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 renderer pro malé scény umožňující renderování objektů a animování postav. V rámci práce byla také implementována knihovna pro cross-device komunikaci zprostředkovávající bluetooth technologii pro komunikaci mezi klienty běžících na různých zařízeních. Dále byla vytvořena knihovna pro server-client komunikaci, která je použitá v implementaci herního serveru, který nabízí možnost online hraní.

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 scene renderer was written which supports rendering simple scenes with objects and animated characters. A cross-device communication library based on bluetooth technology was implemented in this project as well. This communication library allows two game clients running on two different types of devices to communicate with each other. Then a server-client communication library was created. This library was then used to implement a game server which offers online gaming feature.

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

Table of Contents

Preface 1
General Introduction 1
Problem Statement 3
Goals 4
1. Game Inspirations and Our Game 6
1.1. Pokémon 6
1.2. Blood and Glory 8
1.3. Dragon Mania Legends 9
1.4. Our Game - Ninshu Arts 10
2. Implementation Analysis 12
2.1. Existing Tools for Developing Games 12
2.1.1. Unreal Engine 13
2.1.2. Unity 14
2.1.3. Paradox Engine 15
2.1.4. Wave Engine 16
2.2. Programming Environment 19
2.3. Targeted Devices 19
2.4. Game Graphics 21
2.5. Communications 22
2.5.1. Offline Multiplayer 22
2.5.2. Online Multiplayer 23
3. Designing the Game 27
3.1. Game Concept 27
3.2. Combat System 28
3.2.1. Turn-based Combat with Action Elements 28
3.2.2. Combat Techniques - Jutsus 30
3.2.3. Basic Elements 31
3.3. Game Character - Shinobi 33
3.3.1. Shinobi’s Stats 33
3.3.2. Experience and Levels 34
3.3.3. Progressing and Improving 35
3.4. Game Controls 36
4. Technical Documentation 39
4.1. Writing the 3D Renderer 40
4.1.1. SharpDX 40

4.1.2. Direct3D Introduction ... 41
4.1.3. Rendering Framework ... 44
4.1.4. Implementing the 3D Renderer ... 46
4.1.5. Changes Regarding Transitioning to Windows Store App ... 57
4.1.6. Compromises for Multi-platformity ... 59
4.2. Implementing the Communication Libraries ... 60
4.2.1. Bluetooth Communications ... 61
4.2.2. Web Communications ... 68
4.3. Game Logic and Features ... 72
4.3.1. Player ... 73
4.3.2. Jutsus and Missions Data ... 74
4.3.3. Combat Manager ... 76
4.3.4. Bluetooth Multi-player Battle ... 78
4.3.5. Sending Shinobi on Missions ... 82
4.3.6. Hunting and NPCs ... 83
4.4. Creating the GUI ... 84
4.4.1. GUI and XAML ... 84
4.4.2. Battle Screens ... 86
4.5. Game Server ... 87
4.5.1. Basis and Creation ... 88
4.5.2. Game Database ... 89
4.5.3. Web API ... 91
4.5.4. Testing environment ... 93
5. User's Documentation ... 94
5.1. Game Screens ... 94
5.1.1. Village ... 94
5.1.2. Profile ... 95
5.1.3. Missions ... 95
5.1.4. Hunting Assignments ... 96
5.2. Fighting ... 96
5.3. Bluetooth Multi-player ... 97
5.3.1. Pairing the devices for Bluetooth MP ... 98
5.3.2. Connecting for MP Battle ... 99
5.3.3. Bluetooth Troubleshooting ... 99
5.4. Distribution and Installation ... 100
5.5. Online Features ... 101

6. Making the Game 102
6.1. Game Development Stages 102
6.2. Artwork Phase 104
6.2.1. Drawing Character Concepts 104
6.2.2. Modeling Characters 105
6.2.3. Rigging Characters 106
6.2.4. Animating Characters 106
6.3. Issues During Game Design and Implementation Stages 107
6.4. Future Development 109
6.4.1. Things That Were Left Out 110
Conclusion 111
References 113
List of Tables 116

Preface

General Introduction

Computer games (PC games) have been a significant part of software market for many years now. PC games have evolved so much over the years. Mainly the demand for new titles from the players, improvement of computer hardware components and other advancements in information technologies set the direction for the evolution of PC games. If we compare the early games like Spacewar! [43] where players could only see simple 2D objects flying on the screen with today's AAA games with stunningly beautiful realistic graphics, we must admit that it is a huge leap forward.

The success of 3D console titles such as Super Mario 64 [44] increased the interest in hardware accelerated 3D graphics on PCs which soon resulted in attempts to produce affordable solutions. Soon 2D games left the screens of players and they were replaced by 3D titles such as Tomb Rider [45] which was one of the first 3D third-person shooter games and was praised for its revolutionary graphics. The faster graphics accelerators and improving CPU technology resulted in inreasing levels of realism in computer games. PC gaming currently tends strongly towards improvements in 3D graphics. The high-speed internet access coming to households lead to the next evolution step which was online gaming. Over the past years players had the opportunity to see a lot of online games, especially massive multiplayer online role-playing games (MMORPG) which is one of the most developed game genre.

With the substantial development and progress in the area of partable devices such as tablets and smartphones in the past years, the recent gaming trends move towards cross-device and multi-platform games. Because the term „multi-platform application" can have more meanings, we will from now on mean an application that is able to run on different types of devices.

Definition: By the term *multi-platform application* we will from now on mean an application that is able to run on different type sof device.

As the number of portable devices grows, it is only natural for us to consider applications that are capable of running on different types of devices and the same goes to games. Multi-platform games are relatively new to the market and there are not many quality multi-platform games out there. Most of these games are *casual games*. This may include game genres such as following:

- Puzzle games (games for solving puzzles, matching color crystals etc.)
- Endless running games (games where the player character is continuoisly moving forward through a procedured and theoretically endless game map)
- Word & trivia games
- Card & board games
- Hidden objects games

Almost all the games from the previously mentioned genres are simple games where players are tasked to do the same thing over and over again. Even though breaking the standing records and getting higher score than fellow players is a good motivation to keep playing, after a short time players might still lose interest. On the other hand, there are other game genres that require more time and offer more things to do. This may include for example:

- Adventure games
- Role-playing games (RPG)
- Strategy games

Problem Statement

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 games for portable devices. Although many of these games are enjoyable and popular, they share some of the following shortcomings:

1. Many games for portable devices are casual games which are simple and do not offer much of a game content.
2. Role-playing and strategy games for portable devices offer some content to entertain the players, but most of these games are designed for online gaming. Players usually have to connect to the game server in order to play. Casual games are playable offline, but they lack complexity, while strategy and role-playing games offer richer game content, but are not playable offline.
3. Very few games offer quality multiplayer gaming. While it is a common thing for PC games to have a good multiplayer capabilities, the number of games for portable devices supporting multiplayer is very low. Possibility to be able to play with other human players and a chance to compete with each other is a very strong motivation to play the game.
4. Not all portable games are multi-platform and players will not be able to run them on all the devices. You will find games for smartphones and tablets which will not be available on PCs.
5. There are virtually no games on the market that support *offline multiplayer*. This disadvantage applies to PC games as well, not only portable device games. While this is a common feature for handheld consoles and console games, developers for PC games do not feel the need to implement offline multiplayer, if we consider that every common household has internet access. Now if we take multi-platform games into account, the offline multiplayer feature might bring new gaming experience. Contemporary games with offline multiplayer capabilities are virtually nonexistent.

Goals

We have defined multi-platform games as games capable of running on different types of devices. We have also discussed contemporary gaming trends and the state of the game market these days and we have pointed out some shortcomings of what today's games have to offer. With this in mind, we will attempt to develop a new multi-platform game with the following criteria:

a) Multi-platformity – we will develop a single „universal" game client capable of running on different types of devices (PC, notebooks, tablets and smartphones)[1]
b) Multiplayer capabilities – the game will support both offline multiplayer, where players will be able to engage in multiplayer battles even without access to internet or local network, and online multiplayer to offer players a possibility to play and compete with players from distant places
c) Turn-based role-playing game with action elements – we will try to introduce a few game mechanics and game content and incorporate them into a new RPG to offer better gaming experience than simple casual games

Summary

The primary goals of this work are:

- Analyze the current gaming scene and relevant existing games to address shortcomings and things we can include in our game to introduce something new and at the same time follow contemporary gaming trends.

[1] By the term „multi-platform application" we mean an application which is able to run on different types of devices such as desktop PC, notebook, tablet and smartphone as we defined in General Introduction.

- Analyze available technologies and existing tools relevant to the development process and choose a suitable direction of development for our game
- Develop the game from the ground (design, analysis, implementation)
- Provide some additional information on developing a game as a one man team

1. Game Inspirations and Our Game

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. At the end of this chapter, we will summarize the features for the game that we are trying to create.

First we will discuss one of the best selling role-playing game that has obtained many awards, Pokémon. It is well known for its turn-based combat as well as good looking pocket monsters and a nice fantasy world. Then we will turn to contemporary game Blood & Glory which represents one of the current casual games for mobile devices. Finally we will analyze Dragon Mania Legends which is a great example of today's gaming trend.

1.1. Pokémon

Pokémon [1] is a video game made for Nintendo [2] handheld consoles such as Game Boy, 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.

