi-Fi Direct API working on WP8 operating system, it would only be a matter of implementing a new communication unit into our framework. We have plans to extend communication possibilities in the future (feature 1. in future development list, see subchapter 5.4.). For instance, we would like the game to support network communication or server-client communication with a separate server host running on a distant server. That is why we have to think about extensibility right at the start.

3.2.1. Bluetooth Communication Unit

We will use *Bluetooth Rfcomm* Windows Runtime API[16] which is accessible from both Win8 and WP8 operating systems. Data reading and writing is designed to take advantage of established data stream patterns and objects in `Windows.Storage.Streams`.

[16] There are other APIs to implement bluetooth communication for Windows 8.1. If the readers are interested, we refer them to the MSDN article Supporting Bluetooth Devices [18].

We create a new distinct `Communications` library which will represent our communication framework. In this library we have a `Bluetooth.cs` class that encapsulates everything about the bluetooth communication unit.

In `Bluetooth.cs` file we first create an abstract Bluetooth class:

```
public abstract class Bluetooth
{
     public abstract Task Send(string message);
     public abstract Task<string> Receive();
public abstract void Disconnect();
}
```

This abstract class represents a bluetooth unit for exchanging messages over bluetooth. It defines three methods. The `Send()` method sends a message to another bluetooth unit on the other side of the communication line. The `Receive()` methods waits and listens for incoming messages from the other unit on the line. And `Disconnect()` method terminates the communication line and cleans up.

One might think that both communication participants are equal because we are working with P2P type of communication. However, it is not like that. For our bluetooth communication line, one participant is considered server and the other client. The server participant advertises the service. The client participant then searches for advertised services and connects to the server. Note that both participant devices need to be paired in order to be able to connect to each other. The communication line is then established and from this time on, both participants behave as equal. They can both send the message to the one on the other side of the line and listen for the incoming messages. Once the communication is established, we would not tell them apart.

As we need to represent a server participant and a client participant, we will create a bluetooth server unit and bluetooth client unit, both descending from our `Bluetooth` abstract class.

Bluetooth Server

First we create a BluetoothServer.cs class to represent our bluetooth server participant. It is will contain these properties:

```
public class BluetoothServer : Bluetooth
 {
public StreamSocket socket;
          public DataWriter writer;
          public DataReader reader;
          public RfcommServiceProvider rfcommProvider;
          public StreamSocketListener socketListener;
...
}
```

Here we can see the `StreamSocket` object representing a stream over which the data are transmitted. Then we have a `DataWriter` and `DataReader` objects that are associated with the `StreamSocket` for writing the data or reading the data over the stream. The `RfcommServiceProvider` object is used for advertising the service using a `StreamSocketListener`.

First of all, we need to initialize the bluetooth server instance:

```
public async void InitializeRfcommServer()
{
      bluetoothServer.rfcommProvider = await
      RfcommServiceProvider.CreateAsync(RfcommServiceId.FromUuid(
      RfcommChatServiceUuid));

      bluetoothServer.socketListener = new StreamSocketListener();
      bluetoothServer.socketListener.ConnectionReceived +=
      OnConnectionReceived;

      await bluetoothServer.socketListener.BindServiceNameAsync(
      bluetoothServer.rfcommProvider.ServiceId.AsString(),
      SocketProtectionLevel);

      // Set the SDP attributes and start Bluetooth advertising
      InitializeServiceSdpAttributes(bluetoothServer.rfcommProvider)
      bluetoothServer.rfcommProvider.StartAdvertising(
      bluetoothServer.socketListener);
}
```

As we can see from the code snippet, we first need to create a `RfcommServiceProvider` instance. After that we create an instance of socket listener on which we a `ConnectionReceived` event handler. Then we just need to set Service Discovery Protocol (SDP) attributes and start advertising our service and listening for incoming connections. SDP attributes set this way:

```
private void InitializeServiceSdpAttributes(RfcommServiceProvider
rfcommProvider)
{
      var sdpWriter = new DataWriter();

      sdpWriter.WriteByte(SdpServiceNameAttributeType);

      sdpWriter.WriteByte((byte)SdpServiceName.Length);

      sdpWriter.UnicodeEncoding =
      Windows.Storage.Streams.UnicodeEncoding.Utf8;
      sdpWriter.WriteString(SdpServiceName);

rfcommProvider.SdpRawAttributes.Add(SdpServiceNameAttributeId,
      sdpWriter.DetachBuffer());
}
```

Basically, what we need to do is to set up a DataWriter and then encode a service name attribute type, followed by the length of the name and the name value. Finally, we set the SDP attribute on Rfcomm service provider.

For the sake of completeness, here is how we declare service name attribute type:

```
public const byte SdpServiceNameAttributeType = (4 << 3) | 5;
```

SDP attribute type is encoded in SDP attribute as follows. The Attribute Type size in the least significant 3 bits and the SDP Attribute Type value in the most significant 5 bits.

Now when a client participant connects to the service we are advertising, `ConnectionReceived` event is fired followed by its event handler `OnConnectionReceived()` method where we just call server's connect method and start a loop for receiving messages:

```
public void Connect(StreamSocket streamSocket)
{
     socketListener.Dispose();
     socketListener = null;

     socket = streamSocket;

     writer = new DataWriter(socket.OutputStream);
     reader = new DataReader(socket.InputStream);
 }
```

As you can see, the BluetoothServer.Connect() method is quite simple. First, i disposes of the listener which is no longer needed. Then it saves the reference for StreamSocket object that was obtained when `ConnectionReceived` event was fired. Lastly, it creates a `DataWriter` and a `DataReader` instances.

Bluetooth Client

Now we create a BluetoothClient.cs class that will represent a client participant:

```
public class BluetoothClient : Bluetooth
{
      public StreamSocket socket;
      public DataWriter writer;
      public DataReader reader;
      public RfcommDeviceService rfcommProvider;
      public DeviceInformationCollection chatServiceInfoCollection;
}
```

As you can see here from the properties of the class, it has similar fields as `BluetoothServer` class. The only thing that was added here is `DeviceInformationCollection` object that is used to store the data of found services later.

The first thing we must do for bluetooth client is to find nearby services that are advertising:

```
public async Task<int> FindAll() {

      chatServiceInfoCollection = await
            DeviceInformation.FindAllAsync(
            RfcommDeviceService.GetDeviceSelector(
            RfcommServiceId.FromUuid(RfcommChatServiceUuid)));

      if (chatServiceInfoCollection != null)
            return chatServiceInfoCollection.Count;
      else
            return 0;
}
```

In this method we call `DeviceInformation.FindAllAsync()` method which searches for all nearby advertised services. Remember that it will only find services running on those server devices that are paired with our client device. All

found services will be stored in our `chatServiceInfoCollection`. The method then returns the found item counts.

Once we have a collection of available services, we just need to pick the service we want to connect to and initialize the connection to the server:

```
var attributes = await
      bluetoothClient.rfcommProvider.GetSdpRawAttributesAsync();
var attributeReader = DataReader.FromBuffer(
      attributes[SdpServiceNameAttributeId]);
var attributeType = attributeReader.ReadByte();
var serviceNameLength = attributeReader.ReadByte();

// The Service Name attribute requires UTF-8 encoding.
attributeReader.UnicodeEncoding = UnicodeEncoding.Utf8;
var name = attributeReader.ReadString((uint)serviceNameLength);
```

This code snippet simply shows that we need to read the SDP attribute on the Rfcomm service that we picked earlier. First it reads attribute type, after that it read the name length and the name value.

All that is left to do is to connect to the server using the `BluetoothClient.Connect()` method:

```
public async Task Connect()
{
      socket = new StreamSocket();

      await socket.ConnectAsync(rfcommProvider.ConnectionHostName,
      rfcommProvider.ConnectionServiceName);

      writer = new DataWriter(socket.OutputStream);
      reader = new DataReader(socket.InputStream);
}
```

In this simple method we create a new instance of StreamSocket and call `socket.ConnectAsync()` method to establish connection to the bluetooth server. Finally, we initialize the `DataWriter` and `DataReader` instances to write and read messages.

Once the connection between the server participant and the client participant is established, they can exchange messages using the `Send()` and `Receive()` methods. The implementation of these methods is the same for both participants. Here is the first one:

```
public override async Task Send(string message)
{
     writer.WriteUInt32((uint)message.Length);
     writer.WriteString(message);

     await writer.StoreAsync();
}
```

Again, it is not a very complicated thing. Using the `DataWriter writer` instance we first need to encode the length of the message, so the recipient of the message knows how much data they will need to read. After that we write the content of the message and close the message with `writer.StoreAsync()` command.

Lastly, this is how we receive messages:

```
public override async Task<string> Receive()
{
     uint size = await reader.LoadAsync(sizeof(uint));
     if (size < sizeof(uint))
     {
                 return null;
     }
     uint stringLength = reader.ReadUInt32();

     int actualStringLength = await
     reader.LoadAsync(stringLength);
     if (actualStringLength < stringLength)
     {
           return null;
     }

     return reader.ReadString(stringLength);
}
```

Before we start to read the message, we need to check if the underlying socket was closed before we were able to read the whole message. To be sure that the communication was not lost during the transmission, we first read the length that the message is supposed to have. After that we find out the length of the following message data, which we called `actualStringLength` in the code snippet. Then we compare the actual string length with the alleged length. If both lengths are equal, everything is ok and we can read the message content.

3.2.2. Communication Model in Battles

As we mentioned earlier, the participants in bluetooth communication are not equal. One is considered a server and the other is considered a client. Once the pairing and connecting process is done, they start to behave like equal participants in P2P communication and it no longer matters who is the server and who the client. And if we think about how we want conduct battles in Ninshu Arts - two equal players battling each other in P2P multiplayer battle with no server unit, one might think it is convenient for us to keep communication participants equal. But in fact, we do not want that.

We are designing this communication framework with emphasis on portability and extensibility. First, the communication framework as we implemented it could be ported to another game or another program and reused with minimal effort concerning integration. Second, we have to keep in mind that our game as any other game at the market today has to evolve and develop new features in order to stay appealing to players. That could mean, for example, implementing a server application running on a distant server where players could connect and have battles online. In such situation, the communication would be of server-client type where we would have a server unit and two client units - players. The battle itself would happen on the server where all the computing units and data from the battle would be stored. Players would only send their actions to the server and the battle managing application on the server would evaluate the actions and send back results to the client players.

With that said, we will not be keeping our bluetooth communications participants as equals. On the contrary, the game application that posed as bluetooth server will remain a server game application and the other one will be a client game application. That means in a P2P multiplayer battle, the server game application will take on a role of a distant game server and will host a combat managing unit. During a turn in combat, both players send their action to the combat manager. But in our battle communication model that means that only the client player sends the action to the server player while server player “sends the action to himself”, since the combat manager unit is hosted in the server game application.

Now if we would be implementing a new feature for Ninshu Arts with the server application on a distant server and online battles, we would simply need to relocate the combat manager unit to the application running on a distant server and mark both players as client applications.

3.3. Game Logic and Features

In this subchapter we will look into implementing the game logic and game features. That means what actually makes this game a game. So far we were only concerning ourselves with technical questions like how to render a character model, how to make it move in the scene or how to send a message to another game application. This subchapter will be more about turning the things from the game design chapter into something real. We will be dealing with the game content and how to store it, how to load it and how to use it.

To represent the game content and features and everything related to these two, we created a new separate library `GameContent` which is portable and reusable if needed.

3.3.1. Player

First of all we have to think of a way how to represent a player. The player entity is something that will last for a long time. It will contain all the shinobi's attributes and it needs to remember the character's progress. Because all of these things that we just listed need to be remembered even when the game is turned off, we will need to save the player entity to an external file which we will be able to reuse when the player returns to the game again.

Let us take a look at the first class of the `GameContent` library, the `Player.cs` class. This class represents the player entity in the game. First, you will find `string Name`, `long Experience` and `int Level` properties here. As the names suggest, these represent player's name, gained experience and level. Next we will find the properties for every player's stats - `Stamina`, `Speed`, `Attack`, `Defense` and `Resistance`. They are all represented by `int` variable. Then there

are `int` properties for each element. These values describe how proficient the character is in using jutsus of a concrete element. It means if the player uses a lot of fire techniques in battles, (s)he will have a high `Fire` stat. Lastly, the Player class has a collection of jutsus that (s)he possesses - `List<Jutsu> Jutsus`. It is a list of `Jutsu` objects which we will describe later.

Now let us talk about saving the player's data to an external file. Windows Store applications has its own data storage. This feature is provided by `Windows.Storage.ApplicationData`[17] class. We will be using this storage to store our save data:

```
public async Task Save()
{
      StorageFolder localFolder =
            ApplicationData.Current.LocalFolder;

      StorageFile file = await localFolder.CreateFileAsync(
            Constants.PlayerSaveFilename,
            CreationCollisionOption.ReplaceExisting);

      string json = JsonConvert.SerializeObject(this);

      await FileIO.WriteTextAsync(file, json);
}
```

In this method we should notice a few things. First, we are accessing `Constants.cs` static class that we created to hold all the constants in the game. There we defined a file name for our save file to keep it unified. It is a good practice not to hardcode values in the code and define constants instead. Second we are using *Json.NET* framework[18]. We are using this framework to convert player object into *Javascript Object Notation*[19] (JSON) string. The procedure is straightforward. We access the local folder of our Windows Store application, then we create a `StorageFile` object to represent the file. Next we convert a `Player` object into a JSON string and finally we write the text into the file.

[17] *ApplicationData* is a class providing access to the application data store [19].
[18] *Newtonsoft Json.NET* is a high-performance JSON framework for .NET [20].
[19] *JSON* is a lightweight data-interchange format [21].

The process of loading a save file and converting a JSON back into the Player object is exact reversed procedure, but instead of serializing the data we deserialize the data. Here is the important line of code from `Load()` method:

```
player = (Player)JsonConvert.DeserializeObject(json,typeof(Player));
```

Only Windows Store application can access its local storage. In fact, it is possible to access it outside the application, but only if you are not on WP8 device. To prevent messing with the data, the save file should be encrypted.

3.3.2. Jutsus and Missions Data

Now we stand before a big question. We have to think of an elegant way to represent and store a large amount of objects of the same type. We have a lot of jutsus and we definitely plan on adding new jutsus in the future. In addition, we have to consider more people working on the game. The game designer and the game programmer roles don't have to be combined in one person (for the development stages see subchapter 5.1.). If that is the case, we should provide the possibility to a person with no skills and knowledge of programming to be able to edit and add the data. Everything that has been said about jutsus applies to mission as well.

We propose a method where the data on a certain jutsu or a mission is stored in an external Extensible Markup Language[20] (XML) file. There will be a separate XML file for each jutsu/mission. We will name these XML files in an definite way and when they are needed, we simply load them into the memory and work with them. XML has the advantage of being easy to read. So it is not very hard even for a non-programmer person to edit them or create a new XML file with the new data. Or for more comfort, the data on all jutsus and missions can be kept on Google Sheets[21] where all the people from the development team can access and edit them. It is not hard to write an exporter which would take the sheet and export XML files from the parsed data. In addition, we encapsulate the XML files in our `GameContent` library, so the data are safely kept inside a DLL file where no one can mess with them.

---

[20] *XML* is a simple and flexible text format derived from SGML (ISO 8879) [22].
[21] *Google Sheets* are Sheet files stored on Google Drive (cloud storage) [23].

We create a `Jutsu.cs` and `Mission.cs` classes to represent a jutsu and mission objects. `Jutsu` class contains all the properties needed to describe a single jutsu such as jutsu's ID, name, element, damage, hit count and times of those hits and more (for all the jutsu's properties read section 2.2.2. in chapter 2).

We create a `Mission` class in a similar fashion. It contains mission's ID, name, description, duration in seconds and experience and stats rewards for completion (for more details see the section 2.3.3. or source code).

Here is the example XML file `Jutsu_4.xml`:

```
<jutsu>
  <id>4</id>
  <name>Atk_Earth_1</name>
  <description>Atk_Earth_1</description>
  <element>Earth</element>
  <rank>D</rank>
  <damage>30</damage>
  <hitcount>2</hitcount>
  <hittimes>
    <time>0.5</time>
    <time>0.8</time>
  </hittimes>
  <cooldown>0</cooldown>
  <effect>none</effect>
  <effectvalue>0</effectvalue>
</jutsu>
```

And here is the Jutsu class that we load XML data into:

```
public class Jutsu
{
     public int ID { get; set; }
     public string Name { get; set; }
     public string Description { get; set; }
     public Element Element { get; set; }
     public Rank Rank { get; set; }
     public int Damage { get; set; }
     public int HitCount { get; set; }
     public List<double> HitTimes { get; set; }
     public int Cooldown { get; set; }
     public CombatEffect Effect { get; set; }
     public int EffectValue { get; set; }
}
```

Now let us see how we read the XML file and save the data into our Jutsu or Mission instance. Here is the snippet from the `GetResourceTextFile(string filename)` method:

```
using (Stream stream = ...)
using (XmlReader reader = XmlReader.Create(stream)) {
   while (reader.Read() {
      if (reader.NodeType == XmlNodeType.Element) {
   if (reader.Name == "name")
            Name = reader.ReadElementContentAsString();
        else if (reader.Name == "description")
           Description = reader.ReadElementContentAsString();
  else if (reader.Name == "damage")
           Damage = reader.ReadElementContentAsInt();
         ...
```

As we can see from the snippet, we are simply using the `XmlReader` class that is using a `Stream` linked to the respective XML file.