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 we will try to keep in mind while designing our game.

Pros: a lot of pokémons and attacks of various types, rich game content and mechanics

Cons: pure turn-based combat

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, simplicity being the most notable. 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 the enemy NPC 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 in the arena dodging and striking enemy NPC gladiators. All it took was bringing little action into a turn-based system.

Pros: action elements in turn-based combat, intuitive way to control the game character

Cons: not as rich game content, casual game

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 brief story that will keep players entertained 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 wider yellow area, they deal normal damage. And if they 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 plan their attacks. 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.

One of the great things about Dragon Mania is that it is multi-platform. 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.

Pros: combines tactical turn-based combat with action elements, game is multi-platform

Cons: requires internet connection, has no multiplayer

1.4. Our Game - Ninshu Arts

For the purposes of this project and hereinafter in the paper, we will call our game *Ninshu Arts*.

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 analyzed Blood and Glory where there was a lot of action in the turn-based combat but little complexity and poor game content. At last, there was Dragon Mania Legends where we saw little from both worlds, but not exceeding in any of the two.

Ninshu Arts will have turn-based combat with action combat technique system. Action elements will require players to stay alert, responsive and mindful of the game. Also with a diverse combat technique system, players will also have to think tactically.

A multi-platform feature and the multiplayer feature as we defined is not likely to be found in any other games on 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.

The motivation to implement the offline multiplayer feature is described in a following scenario. Two friends meet some place with no internet access nor local network. One player is playing the game on a smartphone and the other on a tablet. It would be great if they could just connect to each other and play. Or some people commute to work and they like to kill some time playing games on their phones, but not just a casual game. Even with no internet, they are still able to play. And when they get home, they could just upload their local save from their phone to the game server, turn on their desktop PC and start playing where they left off. Then they could play online with other players who are connected to the game server.

2. Implementation Analysis

After analyzing current gaming trends and discussing a few games of interest, we have described the main game features we want to implement in our game. This chapter is focused on technical decisions regarding how our grame will be implemented.

The task of creating a game is a demanding one and usually there is a whole developer team involved, because the development process includes many areas (e.g. artwork, game design, game programming, testing). There are two directions that game developers can take:

1) Using an existing and complete game engine that provides developers with a tool designed for creating games, or
2) build the game from the ground.

There are many existing game engines available. Many of them offer a complete solution for developing a new game. They have a graphics core and other components and some sort of development environment to create game scenes and levels etc. In the following subchapter we will analyze some of these game engines.

2.1. Existing Tools for Developing Games

The features of our interest are:

a) Programming language – language used for scripting in-game logic and content
b) Targeted platforms – what platforms can the created games run on
c) Pricing – whether the game engines are free to use or there are royalties associated

By searching internet for game engines, we are able to find numerous search results. However, in this paper we are going to discuss only a few which we find to be popular or relevant to our work.

A popular game engines with C++ as primary programming language:

- Unreal Engine

Some users prefer more user-friendly language like C# over C++. The commonly used game engine for that is:

- Unity

And then we will point out two engines that not only use C# for scripting, but they were also written in C# (meaning graphics core and other components of the engine):

- Paradox Engine
- Wave Engine

2.1.1. Unreal Engine

Unreal Engine [12] is a game engine developed by Epic Games, it showcased in the first-person shooter game Unreal in 1998. Although it was primarily developed for first-person shooter games, it has been successfully used in variety of other games. Latest Unreal Engine 4 offers tools for creating virtual reality and films. It is a professional tool of first-rate quality.

The code of Unreal Engine and all its tools is written in C++. The source code is available to users which is a big advantage. Before Unreal Engine 4 the developers used UnrealScript which was Unreal Engine's native scripting language. The newest Unreal Engine 4 no longer supports UnrealScript, but instread supports game scripting in C++. Unreal Engine 4 also offers so called Blueprint visual scripting system that allows development of game logic without using C++.

Unreal Engine is free to use, but developers need to pay 5% royalties for each released product.

Advantages

- ✓ Complex tools and editors
- ✓ Stunning graphics results
- ✓ Engine source code available to developers

Disadvantages

- ✗ 5% royalties for every released product
- ✗ Some users prefer higher level programming language over C++
- ✗ Does not target all platforms as competition (e.g. Windows Phone)

2.1.2. Unity

Unity [11] is a game engine developed by Unity Technologies. Unity is noteable for its ability to target games to multiple platforms. Within a project, developers have control over delivery to mobile devices, desktops, consoles, even web browsers. Supported target platforms include Windows, Windows Phone, Android, iOS, Mac OS, BlackBerry 10.

Along with Unreal Engine, Unity is one of the most popular and commonly used game engine among game developers. The quality of development environment and tools correspond its popularity. The code of Unity engine is written in C and C++, the game development environmemt (editor) is written in C#. The scripting and game logic coding are done in the C# language which may be deciding factor for many beginner and independent game developers.

Unity has a personal edition which is free to use, but it includes only the basic features. The professional edition is available for developers for $1500 license or $75/month subscription.

Advantages

- ✓ Professional and complex tools
- ✓ More user-friendly coding and scripting with C#
- ✓ Targets wide range of devices

Disadvantages

- ✗ Pricing, if personal edition features are not enough
- ✗ Engine source code provided on occasion via special agreements with Unity development team

2.1.3. Paradox Engine

Paradox Engine is a new competitor in the field of game engines. It is developed by Silicon Studio and was released just in Autumn 2014. Paradox Engine targets Windows Desktop, Windows Store, Windows Phone, Android and iOS devices.

Even though it is not a front-runner among game engines, we mention it here because it is fully written in C# (game engine, graphics core and all its components). It is running on DirectX for Windows platforms and OpenGL/ES for all other platforms. DirectX normally does not support C#, but only C++. Paradox Engine is powered by SharpDX [13] library which allows developers to access DirectX API from .NET C#[2]. Source code of Paradox engine is available to developers. The interesting feature is integration between Game Studio and Microsoft's Visual Studio where developers can edit the code more efficiently. Paradox might does not offer as complete set of tools as Unreal Engine or Unity, but it includes all the basic essentials.

Paradox Engine can be used free of charge in two ways. Either developers can use Paradox official releases and distribute their games without disclosing the source code, but there can be no change to the source code. Or developers can compile Paradox themselves, they are allowed to change the engine's source code, but they must distribute all the source code with the game.

---

[2] SharpDX library was created by Alexander Mutel who is also part of the Paradox Engine developer team.

Advantages

- ✓ Coding in C#
- ✓ Visual Studio integration
- ✓ Source code available to developers
- ✓ Free to use (with some conditions)

Disadvantages

- ✗ Does not offer as complete and complex tools compared to Unreal Engine or Unity
- ✗ Targets less platforms than competition (but all the common)

2.1.4. Wave Engine

Wave Engine [47] like Paradox Engine is relatively new compared to Unity and Unreal Engine. It was released by Wave Corporation in 2013. And just as Paradox Engine, it provides simpler game editor compared to first-rate game engines, but still offers a good environment for game development.

It targets all the common platforms including Windows Store, Windows Phone, Android and iOS. Like Paradox Engine, Wave Engine is also entirely written in C#. Wave Engine uses SharpDX library to access DirectX API for Windows platform games and OpenTK library to access OpenGL for other platforms[3]. Coding and scripting can also be done in the C# language. One of the notable Wave Engine features is component based game development which usually increases productivity and decreses cost of development. Wave Engine source code is not fully accessible by users. The Wave Engine developer team only provided the source code of some internal components to make the creation of custom components easier for game developers.

---

[3] Information on the used libraries was provided by David Ávila, member of Wave Engine developer team, November 11th, 2015.

Using Wave Engine is completely free for Windows Desktop, Windows Store, Windows Phone devices, while for Android and iOS developers need to have Xamarin [48] license.

Advantages

- ✓ Coding in C#
- ✓ Component based development

Disadvantages

- × Does not offer as complete and complex tools compared to Unreal Engine or Unity
- × Targets less platforms than competition (but all the common)
- × Free to use, but not for Android and iOS
- × Not all source code available to users

	Unreal	**Unity**	**Paradox**	**Wave**
Windows Desktop	royalties	free*	free	free
Windows Store App	not supported	free*	free	free
Windows Phone	not supported	free*	free	free
Android	royalties	free*	free	free**
iOS	royalties	free*	free	free**
Mac OS X	royalties	free*	not supported	free
Linux	royalties	free*	not supported	free

Table 1.: Table showing supported platforms for each engine. You can see whether the given engine supports a platform and does not require royalties or is free to use, or if a platform is not supported at all. (* free only with basic tools, ** requires Xamarin license)

Summary

Even though game engines may provide a complex environment and tools to develop new games, they are not always appropriate. First-rate game engines offer a variety of tools and there are usually a lot of tutorials and materials for them, but they are pricey. On the other hand, less used engines are free to use, but do not offer as rich toolset and there is usually less materials to learn how to use the engine or information about the engine itself. Before choosing to use a game engine, one should also account for the time needed to learn how to use the engine. Furthermore, if developers have specific requirements for the game, choosing a game engine might not be the right decision, because they might not have full control over those specific things.