3.3.3. Combat Manager

Combat manager is a unit responsible for managing the battle that was mentioned in section 3.2.2. It keeps all the current battle data about both shinobis participating the battle and keeps log about the actions of each turn. Basically, players send their attack and defence actions to the combat manage within each turn and the combat manager then evaluates them to generate result of the turn.r

In this section we implement the `CombatManager.cs` class which represents our combat manager. But before we go into details of `CombatManager` class, we should take a look at `Turn.cs` class first:

```
public class Turn
{
     public Jutsu P1ExecutedJutsu { get; set; }
     public Jutsu P2ExecutedJutsu { get; set; }
     public int P1DamageTaken { get; set; }
     public int P2DamageTaken { get; set; }
...
}
```

It is easy to see from the previous code snippet that `Turn` object saves the information about what justus players used this turn and how many damage each of them took. Now we need to represent a shinobi fighting in a battle:

```
public class Shinobi
{
     public int HP { get; set; }
     public Player Player { get; private set; }
     public int Dazed { get; set; }
     public int Electrified { get; set; }
     public int[] Cooldowns { get; set; }
}
```

We will retain the data about shinobi's health points (HP), reference to a player that the shinobi represents, indicators if the shinobi is dazed or electrified (and if they are then for how long) and cooldowns on shinobi's jutsus.

Now let us analyze the `CombatManager` class:

```
public class CombatManager
{
     public Shinobi Shinobi1 { get; set; }
     public Shinobi Shinobi2 { get; set; }
     public List<Turn> Turns { get; private set; }
     public JsonMessages.ActionJson Attack1 { get; set; }
     public JsonMessages.ActionJson Attack2 { get; set; }
     public JsonMessages.ActionJson Defend1 { get; set; }
     public JsonMessages.ActionJson Defend2 { get; set; }
     ...
}
```

The CombatManager object keeps references to both Shinobi objects. Next it has list of Turn objects where it keeps the log of the data that were executed in previous turns. And then you can see the two attack ActionJson and two defend ActionJson objects. In these four variables the combat manager keeps the data on attack/defend action performed by both players.

```
public class ActionJson
{
     public int JutsuID { get; set; }
     public SwipeHit[] SwipeHits { get; set; }
     public List<string> Animations { get; set; }
}
```

Specifically, it contains the ID of a jutsu that player picked, an array of swipes (left/right swipe or miss) and list of names of animations from which we can reconstruct the whole animation of a combo. Now let us have a look at how the combat manager evaluates turns:

```
public void EvaluateTurn()
{
      Turn turn = new Turn();
      turn.P1ExecutedJutsu = new Jutsu(Attack1.JutsuID);
      turn.P2ExecutedJutsu = new Jutsu(Attack2.JutsuID);
      turn.P1DamageTaken = CalculateDamageTaken(Shinobi1, Shinobi2,
      turn.P1ExecutedJutsu, turn.P2ExecutedJutsu,...);
      turn.P2DamageTaken = CalculateDamageTaken(Shinobi2, Shinobi1,
      turn.P2ExecutedJutsu, turn.P1ExecutedJutsu,...);
      EditCooldowns();
      Shinobi1.HP -= turn.P1DamageTaken;
      Shinobi2.HP -= turn.P2DamageTaken;
      Turns.Add(turn);
}
```

This is only a snippet from the `EvaluateTurn()` method to show the overall process of evaluating the turn. Firstly, a new `Turn` object is created. We remember the jutsus that players picked for this turn. After that we calculate damage taken for both players and we reduce their HP accordingly. Eventually, we add a `turn` into our `Turns` list for log keeping. `CalculateDamageTaken()` method works as follows:

```
private int CalculateDamageTaken()
{
      int rawDmg = CalculateDamage(defShinobi, atkShinobi,...);
      // iterate through attack swipes
      for (int i = 0; i < attacks.Length; i++)
      {
            // if the attack hit is not miss
            if (attacks[i] != SwipeHit.Miss)
            {
                  int dmg = rawDmg;
                  // if we land counter attack (attack was right and
                  // defend was right or vice versa)
                  if ((attacks[i] == SwipeHit.Right &&
                        defends[i] == SwipeHit.Right) ||
                        (attacks[i] == SwipeHit.Left &&
                        defends[i] == SwipeHit.Left))
                  dmg = (int)(dmg * GetBlockMultiplier(
                        defShinobi, atkShinobi));
            }
      }
}
```

Again, this is only a snippet from the long code of `CalculateDmageTaken()` method. In essence, it first calculates the raw damage using `CalculateDamage()` method which only takes into account the stats of attacking and defending shinobi. After that it iterates through the executed attack and defense swipes and if the attack was blocked, the damage of the hit is reduced by a certain multiplier calculated by `GetBlockMultiplier()` method. The swipe is countered/blocked if the defense swipe was performed with the same direction as attack swipe, i.e. if there is an attack coming from the right side, we have to do the swipe to the right to counter it and vice versa.

`CalculateDamage()` methods computes the raw damage according to the following formula:

```
(attack/defense) * base_dmg * type_modifier * log(2*lvl + 10)
```

It takes into account the attack stat of the attacking shinobi and divides it by defense stat of the defending shinobi. Then it multiplies it with base damage of the jutsu and type modifier which raises the damage or lowers the damage depending on elements of attacker's and defender's jutsu. Finally, it multiplies the previous with decadic logarithm of twice the number of level plus ten. The formula is designed so the power growth of a shinobi is quick at the start, but slows as shinobi progresses. It is supposed to give players that play a moderate time a chance to compete with players that play a long time. Of course, the formula is subject to change in order to balance the game, but for that we need the data from testing.

As for `GetBlockMultiplier()` method, this is a formula that the methods uses:

```
min { (resistance/(5*attack)) + 0.3; 0.7 }
```

It takes resistance stat of the defender and divides by attack stat of the attacker multiplied by five and then adds 0.3. If the resulting value is greater than 0.7, it takes 0.7. We are trying to generate a value between 0.3 and 0.7 depending on resistance and attack stats.

The code snippets from `EvaluateTurn()` and `CalculateDamageTaken()` methods do not contain commands for evaluating the combat effects. They were left out to keep the code snippets clean and because the commands are fairly straightforward. For those readers who are interested, please see the source code.

3.3.4. Multi-player Battle

With our combat manager unit done, let us see how a multi-player battle would work. Multi-player battle can only be initialized after a successful connection was established between the server application and the client application. Once the server and client players are connected, the server application sends the client application a control string. When the client application receives the control string, it sends back the confirmation string. If the server receives the confirmation, the connection is up and running and the server application initializes the battle. We will analyze the battle from both server and client perspectives.

Server Side

When the battle starts, the very first thing that the server application has to do is initialize a new instance of CombatManager which manages the whole battle. After that the server player waits for the client player to send the information about the client player/shinobi to the server player, because the server application hosts the CombatManager and it needs the information about both players.

```
string json = await bluetooth.Receive();
EnemyPlayer = (Player)JsonConvert.DeserializeObject(json,
typeof(Player));
```

As you can see, everything that players send to each other is text in JSON format. What we do is we convert the object we want to send to JSON and then we send it over the bluetooth. The recipient receives the JSON and deserializes it into an appropriate object. After the combat manager is initialized, we jump into a combat loop. The combat loop for the server application is implemented in `ServerCombatLoop()` method:

```
private async void ServerCombatLoop(CombatManager manager)
{
      while (true)
      {
            // ATTACK PHASE
            // ------------
            // we wait for player 2 to attack
            string json = await bluetooth.Receive();
            ActionJson clientAction =
                  (ActionJson)JsonConvert.DeserializeObject(
                  json,typeof(ActionJson));

            // if P1 didn't already finished attacking,
            // we wait for it
            if (!FinishedExecution)
            {
                  UsedSemaphor = true;
                  await WaitForAttack();
            }
            // set attack actions in CombatManager
            manager.Attack1 = ActionMessage;
            manager.Attack2 = clientAction;

            // send P1 attack action to P2 (server to client)
            json = JsonConvert.SerializeObject(ActionMessage);
            await bluetooth.Send(json);
            ...
```

Here is the snippet from the `ServerCombatLoop()` method. Specifically, this is the attack phase in a turn. Before both players can proceed into the defense phase, the combat manager needs to have attack actions of both players. First, the server application waits for the client player to perform an attack action and sends it in an `ActionJson` object over bluetooth. After that we check if the server player is finished executing the attack action, because in fact the server player's action might take longer to perform.

We have two options here how to deal with the waiting for the server player's action to be executed. Either we could use an active waiting and jump into a loop constantly asking if the server player is finished, but we know that it is not really an option, because it uses system resources and do not forget that we have to consider low performance smartphones. Much better alternative would be using passive waiting. In our case, we are using `SemaphoreSlim` class which implements a semaphore and supports calls with `await`. So basically, what we are doing is checking if the server player is finished executing the attack action and if not, we call `WaitForAttack()` method which will put the main thread to sleep until the server

player is finished. Once the server player's attack action is done, we save both attack actions into the `CombatManager` and proceed to the defense phase:

```
// DEFENSE PHASE
// -------------
json = await bluetooth.Receive();
clientAction = (ActionJson)JsonConvert.DeserializeObject(
json, typeof(ActionJson));

// if P1 didn't already finished defending, wait for it
if (!FinishedExecution)
{
UsedSemaphor = true;
await WaitForAttack();
}

manager.Defend1 = ActionMessage;
manager.Defend2 = clientAction;
manager.EvaluateTurn();
// get the result of this turn in json message
TurnResultJson turnResult = manager.GetTurnResut();
json = JsonConvert.SerializeObject(turnResult);
await bluetooth.Send(json);
```

As you can see from the code snippet, the defense phase looks quite similar to the attack phase. We wait for the defense action from the client player. After that we check if the server player is done executing the defense action, if not, we use the semaphore for passive waiting again. After the combat manager receives both defense actions, it evaluates the turn and the turn result is then sent to the client player. Then a new turn begins.

Client Side

Now we will look at the battle from the other side. The first thing that client player needs to do when the battle starts is send the `Player` object to the server application, so that the combat manager has information on both players (for computing damage taken and evaluating turns). The server player sends right back the info on the server player, so that the client player can display current status of the server player. And then the `ClientCombatLoop()` method begins:

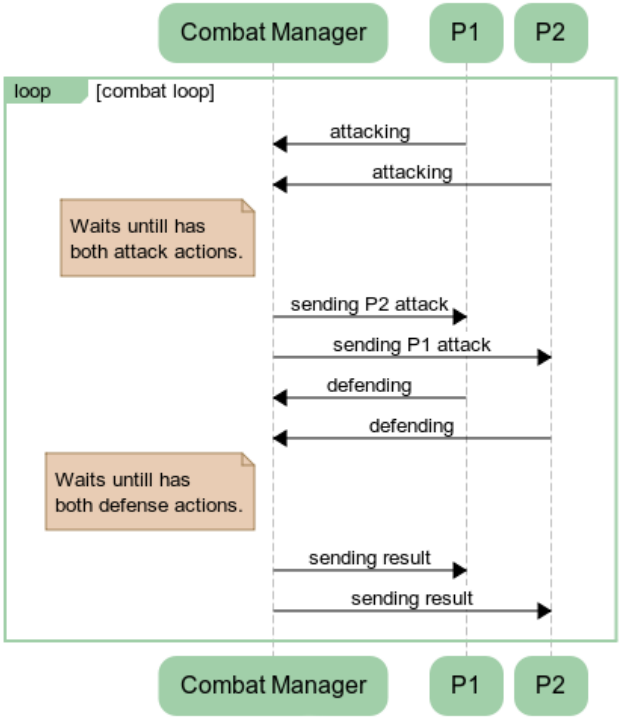
```
private async void ClientCombatLoop()
{
     while (true)
     {
           // ATTACK PHASE
           // ------------
           // receive attack action of P1
           string json = await bluetooth.Receive();
           ActionJson attackJson =
           (ActionJson)JsonConvert.DeserializeObject(
                 json, typeof(ActionJson));
           // remember attack message from enemy
           EnemyActionMessage = attackJson;
```

The attack phase in the `ClientCombatLoop()` only contains code for receiving the attack action from the server player. We need that because we need to animate the server player in order to see how we should counter the incoming attack. Notice that the client player will not receive the attack action of the server player before the client player is done executing his/her attack action. Remember that the server player sends info about his/her attack action only after the combat manager has attack actions of both players. It means that the client player must have already executed the attack action before the call to receive the server player's attack. For better understanding, please see the picture 6. describing the battle cycle of both players.

The defense phase for the client server is described by the following code snippet:

```
// DEFENSE PHASE
// -------------
// Receive the result of the current turn
json = await bluetooth.Receive();
TurnResultJson turnResult =
   (TurnResultJson)JsonConvert.DeserializeObject(json,
      typeof(TurnResultJson));
// … updates status on the screen
```

In the defense phase of the `ClientCombatLoop()` method the client only waits for the result of the turn and updates the players' statuses. As for the attack phase, the client player must have already executed the defense action before this code is reached. The server application waits for both defense actions before evaluating the turn and sending the turn result to the client player (see the picture 6).



Picture 6.: Combat cycle

3.3.5. Sending Shinobi on Missions

In this section we discuss how to implement the feature of sending a shinobi on a mission. As explained in section 2.3.3., missions are a way of passive progressing. We want to be able to send our shinobi on a mission which takes a certain amount of time to complete. When that time is elapsed, our shinobi returns and is rewarded for completion of the mission.

First let us create a new class `ActiveMission.cs` which looks follows:

```
public class ActiveMission
{
     public int ID { get; set; }
     public bool Active { get; set; }
     public DateTime FinishTime { get; set; }
}
```

In the class description we can see the ID of the mission, boolean variable saying if the mission is still active or not (mission is active until the player claims the reward) and time when the mission is supposed to end.

When the player starts the mission, the system creates an ActiveMission instance and calculates the time when it is supposed to finish. It simply takes `DateTime.Now` to get the time at the moment and adds the duration of the mission to calculate the finish time. After that we use `ApplicationData` class, just as we did with `Player` object) to get access to local storage of our Windows Store application and we save the instance of the active mission to an external file. We need to save it to external file because there are missions that take several hours to complete and we need to be able to preserve the information about active mission even when the game is off.

3.3.6. Hunting and NPCs

Implementing the Hunting feature for Ninshu Arts does not require much extra work. Basically we have all we need from the multiplayer battles, only now we will act as if the player is a server player and the client player will not be represented by a human player, but by an NPC unit.

We have a number of NPC units, each with its own preset of jutsus and stats. These NPC units will be stored in our GameContent library as XML files, just like we did with jutsus and missions. When needed, we load the XML file and parse the data into a `NPC` class object.

The `NPC` class is described by the following code snippet:

```
public class NPC
{
     public int ID { get; set; }
     public Player Player { get; private set; }
     public int[] Cooldowns { get; set; }

     private Random rng;
     private int ChanceToHit;
...
}
```

When you look at the code, it seems a little similar to `Shinobi` class. It has the ID of the NPC unit and a reference to a `Player` object. This might look confusing at first, because there is no player controlling the NPC. But the NPC unit must have stats and jutsus stored somewhere and all that is stored in a `Player`

instance. Then it has an array for cooldowns for jutsus. Next there is a random number generator to randomly generate hits, etc. The `ChanceToHit` variable tells us how likely is it for an NPC to land a hit.

3.4. Creating the GUI

The only thing that is left to analyze in our technical documentation is the game presentation layer. It is what connects players with the inner game logic layer where all the game features and mechanisms are. In this subchapter we will shortly go through the process of making GUI for Ninshu Arts using XAML[22]. We will not be going into much detail here. We will only point out some of the interesting features and visual effects achievable in XAML.

3.4.1. GUI and XAML

There are several game screens in the game. We have the main village screen which acts as a signpost to other screens. Then there is screen for mission module, screen for hunting module, screen for profile, screen for multiplayer and others. Each of these screens has their XAML file, for example, `Village.xaml`, `Missions.xaml`, etc.

Every XAML file has an underlying code file corresponding to it. For instance, the `Village.xaml` file has `Village.xaml.cs` file which contains the code for implementing behavior of the XAML elements. In the `Village.xaml.cs` file would find a declaration of `Village` class:

```
public sealed partial class Village : Page
{
     public Village()
     {
           this.InitializeComponent();
     }
...
}
```

[22] We will not be explaining basics of XAML here. It is fairly easy to understand after reading first chapters of almost any tutorial found on internet. For those readers who are more interested in learning XAML, we recommend the book Building Windows 8 Apps with C# and XAML by Jeremy Likness [27].

Let us say, we want to have a button that is supposed to do a certain action when we click on it. In the XAML file, we would define the button as follows:

```
<Button x:Name=”OurButton” Tapped=”OurButton_Tapped”>
    Button
</Button>
```

You can see that we assigned the name to the button. We called it `OurButton`, which is also an identifier which would allow us to reference this exact button in our code. Notice that we created an event handler on `Tapped` event. Since we are in an Windows Universal application, the `Tapped` event catches both touch tap gesture and mouse click. This is very convenient for us because by implementing the behavior for touch devices, we also get the implementation for non-touch devices using mice. Now in our code, we would have a method handling our `Tapped` event:

```
private void OurButton_Tapped(object sender,
    TappedRoutedEventArgs e)
{
    // do something
}
```

Placing and positioning XAML elements to create a complete XAML page is done using XAML layouts[23]. These layouts are designed to provide a way to describe a page layout that would be robust for screen scaling. In most cases, `StackPanel` layout and `Grid` layout and their combinations are sufficient.

By putting elements in a StackPanel, the elements are arranged from top to bottom or from left to right. Grids work a little different. While working with Grid layout, you can first define columns and rows of the grid. After that you can specify for each element in which cell of the grid it will be positioned. When there are more elements in one grid cell, they are positioned over each other. For example, we can create our custom buttons with this:

```
<Grid x:Name="ButtonProfile" Tapped="ButtonProfile_Tapped">
    <Image Source="./Images/button-background.png"
          Stretch="Uniform"/>
    <TextBlock VerticalAlignment="Center"
          HorizontalAlignment="Center" FontSize="40">
          Profile
    </TextBlock>
</Grid>
```

[23] For more information on XAML layouts, please visit the article Defining Layouts (XAML) on MSDN [28].

As you can see, we uses a semitransparent `button-background.png` image and the background of the page is visible behind the button. Then we have a `TextBlock` which is stacked on a `button-background.png` image. You can see how the buttons look like on the pictures 6. and 7.