As for our game project, we would like to use the C# language to code the game. Unreal Engine is not convenient, because it only supports C++ scripting and on top of that is not free to use. On the other hand, Unity and does not provide its source code and is not free either. In addition, we have very specific requirements for our multiplayer features (offline and online multiplayer). Let us consider for a moment that bluetooth is our technology of choice to implement offline multiplayer. If we search for bluetooth plugins for Unity, we will only find a bluetooth plugin for Android [55] and iOS [56] devices (thus connecting a phone to phone or tablet to tablet). There is no simple way to connect a tablet and desktop PC for example. The correct solution in Unity would be developing a new C/C++ plugin for Unity Engine and then incorporating the new plugin in the Unity project. But that would require the Pro version of Unity.

For our game *we will not be using any game engine*, but we will build Ninshu Arts from the ground. Thus we will have full control over the communications which have rather specific requirements and the game itself.

Please note that at the time when work on this project started, Paradox Engine was not even released and Wave Engine was just released (there were not much information on Wave Engine and at that time it did not have all the features it has today).

2.2. Programming Environment

C# has been a very popular choice for programming language among developers for a quite some time now. Over the years the C# language has been extended with a lot of interesting features (e.g. lambda expressions, LINQ, asynchronous programming) and thus it has attracted a wide user community. In addition, the C# language hides many low-level aspects of programming from the users (e.g. pointers, memory allocation and deallocation) and so it allows developers to think in terms of greater level of abstraction. Also this approach is definitely easier for beginner developers to learn. And if we combine this with advance and user-friendly development tools such as Visual Studio, the C# language makes an excelent choice for realization of our ideas.

Nowadays more games are built upon high-level programming languages. As most 3D games are now mostly limited by the power of the graphics card, the potential slowdown due to translation overheads to higher level languages become negligible. For these reasons, we choose C# as the language for the development. We will use Microsoft Visual Studio 2013 as our development environment.

2.3. Targeted Devices

As we stated earlier, we are developing a multi-platform game which means we want our game to be able to run on different types of devices such as desktop PCs, notebooks, tablets and smartphones.

There has been significant development and progress made in the area of portable devices such as smartphones and tablets. In 2013 there have been several millions smartphones sold all over the world and having a smartphone or a tablet at home is a common thing these days. There are three distinct operating systems running on portable devices. They are Android from Google, iOS from Apple and Windows from Microsoft. The first two are dominant in the field of portable devices and the number Windows tablets and smartphones is smaller. This is due to the fact that Windows joined the market of portable devices later than Android and iOS.

Even though Android and iOS are greater in numbers of portable devices, let us not forget the fact that these are only smartphones and tablets. Windows platform is able to target every type of devices, i.e. PCs, notebooks, tablets and smartphones[4]. In addition, the manufacturers are trying to follow the contemporary trend and creating a lot of 2-in-1 devices and hybrid devices (notebooks with touch screens and removable keyboards, devices that can flip the monitor and switch between notebook and tablet mode, etc.). And almost all of these hybrid devices are running on Windows operating system. In addition, desktop PCs and notebooks have been here for decades and Windows operating system is running on most of them. This is why we have decided to develop Ninshu Arts for Windows platform, as we want it to be multi-platform in terms of being able to target every type of devices. And even though Windows is lesser in numbers of portable devices, the number of targeted desktop PCs and notebooks is still substantial and we will be able to approach a large number of potential players. It is also noteworthy that there is not always need to develop a game for all the operating systems, as exclusiveness to one platform can be well sold to the platform developer.

Microsoft introduced the Universal Application model with the release of Windows 8.1 operating system (Windows Phone 8.1 for smartphones) 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. 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 small and large display. Nevertheless, the Universal Application offers an elegant way to write one application that will be runnable an all types of devices.

---

[4] Apple also manufactures its own notebooks and dekstop devices running Mac OS, but their number is rather insignificant.

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.

Note: We consider Windows 8.1 OS, because at the time of writing the thesis it was the latest update of Windows platform. However the game application developed for Windows 8.1 should be runnable and fully functional on Windows 10.

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

2.4. Game Graphics

The game graphics is an important part of the game, since we want to deliver a good gaming experience. As we mentioned earlier, we are trying to develop a turn-based combat game with action elements. To invoke a feeling of action is it almost obligatory to gave a 3D scene with moving camera. For battles in Ninshu Arts, we would like to have a 3D battle scene with moving characters doing attacks as players commands.

DirectX and OpenGL are two common application programming interfaces (API) used to develop graphics applications. In this project, we will choose DirectX developed by Microsoft, since we are targeting Windows devices. Ordinarily, DirectX only supports writing the graphics application in C++. However, there are certain libraries that provide us with the ability to access DirectX API from .NET C#. Namely, the two known libraries are SlimDX and SharpDX. In order to keep all the source code in the C# language, we will be using SharpDX library to access DirectX API. We chose SharpDX, becuase SlimDX is no longer being developed.

We will be developing our own 3D game renderer which will be able to render simple scenes with objects and animated characters in *DirectX* using *SharpDX* library. The character animation will be implemented using skeletal animation.

2.5. Communications

The multiplayer requirements for our game are quite unique, as we are trying to implement both online multiplayer and offline multiplayer, which is a unusual feature for PC and mobile platform games. We shlould carefuly think about what technologies to use.

2.5.1. Offline Multiplayer

As we mentioned previously in previous chapters, 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. We design Ninshu Arts with emphasis on multi-player battles and PvP. We believe that the possibility to play with other people and compete with each other is a great motivation for players to keep playing.

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 library 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. However, 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 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.

2.5.2. Online Multiplayer

Online gaming is a common feature in contemporary games on the market. Providing players with the possibility to play offline multiplayer matches when they do not have internet access is a nice feature, but we also need online multiplayer feature to have a complete multiplayer experience. Through online gaming players can reach competitors far away and test their skills against each other.

---

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

The initial idea was giving players the possibility to play the game on their devices when they are not at home and have no internet on their mobile device. And when they get home, they could connect to internet with their mobile device, upload their data to the game server and synchronize their game data. Then they could for example switch to a desktop PC and play with people all around the world.

To provide players with this gaming experience that we just described we need to have a server that would be accessible through internet. Here are three things we need from our server:

1. Store the players‘ data on the server, so they backup and synchronize the data across all their devices
2. Host online battles, so players can challenge other players from far away
3. Provide a website that serves as some sort of social hub, online statistics about players and possibly a knowledge base for the game

Certainly we could have a single server for each role, but the most elegant, economical and time-saving solution is to have a single game server which can perform all the mentioned functions.

.NET provides us with many technologies and there is one that is perfect to build our game server on – *ASP.NET* [49]. We will implement a simple ASP.NET application that will be hosted on a web server. This ASP.NET application will use *MS-SQL database* [50] which will both hold players‘ save data and information related to hosting online battles, such as data from individual battle instances.

ASP.NET provides us with a few options to build our game website. They are *ASP.NET WebPages*, *ASP.NET WebForms*, *ASP.NET MVC*. Each of them has different workflow and it does not really matter which we choose. It depends on user’s personal preferences. For our project, we will build a website using ASP.NET MVC framework.

Communicating with Web Server over HTTP

HTTP protocol is not just for serving up web pages. It is also a powerful platform for building APIs that expose services and data. HTTP is simple, flexible, and ubiquitous. Almost any platform that you can think of has an HTTP library, so HTTP services can reach a broad range of clients, including browsers, mobile devices, and traditional desktop applications.

Our server application will be hosted on a web server and players will send requests and messages to the server using their game client application. For example, it might be a message containing data about performed attack in battle or on the contrary it might be a request to fetch some data from the server. Since the server application sits on the web server, we will be using HTTP implement the communication between the client and the server.

For these purposes, we will use ASP.NET Web API framework to build a web API[6]. The game clients will communicate with the game server through this web API.

Duplex Communication over HTTP

Classic communication over HTTP has request-respond character. The client sends a request to the server and the server sends a response back to the client. However, we will need to send messages from the server to the client, in other words push the message from the server to the client. For this purpose, it would be best if the communication was duplex (bidirectional). That way both client and server could freely send messages to the other. The normal communication over HTTP is not duplex.

[6] *Web API* is a programmatic interface consisting of one or more publicly exposed endpoints to a defined request-response message system, typically expressed in JSON or XML. For detailed explanation of what web APIs are, please see [51].

There are two well known ways to achiev duplex communication over HTTP or at least simulate the duplex connection. *Web Socket* [52] is a protocol providing full-duplex communication channels over a single TCP connection. It is a relatively new technology, but most major browsers currently support it. Regrettably, most of the web hosting providers do not offer Web Socket technology, thus we will not be using it for our project.

There are ways to simulate duplex communication over HTTP. They are called *push technologies*[7]. We will be using technique called Long polling. In principle, the client sends a request to the server and instead of the server returning an empty response immediately, the server holds the request open and waits for response information to become available. Once it does have new information, the server immediately sends an HTTP response to the client, completing the open HTTP request.

To summarize things, for offline multiplayer we will be using bluetooth. For online multiplayer we will develop an ASP.NET MVC application with MS-SQL database. We will use Web API communicate with game clients. And to simulate duplex connection over HTTP, we will use long polling technique.

[7] Server push describes a style of Internet-based communication where the request for a given transaction is initiated by the publisher or central server. For more information on push techniques, please see [52].