As all the screens are written in a similar way, we will only describe the creation of `Village.xaml` page. Other XAML pages in our game use the same methods to describe the screens.

```
<Grid Background="{ThemeResource
            ApplicationPageBackgroundThemeBrush}">
      <Grid.ColumnDefinitions>
            <ColumnDefinition Width="1*"/>
            <ColumnDefinition Width="6*"/>
            <ColumnDefinition Width="1*"/>
      </Grid.ColumnDefinitions>
      <Image Grid.ColumnSpan="3" Source="./Images/village.jpg"/>
      <!-- RIGHT COLUMN -->
      <Grid Grid.Column="2">
            <Grid.RowDefinitions>
                  <RowDefinition Height="1*"/>
                  <RowDefinition Height="Auto"/>
                  <RowDefinition Height="Auto"/>
            RowDefinition Height="7*"/>
            </Grid.RowDefinitions>
      </Grid>
...
</Grid>
```

In the previous code snippet you can see that we define columns of the grid on the top. The `Width` and `Height` attributes can have exact values or values with star notation which divide the screen in a ratio. The `Auto` value which you can see in the definition of rows adapts to the content.

Picture 6.: Grid columns

Picture 7.: Grid rows

3.4.2. Battle Screens

Let us now talk about a little more interesting matter that is a battle screen. In this project we implemented two battle screens, `MultiplayerBattle.xaml` and `SingleplayerBattle.xaml`. Both are very similar, only one has the code for running a multi-player battle and one for single-player battle.

We will only describe the `MultiplayerBattle.xaml` page here since the `SingleplayerBattle.xaml` is implemented the very same way. Here is how the XAML page of multi-player battle scene looks like:

```
<Page
    x:Class="WindowsStoreClient.MultiplayerBattle"
    ...
    Loaded="Page_Loaded">
    <Grid Background="{ThemeResource
          ApplicationPageBackgroundThemeBrush}">

        <local:D3DMPBattleScene Margin="0"/>
    </Grid>
</Page>
```

This is virtually the whole code of the `MultiplayerBattle.xaml` page. The most important part of this code is the `D3DMPBattleScene` element inside the `Grid` layout. In section 3.1.5. we were talking about combining a Direct3D application into XAML. We said back then that we are going to integrate our rendering engine into XAML using a `SwapChainPanel`. Let us take a look at the `D3DMPBattleScene.xaml` page which contains definition of the `D3DMPBattleScene` element that we just talked about earlier:

```
<SwapChainPanel x:Name="swapChainPanel"
     x:Class="WindowsStoreClient.D3DMPBattleScene"
     ...>
     <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="150"/>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="150"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
</Grid>
// other content
</SwapChainPanel>
```

Instead of having `Page` element as for other XAML files, this XAML file has `SwapChainPanel` as a root element. If you recall, `SwapChainPanel` inherits from `Grid`, so in essence it is an advanced `Grid` layout with a Direct3D application running on the background. This is how the `D3DMPBattleScene` class looks like:

```
public sealed partial class D3DMPBattleScene : SwapChainPanel
{
...
}
```

If you think about it, this is quite a powerful tool for us. XAML alone is fairly potent. Combining XAML with DirectX might bring out a lot of interesting visual effects. As for our project, we use XAML to add buttons for players to control the battle.

Other thing we can do is to use *storyboard animations* to create little animations to make the game look more alive. Here is the example of a storyboard animation:

```
<Storyboard x:Name="ElementAnimation">
<DoubleAnimation Storyboard.TargetName="ElementImage"
     Storyboard.TargetProperty="Opacity"
     From="0" To="1" Duration="0:0:0.5"
     AutoReverse="True" RepeatBehavior="Forever"/>
</Storyboard>
```

This particular storyboard animations animates a semitransparent image of an element when a shinobi executes a jutsu and changes the opacity of the image from 0 to 1 and backwards. The animations starts after we call `ElementAnimation.Begin()`.

Lastly, we used a `Canvas` element inside of our `SwapChainPanel`. `Canvas` allows us to draw points, lines and shapes on it. So if we actually have a `Canvas` on top of a `SwapChainPanel`, we can draw on top of a 3D scene rendered by our engine.

4. User's Documentation

This chapter will work as a game manual for the players. We describe all the game screens here. After reading this, the reader should be able to tell what to find where and have an idea how things work. Chapter 4 along with chapter 2 should provide all the necessary information for players to understand Ninshu Arts and know how to play it.

4.1. Game Screens

In this subchapter we describe the game screens to the reader. We will try to explain what each screen contains and what it provides.

4.1.1. Village

The village is a first thing that players will see when they enter the game. It serves as a signpost to other places of the game. When looking at the village screen, you will see icons leading to other places. Clicking on the appropriate icon will take you to the place you want.

Picture 8.: Village

4.1.2. Profile

In the profile screen you will find all the informations on your character. This is a place where you can come when you need to find out your shinobi's level or check the general stats or elemental stats.

Picture 9.: Profile

4.1.3. Missions

You can visit the missions screen to send your sinobi on a mission. When you first come to the missions screen, you will see the missions that are available to you. After you pick the one you want, you can start the mission. Each mission takes a certain amount of time to complete. If you come back to the missions screen and your shinobi is still not back from the mission, you will see how much time is left. After the mission is completed, you can come to this screen and claim your reward.

4.1.4. Hunting Assignments

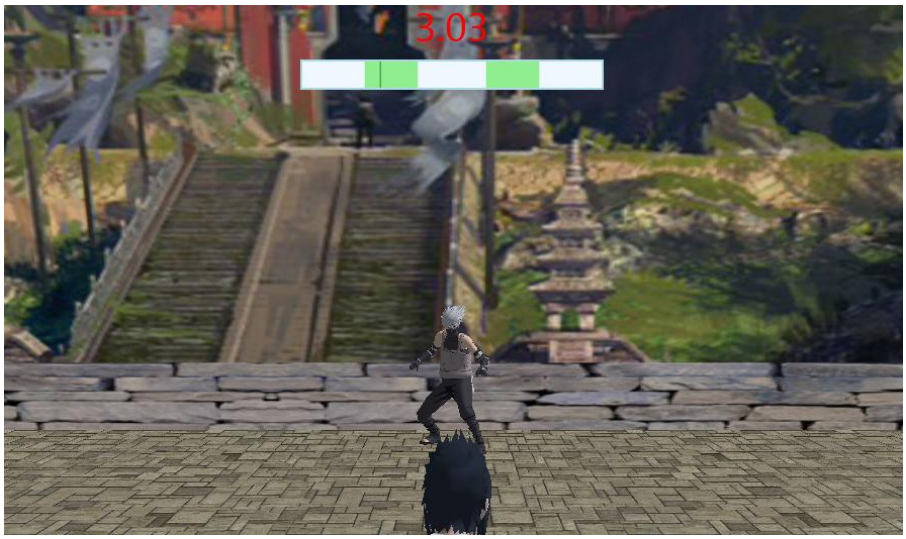
Your shinobi can take hunting assignments on this screen. The game will find you an assignment suitable for your level. You will be sent to hunt down enemies of the village and dispose of them. After you start the hunting assignment, you will be sent into a single-player battle, where you have to defeat your opponent. If you are successful, you will get a reward for completing the hunting assignment.

4.2. Fighting

This subchapter will give players instructions to win the battle. We briefly review the course of the battle and explain what players should do.

First, let us remind the readers how the battle goes on. The battle is conducted in turns. Each turn has the attack phase and the defense phase. Both shinobis are attacking at the same time and defending at the same time. At the start of each turn, players selects a jutsu that they want to use in that turn. To find out information about the jutsu, right click on it with mouse or use hold gesture, if you are using a touch device (hold your finger on jutsu icon). Then you select a jutsu by clicking/tapping on it. After that the attack phase begins.

Each jutsu has a certain number of hits. That means how many hits players can execute during the attack phase. When the attack phase starts, time panel with green colored intervals is displayed at the top of the screen. Players are supposed to execute the hit when the indicator is inside the green area. Hits can be executed by either swiping with a finger in left or right direction, if they are playing on a touch device, or clicking a mouse and dragging the mouse in left or right direction. Please, be reminded that the hit needs to be executed while the time panel indicator is in the green area. If player misses the green area, the hit is not valid and is considered a miss. See the picture below for better understanding.



Picture 10.: Battle

After the attack phase ends, the defense phase starts. Now players have to counter the incoming attacks. Right after the defense phase starts, players are shown the animation of their enemy attack combo. The enemy shinobi either punches or kicks from left or right direction. To block/counter their incoming hit, player needs to perform a swipe to the direction from where the attack is coming, i.e. if you see the enemy shinobi doing a kick from the right, you have to perform a right swipe to counter, and vice versa. The animations may vary, so you need to pay attention. Also the animations are not always the same.

4.3. Multi-player

The multi-player (MP) battle is fought in a way that was described in subchapter 4.2. But compared to a single-player battle, you will be fighting against a human player, not an NPC. To engage in a battle with your friend, you need to establish the connection in the game. And to do that, the first need to pair your devices over bluetooth correctly.

MP battles can be fought between any type of devices. It can be a desktop PC, notebook, tablet or smartphone; provided that the devices are running Windows 8.1 or Windows Phone 8.1 operating systems and they have a bluetooth adapter.

MP screen can be accessed from the village screen. On the first page you will see two options - host or join. Click on “host”, if you want to be the server player and host the game, or click on “join” to be the client player.

Warning: In case the MP battle is fought on a smartphone and a non-smartphone device, it is highly recommended that the MP game is hosted on a non-smartphone device. The reason is simple, PCs, notebooks and tablets are more powerful devices than smartphones. To ensure the best game experience for both players, it is recommended to follow this warning.

4.3.1. Pairing the devices

Pairing the devices over bluetooth might be tricky at times. We would like to point out that it is not a flaw in our game. If you have ever dealt with bluetooth technology before, you surely know that pairing the devices might take some time to do successfully. With that said, here is a set of instructions to pair the devices (in this example we will be pairing the notebook and the smartphone):

1. On the server application go to the MP screen and click on Hosting
2. Now we recommend to split the screen to see the Ninshu Arts on one half and the desktop on the other half (make sure that you are in desktop, it is important!)
3. On the server application click on Start Advertising
4. On the client device go to bluetooth settings (make sure bluetooth is enabled on both devices), find the server device and click to pair with the device
5. The notification should appear on both devices, asking to check if the confirmation number code matches; click yes
6. After that devices should be paired correctly

The pairing of the devices should be persistent. Usually, the pairing needs to be done once and then the players can engage in any number of battles. However, if for any reason, you are not able to connect to the server player in the game, try to remove the paired device from both devices and start the pairing process from the start.

Note: If you are pairing a smartphone to a smartphone, it is important that the client device starts the pairing when the server is advertising the service.

4.3.2. Connecting for MP Battle

Once the devices are connected, one player assumes the role of a server player and hosts the game and the other player assumes the role of a client player and joins a hosted game. To make a connection between two game applications, the server player needs to go the the host game screen and click on Start Advertising. While the server player advertises the service, the client player needs to go to the join game screen and search for advertised games. If the devices are paired correctly, the client player should be able to find the server application. After the advertised application is found, the client player clicks on the name in the list and establishes a connection to the server. On a successful attempt, both players get the notification. When both players are connected, the server player can start the game.

4.3.3. Troubleshooting

If for any reason the client player is not able to find the server player or if the client finds the server, but cannot make a connection, try one of the following:

1. Make sure bluetooth is enabled on both devices
2. Make sure the drivers are fully updated
3. On both devices remove the paired device and try pairing the devices from the very beginning

4.4. Distribution and Installation

As all Windows Store applications, Ninshu Arts will be distributed and installed automatically through Windows Store in the future. It cannot be found and installed through Windows Store yet, because first it needs to go through the certification process. For the time being, Ninshu Arts can be acquired as a downloadable package. Here are the steps to install the application from the downloaded package:

1. Extract the RAR or ZIP archive
2. You will see .appxupload file and a folder; go to the folder
3. In the folder you will see .ps1 file; right-click on in and click on "Run with Powershell"
4. Follow the instructions on screen
5. You will be asked to sign log into your Microsoft account (the one you use to log into your Windows 8.1 / Windows Phone 8.1 system)

Note that this method only works for devices with Windows 8.1 OS.

Another way to launch Ninshu Arts on your device (this works for phone devices as well, but you need to have your phone unlocked for development), is to open game project in Visual Studio and use Deploy or just simply launch the project.

Note: The package provided in the attachment to this paper is fully working and it includes a preset character (stats and skill set) to make it easier and more comfortable to try and test the game.

5. Making the Game

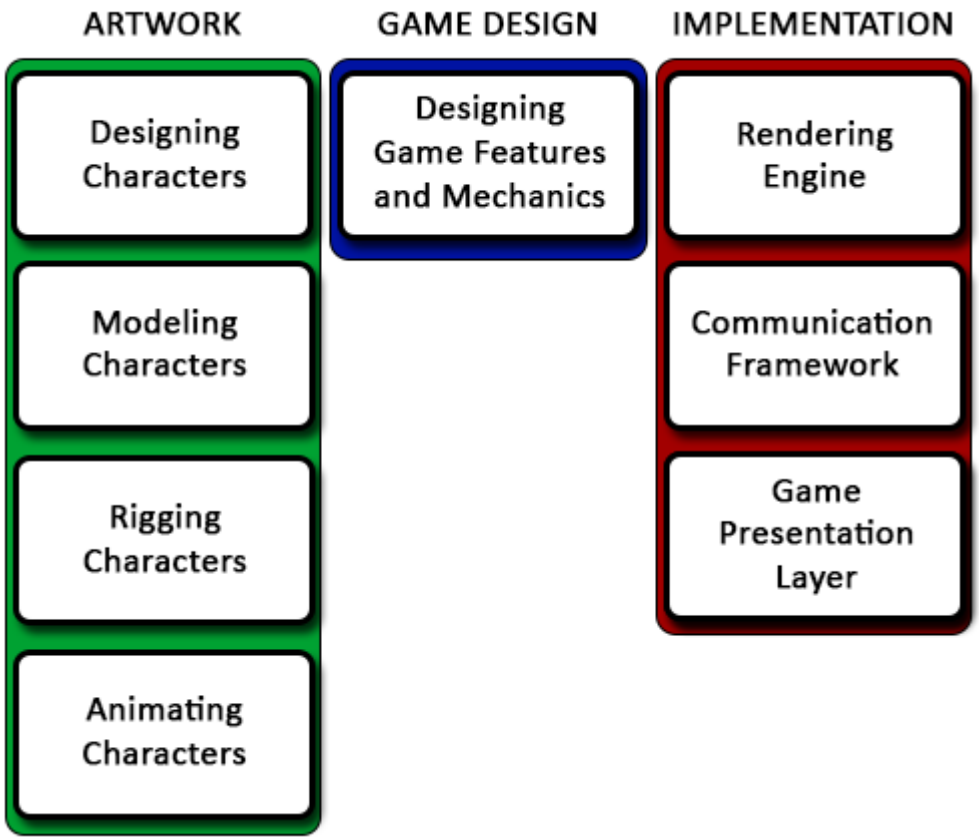
In this chapter I am going to sum up the process of making the whole game. Creating a new game from nothing might not look complicated to someone, but in reality it is a very complex and difficult task where one needs to focus on many different things. It is not a task that you could do on your own. Usually a whole development team is needed. However, in this chapter I would like to describe how a one-man team would cope with such a task. Also, during the process of developing Ninshu Arts I have run into some problems regarding the multi-platformity feature. I would like to provide some informations on how I tried to cope with these issues.

5.1. Game Development Stages

This subchapter should summarize the process of Ninshu Arts development and shortly describe the individual stages of the development. Even though the stages of my development process might differ from the process of other development teams, I believe it is overall the same.

The stages of Ninshu Arts development:

1. Artwork stage
    a. Drawing character concepts
    b. Modeling characters
    c. Rigging characters
    d. Animating characters
2. Designing stage
3. Implementation stage
    a. Rendering engine
    b. Communication framework
    c. Game logic layer
    d. Game presentation layer



Picture 11.: Development stages

As you can see the picture, there were three separate areas in the Ninshu Arts development process. Ordinarily, the development team would divide work and artists would be working on the artwork stage, game designers would be working on game design and programmers would be working on the game implementation. Even though the three stages are connected, the whole process could be parallelized to shorten the development time.

In Ninshu Arts we have a 3D scene where characters fight against each other. The main focus in the artwork stage should be on characters. We need to have a 3D representation of characters in our engine. In order to do that, we need rigged and animated models. Usually, the first step would be acquiring concept drawings which are then used by modelers to model the characters. Models are then rigged and animated. The stage 1. of the Ninshu Arts development will be described in subchapter 5.2.

The stages 2. and 3. of the development process is actually what we covered in chapter 2 (game design) and chapter 3 (game implementation). I will not be going over those again. However, I will point out some issues that I ran into when I was developing Ninshu Arts. These will be discussed later in subchapter 5.3.

Of course, there are other things that need to be done. For instance, testing stages are very important part of the development process. I did not include it to the previous stages because large scale testing is not part of this work.

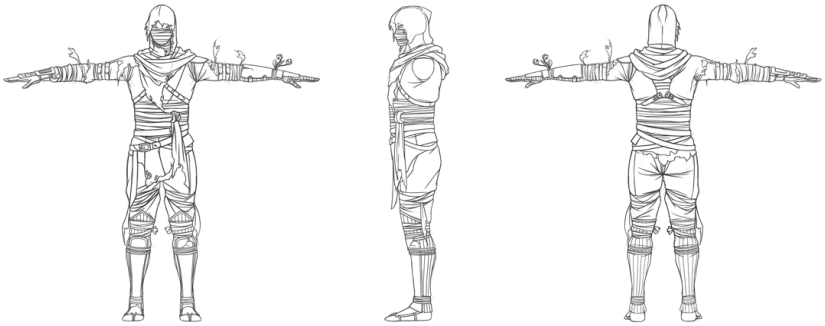
5.2. Artwork Phase

The artwork phase stand at the beginning of the development pipeline. Along with the game design phase, it is basically creating something out of nothing. In Ninshu Arts project we need 3D models of characters. In the following four sections, I will try to describe the process of coming from character drawings to the point where you have rigged and animated character models.