3. Designing the Game

In this chapter we look into the game designing phase. We shall introduce all important game mechanics with respect to the things that we pointed out in the chapter 1. 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.

3.1. Game Concept

Ninshu Arts is a role playing game. At the start the players create their own character called *Shinobi*[8] 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 3.3.3. 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*[9]. As players progress in game, they obtain and learn new jutsus to have more tools to use in combat.

The readers of Japanese manga[10] series *Naruto*[11] 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.

---

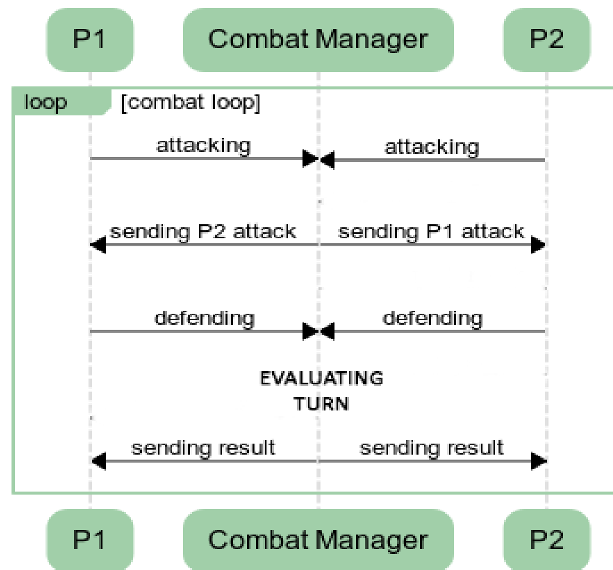
[8] *Shinobi* or Ninja is a term for a covert agent in feudal Japan.
[9] *Jutsu* is a word from Japanese which translated means “technique”, “method” or “spell”.
[10] *Manga* are comics created in Japan or by Japanese creators conforming to a style developed in Japan in the late 19th century.
[11] *Naruto* is a Japanese manga series written and illustrated by Masashi Kishimoto.

3.2. 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.

3.2.1. 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 easy. 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.

3.2.2. 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 3.2.3. 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.

3.2.3. 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.

3.3. 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.

3.3.1. 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.

3.3.2. 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.

3.3.3. 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.

3.4. Game Controls

One of our objectives is to create a multi-platform game. 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 which is the most common way to interact with the game. 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 a 3D renderer 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. However, 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 or on the contrary big buttons take too much place on a small screen.

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.

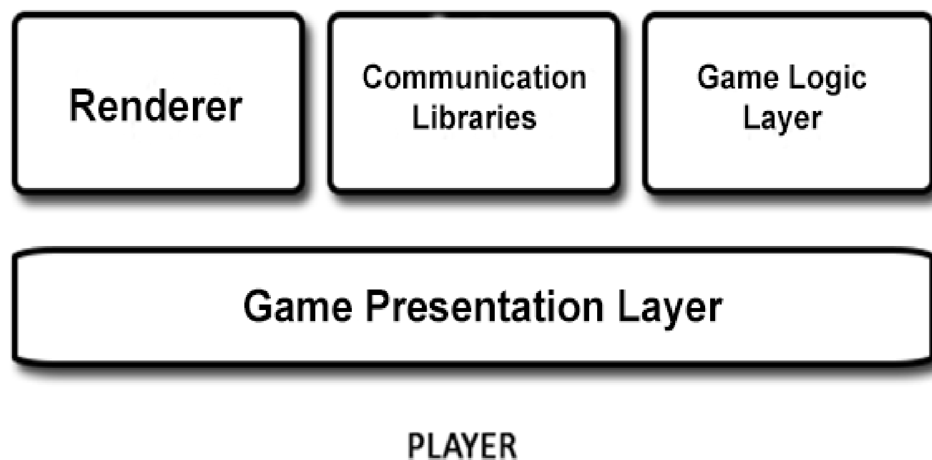
Even if we do not distinguish kicks and punches and take only direction information into account, it would not look very action if characters were doing only punches and simple moves. We need to incorporate both punches and kicks animations into the combo to make a game look action. We could organize 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.

4. Technical Documentation

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

- 3D scene renderer (described in subchapter 4.1.)
- communication libraries (described in subchapter 4.2.)
- game logic layer (described in subchapter 4.3.)
- game presentation layer (described in subchapter 4.4.)

The 3D renderer 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 libraries are responsible for providing bluetooth or HTTP message exchanging. 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

4.1. Writing the 3D Renderer

The 3D scene renderer is a unit responsible for rendering a 3D scene and animating characters. It represents 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. We will be developing a 3D renderer as DirectX application in C# using SharpDX library. In this chapter we describe the process of writing a DirectX 3D graphic application to render a 3D scene and animating characters.

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.

4.1.1. 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[12] applications and WP8 applications. Let us remind the readers that we are developing a universal multi-platform app, which targets all types of devices, not a regular desktop application.

---

[12] *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.

4.1.2. 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 4.1.2., 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).

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[13].

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

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).

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. 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 (this problem will be explained more in detail in subchapter 6.3).

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 renderer 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[14].

4.1.3. 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 create the `Renderer` library which encapsulates the 3D renderer and provides an API to work with it from the outside.

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[15].

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.

[14] 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.

[15] For more detailed explanation of initialization commands and their 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 `D3DApplication.cs` class descending our base class that implement a Direct3D application for our univeral application. The source code of mentioned file is short and fairly self-explanatory on the first look.

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[16] (CMO). It also includes methods providing mesh animation.

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.

---

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

4.1.4. Implementing the 3D Renderer

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 implements texture sampling and computes lighting using *Blinn-Phong light model*[17]. For those readers, who are interested in the code, please see the source files.

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.

[17] *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.

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 3D Renderer to work on WP8 devices.

A question arise - 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 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[18].

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:

---

[18] LoadTexture class uses helper classes located in DDS folder of the Renderer library project. These files were adopted from Alexander Mutel's SharpDX repositories.

```
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` 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()`[19].

And with this we are done with `CreateDeviceDependentResources()` method. Our quad renderer also needs to implement `DoRender()` method. Here is a code snippet:

```
context.PixelShader.SetShaderResource(0, textureView);
context.PixelShader.SetSampler(0, samplerState);
context.InputAssembler.SetVertexBuffers(0, quadBinding);
context.InputAssembler.SetIndexBuffer(indexBuffer,
```

[19] `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.

```
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);
```

To render vertices saved in a vertex buffer we only need to set shader resources and sampler for pixel shader. Then we 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 3D renderer. 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 3D renderer. For this purpose we create a `Mesh` class. 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[20]. 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.

---

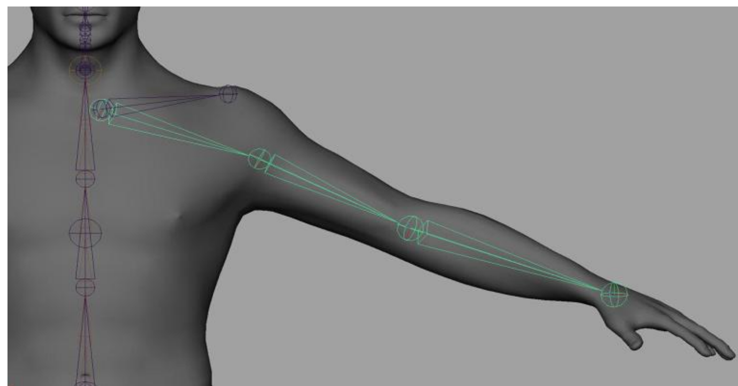
[20] Readers can see the source code of Mesh class for the details about the internal structure of a CMO file or visit DirectX Tool Kit homepage [16] where they can find a CMO file parser written in C++.

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[21].

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.



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

---

[21] 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.

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 `D3DScene` class which inherits from `D3DApplication` 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. For example this is how a `QuarRenderer` is initialized:

```
var q = new QuadRenderer(config.Vertices, config.Texture);
QuadRenderers.Add(ToDispose(q));
q.Initialize(this);
```

As for the `Render()` method of `D3DScene` 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.

4.1.5. 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 3D renderer 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 `D3DApplication`. 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[22].

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

Within the `D3DScene` 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

[22] 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.

we combine this with a simple XAML animation, we get a dynamic loading screen rather than unappealing static splash screen.

4.1.6. 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`.

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.

There are several options to solve our problem. 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. We 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 scene 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, we have to take into account even the low budget smartphones with low performance CPUs and GPUs. One of the issues while testing the renderer 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.

We could precompile our shaders using `fxc.exe` utility which is available in DirectX SDK, for example. Now, the renderer only has to load the precompiled vertex shader and pixel shader and use them.

4.2. Implementing the Communication Libraries

The communication libraries are responsible for any communications that will be used for multiplayer gaming. As we already mentioned before, one of the objectives of this project is to implement a communication library capable of providing the ability to exchange messages between two devices of different types,

i.e. 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. All the bluetooth communications will be implemented in `BluetoothCommunications` library.

Our project also needs other type of communications. We need to be able to exchange messages between game clients and the game server which will be hosted on a web server and accessible through internet. We decided to use HTTP protocol to communicate with the server application which will be ASP.NET MVC application with Web API to communicate with game clients. To simulate the duplex communication, we decided to use long polling technique. The library providing the communications with the server application will be called `WebCommunications`. (The analysis of mentioned technologies and the discussion on communications topic was done in chapter 2 Implementation Analysis, for more detials see section 2.5.)

4.2.1. Bluetooth Communications

We will use *Bluetooth Rfcomm* Windows Runtime API[23] 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`.

We create a new distinct `BluetoothCommunications` library which will provide the bluetooth communications. In this library we have a `Bluetooth.cs` class that encapsulates everything about the bluetooth communication unit.

[23] 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].

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 Initialize()
{
      rfcommProvider = await
      RfcommServiceProvider.CreateAsync(RfcommServiceId.FromUuid(
      RfcommChatServiceUuid));

      socketListener = new StreamSocketListener();

      awaitsocketListener.BindServiceNameAsync(
      bluetoothServer.rfcommProvider.ServiceId.AsString(),
      SocketProtectionLevel);

      // Set the SDP attributes and start Bluetooth advertising
      InitializeServiceSdpAttributes(bluetoothServer.rfcommProvider)
      rfcommProvider.StartAdvertising(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 need to set Service Discovery Protocol (SDP) attributes and start advertising our service and listening for incoming connections. SDP attributes set using `InitializeServiceSdpAttributes()` method. SDP attributes that are used in this method are set when `BluetoothServer` instance is created. We need to pass SDP attributes to the constructor. It is imperative that the SDP attributes passed to `BluetoothServer` are the same that the `BluetoothClient` objects will be using to find advertised services.

Now when a client participant connects to the service we are advertising, `socketListener.ConnectionReceived` event is fired. We usually need to handle this event by displaying a text message to the users, for example. To set an event

handler from outside to the internal `ConnectionReceived` event we use `BluetoothServer.SetConnectionReceived(eventHandler)`. In a method handling the connected event, we need to call `BluetoothServer.Connect()` method to start the communication. The method looks as follows:

```
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 `Connect()` method is quite simple. First, it 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. This is to ensure that we are finding only the advertising bluetooth services that are of our interest. Both bluetooth server and client must have the same SDP attribute.

All we need to do now 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();
}
```

Using the `DataWriter` 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.

Communication Model in Offline 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 processes are done, they start to behave like equal participants in P2P communication and it no longer matters who is the server and who the client.

In the terms of communications two participants pose as equals, but in the terms of battle participants they are not. The player running the bluetooth server will take the role of both the game server and the game client. The other player will be just the game client. That means in a P2P offline multiplayer battle, the server participant 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.

When we implement online multiplayer 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.

4.2.2. Web Communications

Entire communications with the web server will be encapsulated in the `WebCommunications` library. Communicating with a game server has a client-server form, so we will implement `WebServer` class for handling communications on the server side and `WebClient` class for the client side.

Web Server

The most important function of `WebServer` class is to simulate duplex communication between the game server and the game client. For this purpose use long polling technique (as we discussed in section 2.5.2.). To remind the readers, long polling is a technique, when the server does not return the response to the client immediately, but rather hold the request open until it has requested data for the client. Only then it sends back the response with requested data.

Naive solution would be using active waiting, where on a new request the server unit would sleep the thread and checks if the requested data are ready every few seconds. As this solution consumes too much system resources, we will use something different. We will be using `TaskCompletionSource` object, since it provides asynchronous waiting which gives better results than active waiting.

If we look into the code of `WebServer` class, the most interesting thing we find is:

```
ConcurrentDictionary<string,TaskCompletionSource<string>> clients
```

We are using `ConcurrentDictionary` instead of classic `Dictionary`, because `ConcurrentDictionary` collection is thread safe. This dictionary will hold all the pending requests from the clients. Note that there is `string` in `TaskCompletionSource<string>`, because results the `WebServer` will be sending back to `WebClients` will be `string` values.

`WebServer` class then contains 2 methods. The first one is `AcceptClient()`:

```
public async Task<string> AcceptClient(string userID)
{
      bool added = clients.TryAdd(userID, new
            TaskCompletionSource<string>());

      if (!added) clients[userID] = new
            TaskCompletionSource<string>();

      await clients[userID].Task;

      TaskCompletionSource<string> tsc = null;
      clients.TryRemove(userID, out tsc);
      return tsc.Task.Result;
}
```

This method is called when the game server gets a request for some data that are not ready to be sent back to the client. The input parameter `userID` serves as an identifier and a key in `clients` dictionary, so that we can identify exactly which user we want to send the message to. The `AcceptClient()` method also checks the dictionary for duplicity. If we already have pending request from the particular client, we simply delete the last request and replace it with a new one (new `TaskCompletionSource`).

The other method in `WebServer` class is `Send()`:

```
public void Send(string message, string userID)
{
     clients[userID].SetResult(message);
}
```

When we have all the requested data ready, we call `Send()` method on the server which gets a particular `TaskCompletionSource` object from the dictionary, which was asynchronously waiting, and we awake the it by calling `SetResult(message)` with the `message` data that we want to send back to the client.

Web Client

`WebClient` class represents an object that will provide communication with the game server from the client side. First it needs a way to send HTTP requests. For this purpose, we use `HttpClient` object. HttpClient class provides us with everything we need for running `WebClient`, e.g. sending GET/POST requests, sending authorized requests etc.

The first method we will be discussing is `Send()` method:

```
public async Task<bool> Send(string uri, string message)
{
     ...
     HttpResponseMessage response = await
     HttpClient.PostAsync(uri, new StringContent(message,
           Encoding.UTF8, "application/json"));

     if (response.IsSuccessStatusCode)
           return true;
     else
           return false;
     ...
}
```

This `Send()` method is called when we need to send some data to the server and we do not need any data in response. As you can see from the previous code snippet, `PostAsync()` method is called on `HttpClient` object. It sends a POST request to the certain address with a message. Note that we specified the string content as `"application/json"`, since Web API usually works with JSON and XAML. On the other hand, if we need to request some data from the server, we use `Receive()` method:

```
public async Task<string> Receive(string uri)
{
     HttpResponseMessage response = await
           HttpClient.GetAsync(uri);
     // check if everything was OK on the server side
     // (authentication etc.), eg code 200
     if (response.IsSuccessStatusCode)
           return await response.Content.ReadAsStringAsync();
     else
           return null;
}
```

In comparison with `Send()` method, `Receive()` methods sends GET request to the certain address. This is called when we need to receive data from the server. It registers itself on the server and the requests sleeps on the server (long polling) until the server can send back data or we hit request time-out. When we need to ensure, that we get back requested data when calling `Receive()` and avoid time-out problem, we use `ReceivePersistent()` method instead. This method is the same as `Receive()` methods, the only difference is that when getting time-out response from the server, it sends a new GET request.

To make sure the game server attends to only registered users, we need to have some kind of authorization. ASP.NET MVC authentication system works with *Tokens*. The user first needs to log in and provide valid username and password. Then he gets authorization token which he sends with every request that needs authorization in a header. To log in and receive an authorization token, we use `Login()` method:

```
public async Task<bool> Login(string uri, StringContent loginData)
{
      HttpResponseMessage response = await
            HttpClient.PostAsync(uri, loginData);
      if (response.IsSuccessStatusCode)
      {
            Token = await GetToken(response);
            // add the token to the request header

            HttpClient.DefaultRequestHeaders.Authorization = new
                  AuthenticationHeaderValue("Bearer", Token);
                        return true;
       }
       else
            return false;
}
```

As we can see from the code snippet, we are sending a POST request to the certain address used for authentication. We are sending username and password in `StringContent loginData` which need to have a specific form, if we are working with ASP.NET MVC. loginData need to be in "`application/x-www-form-urlencoded`" format. For example, it might look like this:

```
"grant_type=password&userName=myUserName&password=myPassword"
```

If the provided username and password were valid, the server returns a token which is then stored in `HttpClient.DefaultRequestHeaders`.Authorization. After this header is set, all the calls are authorized.

Using WebServer and WebClient with ASP.NET MVC

After importing `WebCommunications` library we have access to our `WebServer` and `WebClient` classes. WebServer will be used in our ASP.NET MVC project, where it will help manage requests. `WebClient` will be used in our game client to send requests to the game server.

As me mentioned earlier, our ASP.NET application will implement a Web API available to public. In this Web API we will have public addresses, each address for each action our game server provides. For example, our server will receive log in requests, accepting messages containing data on executed attacks in battles, requests to get result of the battle turn, etc. Here are examples:

```
http://baseaddress.com/api/Account/Login
http://baseaddress.com/api/Battle/SendAttack
http://baseaddress.com/api/Battle/GetResult
```

If we wanted to send a message containing data on executed attack, we would use the `WebClient.Send(uri, action)` method where `uri` would be `/api/Battle/SendAttack`. It is the address where the server accepts these kinds of requests. The logic on processing individual requests in coded for each action in ASP.NET application. We will me talking more in detail about the actual ASP.NET application later in subchapter 4.5.

When we need the `WebServer` to push a message to a client, the client needs to be listening for the response first. This is done by calling `WebClient.Receive(uri)`, where `uri` could be `/api/Battle/GetResult`. On this address the server accepts the requests for waiting clients and calls `WebServer.AccaptClient()`. When the data are ready to be sent back to the client, it we call `WebServer.Send(data, userId)` and the client receives the data through `WebClient.Receive()` method.

4.3. Game Logic and Features

In this subchapter we will look into implementing the game logic and game features. 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`.

4.3.1. 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. 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`[24] 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);
}
```

[24] *ApplicationData* is a class providing access to the application data store [19].

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[25]. We are using this framework to convert player object into *Javascript Object Notation*[26] (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. All manipulations with players‘ save data is encapsulated in `PlayersData` static class which also takes care of saving and loading saved data on online game server.

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.

4.3.2. 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 do not have to be combined in one person. If that is the case, we should provide the possibility for 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.

---

[25] *Newtonsoft Json.NET* is a high-performance JSON framework for .NET [20].

[26] *JSON* is a lightweight data-interchange format [21].

We propose a method where the data on a certain jutsu or a mission is stored in an external Extensible Markup Language[27] (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[28] 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.

Note: On our project we are working with both XML and JSON formats. The reason for that is we only use XML format to store data that could come from game designers (people with no programming skills), because XML is easily readable than JSON. We use JSON format for everything else, especially for exchanging messages in communications.

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.

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.

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>
```

[27] *XML* is a simple and flexible text format derived from SGML (ISO 8879) [22].

[28] *Google Sheets* are Sheet files stored on Google Drive (cloud storage) [23].

```
    <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:

```
[XmlRoot("jutsu")]
public class Jutsu
{
        [XmlElement("id")]
        public int ID { get; set; }

        [XmlElement("name")]
        public string Name { get; set; }

        [XmlElement("hitcount")]
        public int HitCount { get; set; }

        [XmlArray("hittimes")]
        [XmlArrayItem("time")]
        public List<double> HitTimes { get; set; }

        [XmlElement("cooldown")]
        public int Cooldown { get; set; }

        [XmlElement("effect")]
        public CombatEffect Effect { get; set; }

        [XmlElement("effectvalue")]
        public int EffectValue { get; set; }
           ...
}
```

Please notice the `XmlRoot`, `XmlElement` keywords. We use `XmlSerializer` class to deserialize our data which are stored on external files.

4.3.3. Combat Manager

Combat manager is a unit responsible for managing the battle. 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.

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

```
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.

The `CombatManager` is a static class that takes the data executed in one turn and calculates the result to move on to the next turn. It is created as a static class, because we want to separate data on the state of battle in a different object, that is `BattleInstance`. This will be useful when we move the domain of offline and move to online gaming. The game server will retain a lot of battles which will be stored in individual `BattleInstance` objects. Then `CombatManager` is called to process the battle instances that are ready to evaluate and ready to move to next turn.

Players send their attacks to the combat manager/to the battle instance using `ActionJson` objects:

```
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. The combat manager's most notable method is `EvaluateTurn()`. We will not be describing it here in detail, because it includes calculations with numbers mostly which is not very interesting. Basically, it tries to incorporate all the mechanics that we introduced in the game design chapter. It takes players's stats, jutsus that were used, jutsus info, elemental types and others. It then edits the cooldowns on techniques and evaluates status change on players in order to move to the next turn.

4.3.4. Bluetooth 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);
```

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:

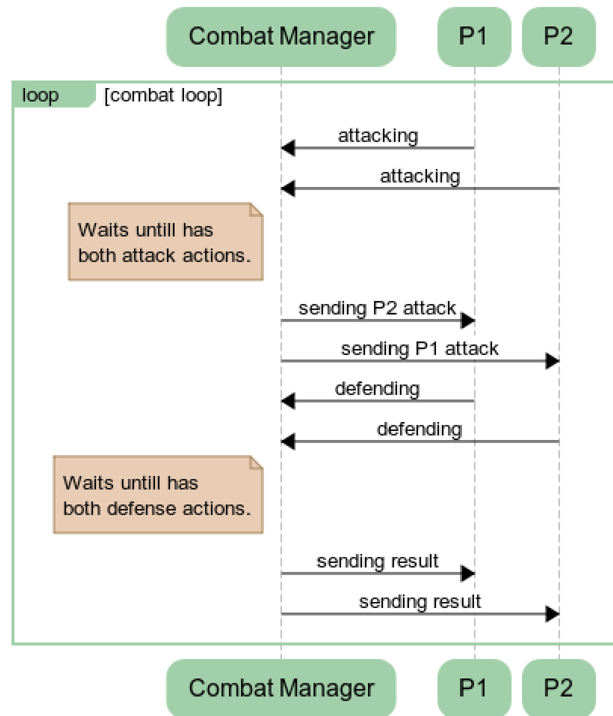
```
private async void ClientCombatLoop()
{
     while (true)
     {
           // ATTACK PHASE
           // ------------
           // receive attack action of P1
           string json = await bluetooth.Receive();
           ActionJson attackJson =
           (ActionJson)JsonConvert.DeserializeObject(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

4.3.5. Sending Shinobi on Missions

In this section we discuss how to implement the feature of sending a shinobi on a mission. 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.

4.3.6. Hunting and NPCs

The Hunting feature for Ninshu Arts can use a lof of the code we already wrote. 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.

4.4. Creating the GUI

In this subchapter we will briefly go over game presentation layer. It is what connects players with the inner game logic layer where all the game features and mechanics are. To create a GUI for our game, we will use XAML[29]. 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.

4.4.1. GUI and XAML

We will only go over this part quickly, since XAML is not very difficult and many things are apparent at first glance.

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.

Placing and positioning XAML elements to create a complete XAML page is done using XAML layouts[30]. 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.

As all the screens are written in a similar way, we will only provide the code snippet of `Village.xaml` page as an example. Other XAML pages were created in a similar way.

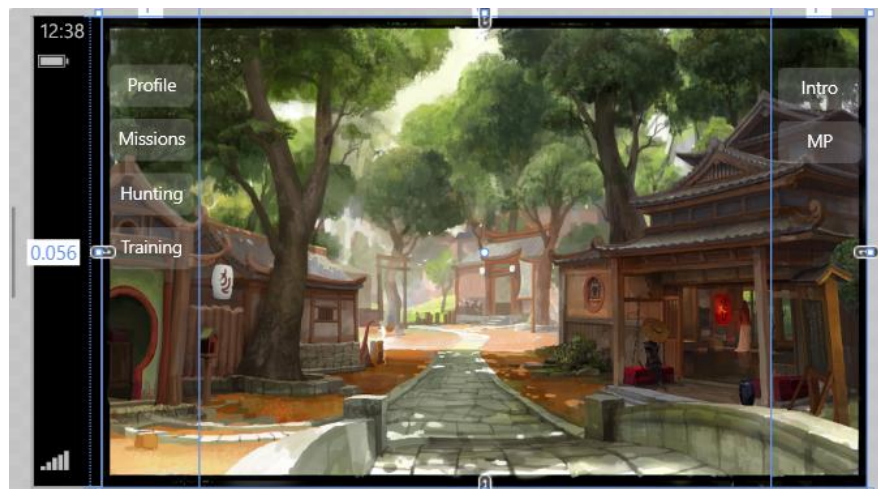
```
<Grid Background="{ThemeResource
        ApplicationPageBackgroundThemeBrush}">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="1*"/>
        <ColumnDefinition Width="6*"/>
```

---

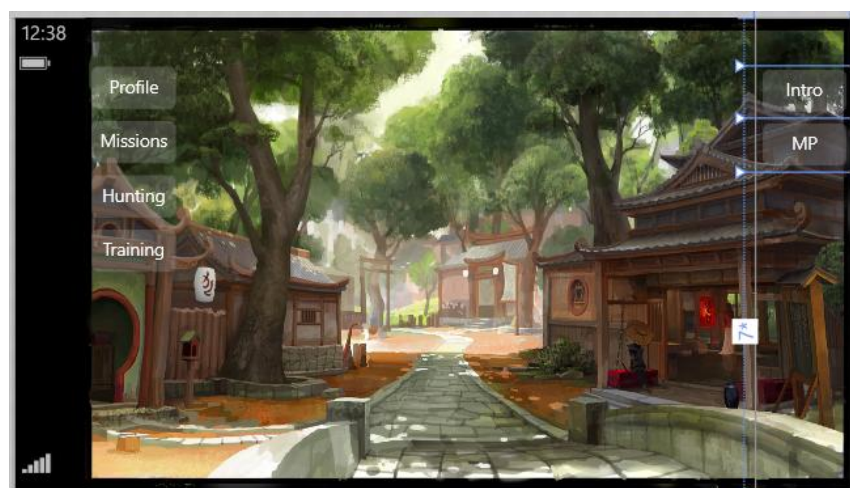
[29] We will not be explaining basics of XAML here. It is fairly easy to understand after reading first chapters of almost any tutorial found on internet. For those readers who are more interested in learning XAML, we recommend the book Building Windows 8 Apps with C# and XAML by Jeremy Likness [27].

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

```
            <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>
```



Picture 6.: Grid columns



Picture 7.: Grid rows

4.4.2. Battle Screens

Let us now talk about a little more interesting matter that is a battle screen. In this project we implemented three battle screens, `MultiplayerBattle.xaml`, `SingleplayerBattle.xaml` and `ArenaBattle.xaml`. The three are very similar.