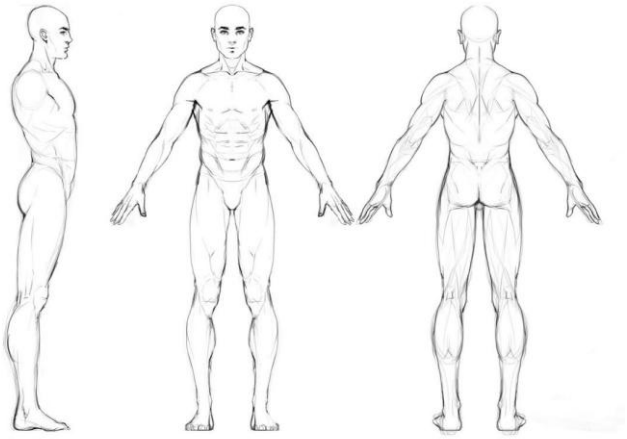
5.2.1. Drawing Character Concepts

This is a very first step of the artwork stage. In this step artist are tasked to design a character and create concept drawings. It is not a work for anyone. Not everyone has talent and the right sense which are definitely needed for this kind of job. Modelers need concept drawings as a template or an example in order to be able to create models. Usually the artists need to provide drawings of the designed character in front view and side views, possibly top view. The more drawings, the better, but sometimes the front view and left/right view is sufficient. Also the characters are drawn in T-Pose or sometimes in A-Pose (see pictures below).

As for Ninshu Arts project, I skipped this step. Sadly, I was not blessed with talent for art and unfortunately artists that I asked to work with me were all too busy to help me.



Picture 12.: Concept drawing of a character in T-Pose



Picture 13.: Concept drawing of a character in A-Pose

5.2.2. Modeling Characters

When the concept drawings are done, they are handed to modelers. Every modeler has his/her own skill set and can use different software to create models. Modelers nowadays usually work with Autodesk Maya [32], Autodesk 3DS Max [33], ZBrush [34] or Blender [35]. There is a lot of other available tools, but these four are among those commonly used. Maya and 3DS Max are products of Autodesk which is a major company in the field. ZBrush is favourite for its different approach. While in Maya and 3DS Max modelers are working with faces, vertices and edges, ZBrush provides sculpting brushes, where you start with a shape and manipulate it

and sculpt it. Blender is also popular, mainly because it is free, while the first three are very expensive.

In this step of artwork stage, I was not creating models from concept drawings, because I was not provided by the drawings. In addition, creating a single model can take weeks or even a month. The models that are used in this project are downloaded free models and edited in Maya to suit our purposes[24].

5.2.3. Rigging Characters

Next step in the artwork stage is rigging the characters. In the process of rigging, the artist creates a skeleton for a model (also called a rig). This skeleton is then used to map animation data to it. It is essential to have a rigged model, because our engine implements skeletal animation. Rigging can be done in any of the mentioned tools, except for ZBrush, which is only a modelling tool. Nevertheless, Maya, 3DS Max and Blender have tools that modelers can use to rig a model.

As for Ninshu Arts, I was personally rigging the edited models in Maya. Maya is not just a modeling software, it is mainly an animating software. So naturally it includes tools to rig and animate characters. Usually, it is better for a single person to work on the rigs because sometimes it is essential that the models have the same number of bones and the same bone structure.

Readers might be interested to know that there are auto-rigging services that take only a few minutes. Users only need to upload a model in T-Pose or A-Pose, usually in FBX format. Mixamo[25] offers many services including auto-rigging which works surprisingly well. However, these are paid services.

5.2.4. Animating Characters

Animating is the last step of the artwork stage. When the models are rigged, the only things that is left is to make the character models move. Maya, 3DS Max

[24] Free models were downloaded on TF3DM [42].
[25] Mixamo is an online service providing models, auto-rigging and animating [36].

and Blender have tools to animate character models. Every software is different though. Skills in one software do not essentially transfer to the other. Usually, large companies are oriented in one software that they are using and they focus on it.

When developing Ninshu Arts, I have tried a little different approach. All the three mentioned applications are based on moving the bones and working with keyframes to create the animation. I was trying to animate characters using motion capture[26] (mocap) and using the data from mocap to map it on my rigged characters. The problem is that big mocap studios use equipment that is not affordable to small development teams. However, there is a way how to get an affordable small mocap studio. The only thing that you need is a device called Kinect[27]. Kinect has infrared sensors and is able to capture the body movement. There are two applications that are able to record mocap data using the Kinect - iPi Soft [39] and Brekel Pro Body [40]. The mocap data are then imported into Autodesk MotionBuilder [41] which is a specialized software for animating. Along with mocap data you are able to import your rigged model. You can then work on your animations in MotionBuilder where you can edit animations, cut animations or create entirely new animations.

5.3. Issues During Game Design and Implementation Stages

Ninshu Arts is a multi-platform game featuring PvP multiplayer battles with cross-device communication. This single sentence presents a lot of problems. Since it was supposed to be a multi-platform game, there were a lot of matters in game design stage that needed to be handled with caution. Multi-platformity was not easily achieved in the implementation stage either. The fact that Ninshu Art is supposed to target so many different types of devices is challenging on its own. Each device has different hardware capabilities. In addition, not all the devices that we target are running the same operating systems. Smartphones are running on Windows Phone 8.1 OS, while tablets, notebooks and desktop PCs are running on Windows 8.1 OS. It goes without saying that these two OS have different capabilities.

[26] *Motion capture* is the process of recording movement of objects or people [37].
[27] *Kinect* is a sensor developed by Microsoft [38].

While I was working on the rendering engine, I found out that not all the devices running WP8 and Win8 operating systems support DirectX feature level `11_0`, even though Direct3D 11 is supported by both OS. If you recall, we pointed out this problem in subchapter 3.1. when we were implementing the game rendering engine. In order to map the devices that I could eventually target, I did a little research and testing on different types of devices. I would like to present the results of the research here. The following table contains data on tested devices.

Device	*OS*	*Feature Levels*	*Shader Model*	*Shader Profiles*
desktop PCs	Win 8.1	`11_1, 11_0`	SM5	`5_0`
notebooks	Win 8.1	`11_1, 11_0`	SM5	`5_0`
HTC 8X	WP 8.1	`9_3`	SM2	`4_0_level_9_3`
Samsung Ativ S	WP 8.1	`9_3`	SM2	`4_0_level_9_3`
Lumia 930	WP 8.1	`9_3`	SM2	`4_0_level_9_3`
Surface 3	Win 8.1	`11_1`	SM5	`5_0`
Surface 2	WinRT 8.1	`9_1`	SM2	`4_0_level_9_1`

Table 1.: Tested devices with their supported feature levels, shader models and profiles

As you can see from the previous table, desktop PCs and notebooks running Windows 8.1 OS do not have problems with running Ninshu Arts with full capabilities of our rendering engine. Virtually all the smartphones with Windows Phone 8.1 OS can use our rendering engine with slight limitations that were mentioned in subchapter 3.1. due to the support of feature level `9_3` and SM2. I also conducted testing on two tablet devices. One of them is Surface 3 tablet from Microsoft which is running Windows 8.1 OS and has no problem at all. On the other hand its older predecessor, Surface 2 tablet, was very problematic. Surface 2 is running the Windows 8.1 RT[28] OS which is no longer developed. It appeared on several devices, but after two years it was written off. It only supported feature level `9_1`. Also SharpDX library, as it appeared after some testing and researching, does not support Surface 2 devices. That is the reason why I was not able to run the

[28] *Windows RT* is an OS developed by Microsoft running on devices with ARM processors. For more information, please see [31].

rendering engine on a Surface 2 tablet (please note that Ninshu Arts does not target devices with Windows 8.1 RT OS.

5.4. Future Development

Ninshu Arts project is only a starting point for a much larger project that I plan to implement in the future. In order to compete with other mobile games at the market today, Ninshu Arts also needs to grow. Here is a short list of features that are planned to be implemented.

Feature list for future development:

1. Online server application to host PvP battles on internet
2. Online social hub (online village)
3. Multi PvP battles with more than two players (2v2, 3v3)
4. Particle system for visual effects

Being able to play an RPG game where you improve and progress your character and are able to play with your friends no matter on what device (Microsoft Windows 8 based) you play was just a first step. I always thought about creating a game where you can play alone offline and progress or play with a friend in offline multiplayer and when you come home where you have a connection to internet, you can play with players all over the world. The features 1. and 2. in the feature list correspond to this idea. In addition, having an online village where players can trade scrolls and challenge each other brings a lot of new opportunities.

If playing with a fellow player is fun, playing with more must be even better. I have never had intention to stop at 1v1 battles. Imagine a battle where you play team vs. team. The possibilities and combinations of outcomes is much greater. That is a feature 3 in the future development list.

The previous points were only talking about new game mechanics, bringing something new to do. We cannot neglect the visual side of the game. So far there is only faint visual interpretation of jutsus when shinobis are attacking. With a particle

system in our rendering engine we could create fire in the scene, for example. Fire techniques could actually shoot fire missiles at the enemy, etc.