We will only describe the `MultiplayerBattle.xaml` page here since the other are 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 4.1.5. we were talking about combining a Direct3D application into XAML. We said back then that we are going to integrate our 3D renderer 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 `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 renderer. This is used to draw a trail after swiping over the screen by finger or mouse.

4.5. Game Server

In this subchapter we will discuss how our game server was implemented. To remind the readers, we are implementing ASP.NET MVC application with Wep API and we use MS-SQL database to store our data.

We are going to ignore the website part of our ASP.NET application, since it is not relevant to our work. We will design a simple database to keep the data that we need and we will be focusing primarily on Web API which processes requests from game clients manages the communication between the game server and clients.

4.5.1. Basis and Creation

The game server will be hosting games. The server will hold the instances of battles in progress. All playing clients will be sending their executed attacks and defences to the server which will assign the attack/defence actions to the right battle instance and when the server has all the needed data to in a battle turn, it evalues the turn and then sends results to appropriate players.

The game logic is all coded in our `GameContent` library, specifically in our `CombatManager` class which responsible for all combat data evaluating. Our ASP.NET application will only include the `GameContent` library and it has access to the `CombatManager`.

We will create a new *ASP.NET Web API* project to our solution in Visual Studio 2013. This will create a preset application that provides with a few things right from the start. First, it creates a default website which has some info pages that are really not that important at the moment. We are going to ignore it.

More interesting that is already done in the preset of our newly created ASP.NET application is *Identity*[31] membership system. It is very convenient to use, because after the first registration all the tables related to membership system are created. This is done thanks to *Entity Framework*[32] which is typically paired with ASP.NET MVC. This includes tables for Users, Roles and others.

---

[31] *ASP.NET Identity* system is designed to replace previous ASP.NET Membership and Simple Membership systems, for more information please see [53].

[32] *Entity Framework* is a set of technologies in ADO.NET that support development of data-oriented software applications, for more information please see [54].

4.5.2. Game Database

Our server application needs to store two types of data. First, it is players' data. Players need to be able to send their saved data to the game server in order to backup and synchronize their save to other devices. Second, the server application needs to store data on battles in progress, since HTTP communication is stateless. For each type, we will have one table in our database – *Players* and *Battles*.

`Player` and `Battle` entities can be found in `Models/ArenaModels.cs` file. Internal structures of entities correspond to the collumns in respective tables, i.e. the fields in Player entity corresponds to collumns in Players table.

Players

Table Players contains the data of Ninshu Arts players. It is in one to one relation with table Users (which belongs to Identity membership system). Table Players will have a foreign key from table Users. An entry in Users table corresponds to one entry in Players table. It means that each user has one entry in Players table and one place to store their save data. Here you can see a code snippet from `Player` entity:

```
public class Player
{
     [ForeignKey("ApplicationUser")]
     public string ID { get; set; }

     public string Data { get; set; }
     public int Rating { get; set; }
     public DateTime Queue { get; set; }
}
```

The `int Rating` field tells us what rating a player has. This will later help us to create statistics to provide players with information on how strong particular players are. The `DateTime Queue` is a time when players enqueued themselves for online match. This is used in matchmaking system to find players their opponents. Finally the `string Data` contains converted players's saves into JSON format.

Battles

Battles table contains stored data on battle instances. A single battle instance corresponds to one row in Battles table. For each battle instance we need to store following data:

```
public class Battle
{
        public int ID { get; set; }

        public string Player1ID { get; set; }
        public string Player2ID { get; set; }

        // additional data
        public string Shinobi1 { get; set; }
        public string Shinobi2 { get; set; }
        public string Attack1 { get; set; }
        public string Attack2 { get; set; }
        public string Defend1 { get; set; }
        public string Defend2 { get; set; }
        public string Turn { get; set; }
}
```

For each battle we need to store `Shinobi` objects for both players. Each `Shinobi` object contains data on current condition in battle, i.e. how many HP they have left, status condition, cooldowns etc. Then there are attack and defend actions. These are `ActionJson` objects sent from players. Each time a players sends an executed attack/defence action, it is stored to the right battle instance in these fields. This one row in Battles table contains data on a specific battle in a particular turn, so we easily are able to tell the state of a battle in progress.

After entities we need to discuss database context. Database contexts in ASP.NET MVC is responsible communication between the web application and the database. We have defined our database context as follows:

```
public class ApplicationDb
{
     public DbSet<Player> Players { get; set; }
     public DbSet<Battle> Battles { get; set; }
     ...
}
```

The `DbSet` objects correspond to the tables in the database. While working with the context, we use `ApplicationDb.Players` or `ApplicationDb.Battles` to work with the particular table. Lastly, you will find `OnModelCreating()` method

which is called when the tables in the database is being created. This method states the relations between tables, specifies additional information of individual fields etc.

4.5.3. Web API

Let us now discuss the Web API of our server application. Web API manages the communication on the server. Web API consists of *Controllers* and *Actions*. Usually actions that have something in common are grouped in a single controller. For example, all our actions which concern battles will be in our `BattleController` class. Our `BattleController` has `SendAttack` action for instance. The address that players would be sending their attacks data to would be:

```
http://baseaddress/api/Battle/SendAttack
```

Base address depends on where the ASP.NET application is hosted. The "Battle" in the uri corresponds to the name of `BatteController` and "SendAttack" to the action defined in the controller. The address looks like this, because it was defined in `App_Start/WebApiConfig.cs` file. In ASP.NET it is called *routing*.

We just clarified that each controller has some actions defined. Let us look at our `BattleController`. It provides following actions:

- *Send attack* – action that processes attacks sent from players. It identifies the player that sent the attack action and stores the attack into a appropriate row in Battle table. Then it checks if the processed battle has both attack actions. If it does, it sends enemy attacks to both players to move on to defense phase of the battle.
- *Send defense* – action that processes defences sent from players. It also identifies the player that sent the defence action and stores data in the database in appropriate battle. Then it checks if the processed battle has both defence actions. If it does, it calls `CombatManager` to evaluate the turn. It then uses `WebServer.Send()` method to send turn results to both players. Calling `WebServer.Send()` method will release the asynchronous waiting in `GetResult()` action.

- *Get result* – action that accepts players that are waiting to get the result of a turn in battle. Usually the server will not have the turn results immediately. This is where we use `WebServer` object to hold the requests open by calling `WebServer.AcceptClient()`. The request is put on hold (asynchronous waiting) and we push the results to clients after the server evaluated the data.

Our `BattleController` class looks as follows:

```
public class BattleController : ApiController
{
     private ApplicationDb db = new ApplicationDb();
     private static WebServer Server = new WebServer();

     // actions definitions
}
```

The `ApplicationDb` is a database context that provides us the connection to our database, so we can make queries. Then we have our `WebServer` object to help simulate duplex connection when needed.

Besides `BattleController` we also have `QueueController` which manages queue system. Using queue system are players able to find opponents for online battle. It has following actions:

- *Enter* – action when players enter a queue to wait for opponents. If there are no players waiting, it enqueues the player by setting time they entered the queue in the `Player.Queue` field in the Players table. If there are players waiting, it takes a player from the queue and matches him with the new player. Then a battle instance is created and both players will receive data on started battle.
- *Leave* – action when player leaves a queue. It looks up the player in Players table and sets the `Queue` time to default value.

Last noteworthy controllers are `AccountController` and `PlayerController`. The first processes actions regarding the user account such as registering a new user, password change etc. and the other contains actions for saving and loading players' saved data.

4.5.4. Testing environment

All the development and testing of the server application was done in Visual Studio 2013. Other needed and used tools are Microsoft SQL Server 2014 for running an MS-SQL database on a local work station. We also used Microsoft SQL Managment Studio to easily execute the initial database setup.

Since two Windows Store applications cannot run on one device, we needed to test them on two different devices. In order to debug the server application on one device in a local network and be able to access the server application from another device in the network, we needed to setup the local IIS server. The server application is then deployed to the local IIS and is accessible to all computers in a local network.

In this simulated environment, we performed stress test. Our estimate on players count that are using our game server is hundreds to thousands. The server application managed the stress test fine and was able to hold hundreds till thousand open requests and process them.

When the server application is deployed on a local IIS and is accessible on a certain address, for example `http://10.0.0.20/NinshuServer`, this uri needs to be set to the `WebServerBaseAdress` constant in `GameContent/Constants.cs` file. If this is not set correctly, the game client will not have server to connect to.