5.4.1. Things That Were Left Out

In this section I would like to mention a few things that I originally planned to include or implement in this project, but due to insufficient work of artists and a little time shortage and other factors, they were eventually left out.

The *spritesheet animation*[29] engine was created in Ninshu Arts in order to provide at least some visual effects of jutsus. In principle, we have a number of images of an object as it moves in time. The animation is achieved on a flipbook principle where one image is replaced by the following so fast it creates an impression of movement. The images are arranged in a single spritesheet file so that the spritesheet engine has to access the file system only once to load all the images at once. We then retrieve a single image that is needed from the spritesheet. Although the spritesheet animation is present in our game, we are missing the artwork that artists were supposed to provide us.

I originally planned to implement saving the player's data to cloud in order to provide the possibility to play Ninshu Arts on multiple devices with a single save file. Windows Store applications can use the `ApplicationData` class which we already are familiar with. But this time instead of using `ApplicationData.LocalFolder` to access the local storage of the application, we would be using `ApplicationData.RoamingFolder` class. Unfortunately for me, `RoamingFolder` class is not accessible from WP8 operating systems at the time of writing the paper. Hopefully, this will change in the future and I will be able to provide the players with unified game experience.

[29] *Spritesheet animation* is a technique to create an impression of movement using image spritesheet. For more details on the topic, we refer readers to this article [30].

Conclusion

In these last lines I would like to remind the objectives of this work, summarize the work that was done and evaluate effort that was made. The goal of this work was to create a multi-platform role-playing game supporting cross-device communication. The game should provide these following features. First, the game application was to be developed as an Widnows Universal application that should be able to run on all different types of devices with Windows 8.1 and Windows Phone 8.1 systems. Second, the game is supposed to use the communication framework that I was tasked to implement in order to provide the possibility for players to play multiplayer battles no matter what devices they are playing on. The primary objective of the thesis was to research the topic of multi-platformity in games and cross-device communication capabilities. In addition, with this work I attempted to create a complex DirectX application written not in usual C++, but in .NET C# using SharpDX library.

In the first chapter of this paper, the ideas for the game were introduced. Then we explored a few game titles to both gather some inspiration and to point out shortcomings and things that Ninshu Arts should improve. After that follows the game design phase which is described in chapter 2. Here we were talking about different game mechanics and features that our game should offer. We were also discussing what the game can actually do with regards to multi-platformity and cross-device communication. When we were done talking about the game design, we started to turn the game into something real. Chapter 3 takes us through the process of implementation of all the parts of the game. These include the game rendering engine, the bluetooth communication framework, the game logic layer and the game presentation layer. In chapter 4 readers can familiarize themselves with the game. This chapter provides a game manual where players learn how to connect to the multiplayer battle or where to see the character statistics and more. Lastly, chapter 5 was written to provide the readers with an overall process of developing a game. After reading this chapter, the readers should have an idea what game development involves and how a one-man team copes with the work of a whole development team.

In the process of this project, I was able to create a game rendering engine written in C# that is capable of rendering characters in 3D scene and animating them. Then I implemented a communication framework supporting bluetooth peer-to-peer communication that is capable of exchanging messages between devices of different types. On top of the rendering engine and the communication framework I attempted to design a game featuring an interesting concept of progressing and player vs. player cross-device multiplayer battles. In the whole process of writing this paper I tried to keep in mind the primary research objectives. I tried to point out issues and problems concerning multi-platformity and cross-device communication and possibly offer solutions and discussions on the matter. Lastly, as I also did some work in artwork stage of game development myself while working on this project, I tried to capture it in a few lines. The last chapter also summarizes the whole process of game making and possibly offers those readers, who think about game development, at least an idea of what it all requires and what it all involves.

I would also like to point out a few shortcomings of this work. Due to bad cooperation with artists, they were not able to provide materials to implement more visual effects in game. Or perhaps it was my bad management skills that were not good enough to make artists finish in time. I also wanted the game to provide cloud synchronization of the save data, but unfortunately the capabilities of Windows Phone 8.1 operating system is not as good as the capabilities of Windows 8.1 system. And since the game is in a beta-test phase now, you will see temporary working names of jutsus in the game, etc.

For me, Ninshu Arts is only a starting point of a much larger project that I am determined to pursue. There is a lot of things waiting to be implemented in the future. At the time of finishing this paper, the game is in a beta-test phase. There are dozens of players involved in the testing and they are eager for more.

References

[1] Pokémon official web page - http://www.pokemon.com/, Last checked 2015-06-11

[2] Nintendo official - http://www.nintendo.com/, Last checked 2015-06-11

[3] Blood and Glory game info page - http://www.glu.com/games/viewGame/6, Last checked 2015-06-11

[4] Glu Mobile homepage - http://www.glu.com/, Last checked 2015-06-11

[5] Dragon Mania Legends game info page - http://dragon-mania-legends.wikia.com/, Last checked 2015-06-11

[6] Gameloft official web page - http://www.gameloft.com/, Last checked 2015-06-11

[7] Dungeon Hunter game info page - http://dungeonhunter5.com/, Last checked 2015-06-11

[8] Game Link Cable article on Wikipedia - https://en.wikipedia.org/wiki/Game_Link_Cable, Last checked 2015-06-20

[9] Wi-Fi Direct info page - http://www.wi-fi.org/discover-wi-fi/wi-fi-direct, Last checked 2015-06-20

[10] Bluetooth official web page - http://www.bluetooth.com/, Last checked 2015-06-20

[11] Unity3D official web page - http://unity3d.com/, Last checked 2015-06-20

[12] Unreal Engine 4 official web page - https://www.unrealengine.com, Last checked 2015-06-20

[13] SharpDX official web page - http://sharpdx.org/, Last checked 2015-06-20

[14] Justin Stenning, ‘Direct3D Rendering Cookbook’, (2014)

[15] XAML article on MSDN - https://msdn.microsoft.com/en-us/library/cc295302.aspx, Last checked 2015-06-21

[16] DirectX Tool Kit page on Codeplex - https://directxtk.codeplex.com/, Last checked 2015-06-21

[17] Frank D. Luna, ‘Introduction to 3D Game Programming with DirectX 11’, (2012)

[18] Supporting Bluetooth Devices article on MSDN - https://msdn.microsoft.com/en-us/library/windows/apps/xaml/dn264587, Last checked 2015-06-30

[19] ApplicationData class documentation on MSDN - https://msdn.microsoft.com/cs-cz/library/windows/apps/br241587, Last checked 2015-06-30

[20] Newtonsoft Json.NET official web page - http://www.newtonsoft.com/, Last checked 2015-07-15

[21] JSON info page - http://json.org/, Last checked 2015-07-15

[22] XML info page - http://www.w3.org/XML/, Last checked 2015-07-15

[23] Google Sheets info page - https://www.google.com/sheets/about/, Last checked 2015-07-20

[24] DirectX article on Wikipedia - https://en.wikipedia.org/wiki/DirectX, Last checked 2015-07-15

[25] J. Žára, B. Beneš, J. Sochor, P. Felkel, ‘Moderní počítačová grafika’, 2nd edition, (2004)

[26] FBX format article on Wikipedia - https://en.wikipedia.org/wiki/FBX, Last checked 2015-07-25

[27] Jeremy Likness, ‘Building Windows 8 Apps with C# and XAML’, (2012)

[28] Defining Layouts (XAML) article on MSDN - https://msdn.microsoft.com/en-us/library/windows/apps/xaml/hh465337.aspx, Last checked 2015-07-25

[29] Immediate and Deferred Rendering article on MSDN - https://msdn.microsoft.com/en-us/library/windows/desktop/ff476892(v=vs.85).aspx, Last checked 2015-07-25

[30] Spritesheet Animation article - http://gamedevelopment.tutsplus.com/tutorials/an-introduction-to-spritesheet-animation--gamedev-13099, Last checked 2015-07-25

[31] Windows RT article on Wikipedia - https://en.wikipedia.org/wiki/Windows_RT, Last checked 2015-07-27

[32] Autodesk Maya - http://www.autodesk.com/products/maya/overview, Last checked 2015-07-27

[33] Autodesk 3DS Max official web page - http://www.autodesk.com/products/3ds-max/overview, Last checked 2015-07-27

[34] ZBrush official web page - http://pixologic.com/, Last checked 2015-07-27

[35] Blender official web page - https://www.blender.org/, Last checked 2015-07-27

[36] Mixamo official web page - https://www.mixamo.com/, Last checked 2015-07-28

[37] Motion capture article on Wikipedia - https://en.wikipedia.org/wiki/Motion_capture, Last checked 2015-07-28

[38] Kinect info page - https://www.microsoft.com/en-us/kinectforwindows/, Last checked 2015-07-28

[39] iPi Soft official web page - http://ipisoft.com/, Last checked 2015-07-28

[40] Brekel Pro Body official web page - http://brekel.com/brekel-kinect-pro-body/, Last checked 2015-07-28

[41] Autodesk MotionBuilder official web page - http://www.autodesk.com/products/motionbuilder/overview, Last checked 2015-07-28

[42] TF3DM, 3D models database - http://tf3dm.com/, Last checked 2015-07-08

List of Tables