5. User's Documentation

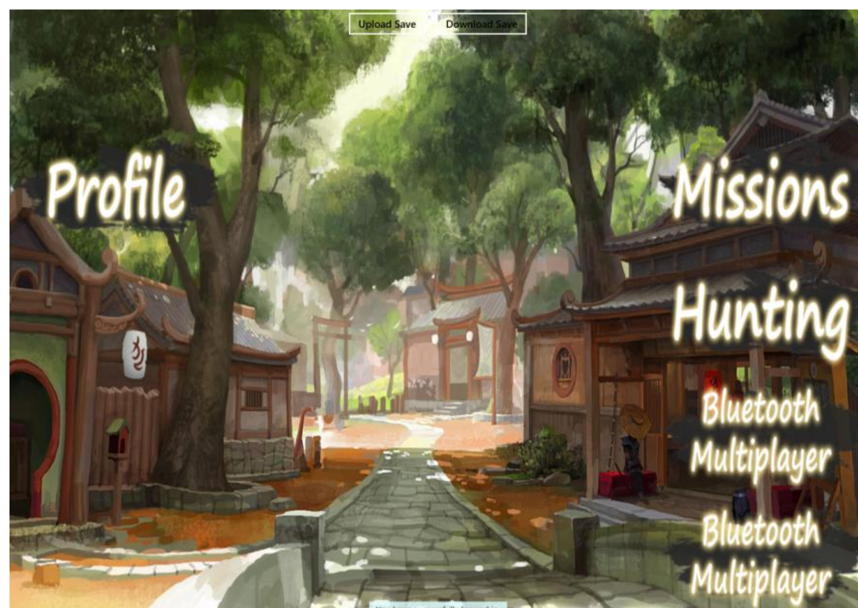
This chapter will work as a game manual for the players. We describe all the game screens here. After reading this, the reader should be able to tell what to find where and have an idea how things work. Chapter 4 along with chapter 2 should provide all the necessary information for players to understand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

5.1. 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.

5.1.1. 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

5.1.2. 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

5.1.3. 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.

5.1.4. 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.

5.2. 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.

5.3. Bluetooth Multi-player

The bluetooth multi-player (MP) battle is fought the same way that was described in subchapter 5.2. 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.

Caution: 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.

5.3.1. Pairing the devices for Bluetooth MP

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 and your bluetooth settings windows is closed, 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.

5.3.2. 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.

5.3.3. Bluetooth 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. On PCs go to bluetooth settings and check „“Allow bluetooth devices to find this computer“

5.4. 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 long 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 and Windows 10.

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.

5.5. Online Features

On the Village screen you can see the login panel on the top. Players use this panel to register and log in to the online game server. After players are logged in, they are able to:

- Upload saved data to the game server to backup as well as to synchronize the save to their other devices.
- Download saved data from the game server to the local device
- Online multiplayer – the button appears after players log in. After clicking this button players are redirected to Arena Lobby page where they enter a queue to look for opponents that are also playing online. When the game server finds an opponent, the waiting player is moved into a battle scene where the fight begins.

6. Making the Game

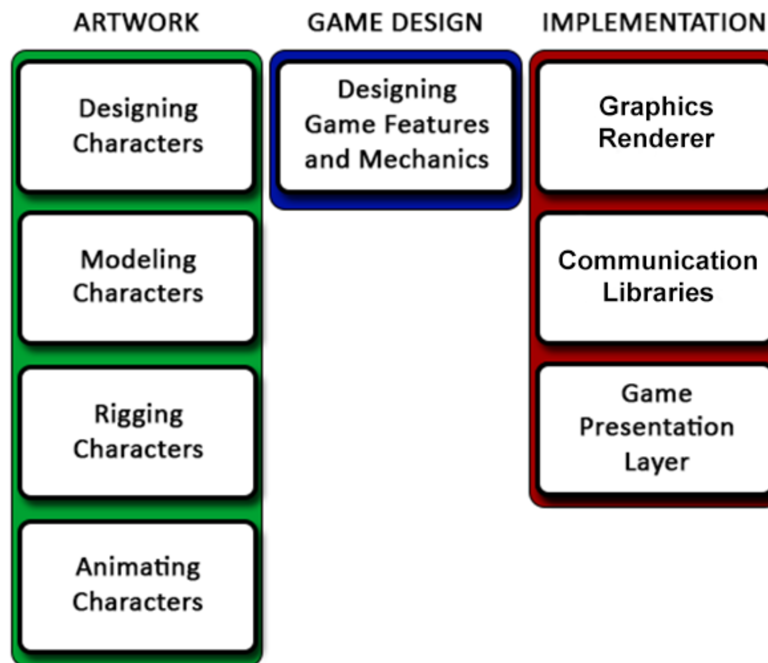
In this chapter we are going to sum up the process of making the whole game. Creating a new game from nothing might not look complicated to someone, but in reality it is a very complex and difficult task where one needs to focus on many different things. It is not a task that you could do on your own.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

6.1. 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 our development process might differ from the process of other development teams, we 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. 3D renderer
    b. Communication library
    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 renderer. 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 6.2.

The stages 2. and 3. of the development process is actually what we covered in chapter 3 (game design) and chapter 4 (game implementation). We will not be going over those again. However, we will point out some issues that we ran into while developing Ninshu Arts. These will be discussed later in subchapter 6.3.

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

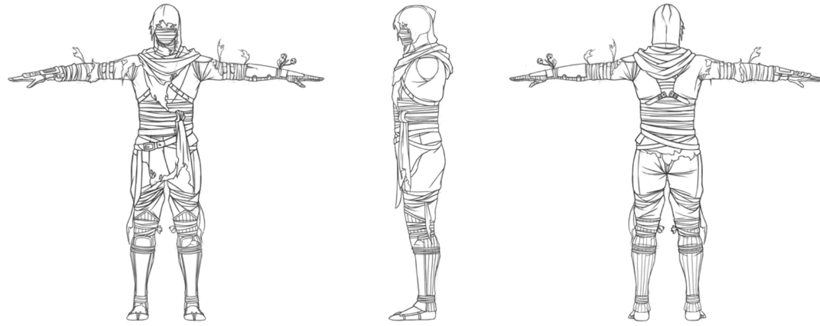
6.2. Artwork Phase

The artwork phase stands 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, we will try to describe the process of coming from character drawings to the point where you have rigged and animated character models.

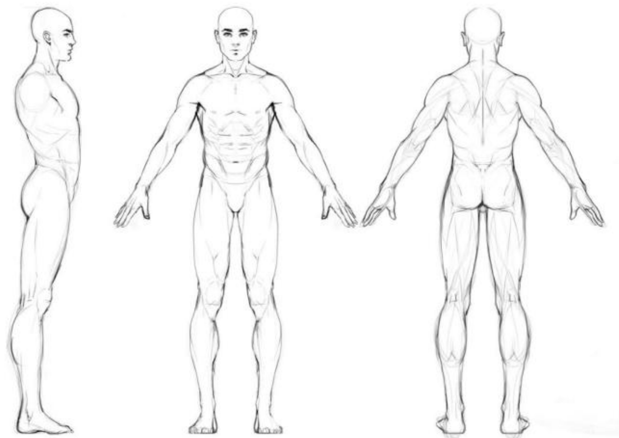
6.2.1. Drawing Character Concepts

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, this step was skipped, because we could not find suitable artists to colaborate with.



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



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

6.2.2. 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, we were not creating models from concept drawings, because we were not provided by the drawings from any artist. 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[33].

6.2.3. 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 renderer 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, we personally rigged 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[34] offers many services including auto-rigging which works surprisingly well. However, these are paid services.

6.2.4. 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

---

[33] Free models were downloaded on TF3DM [42].

[34] 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.

In the process of developing Ninshu Arts we tried a little different approach. All the three mentioned applications are based on moving the bones and working with keyframes to create the animation. We were trying to animate characters using motion capture[35] (mocap) and using the data from mocap to map it on the 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[36]. 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.

6.3. 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.

---

[35] *Motion capture* is the process of recording movement of objects or people [37].

[36] *Kinect* is a sensor developed by Microsoft [38].

While we were working on the 3D renderer, we 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 4.1., when we were implementing the game 3D renderer. In order to map the devices that we could eventually target, we did a little research and testing on different types of devices. We 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 2.: 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 3D renderer. Virtually all the smartphones with Windows Phone 8.1 OS can use our 3D renderer with slight limitations that were mentioned in subchapter 4.1. due to the support of feature level `9_3` and SM2. We 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[37] 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 we were not able to run the 3D renderer on

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

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

6.4. Future Development

Ninshu Arts project is only a starting point for a much larger project that we plan to extend 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. In-game online social hub (online village)
2. Multi PvP battles with more than two players (2v2, 3v3)
3. Implement more visual effects in the 3D scene renderer

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 (Windows based) you play was just a first step. Having an online village which could function as a social hub and 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. Team battles might introduce a totally new gaming experience even for vetaran players. The possibilities and combinations of outcomes is much greater. That is a feature 3 in the future development list.

The previous points were only concerning 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. To provide the players with effects that would catch their eyes is also important.

6.4.1. Things That Were Left Out

In this section we would like to mention a few things that were 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*[38] engine was created in Ninshu Arts in order to provide 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.

[38] *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

At the end of this thesis we would like to recapitulate the goals of this project and summarize the things that were done. In this work we were supposed to analyze the current trends in the gaming world. With this analysis and some inspiration taken from a few existing titles we formulated the game that we wanted to create. We decided to develop a multi-platform game with emphasis on multiplayer features. Our next goal was to analyze available technologies and tools relevant to game development and to our game. The next objective was to develop the game from the ground. In the process of making the game, we covered almost all the stages of game development, including artwork, game design, game implementation and testing. As we covered so many areas, we set our last objective to be providing some information on non-programming activities of the development and summing up the overall development process.

In the first chapter of this thesis we got some inspiration analyzing some great game titles. After that we decided to implement an RPG game providing both offline and online multiplayer capabilities. Offline communication was supposed to be implemented using bluetooth technology, while online gaming was supposed to feature a game server application which would enable players to play online. We also decided to follow the gaming trend and make a multi-platform game, meaning a game that is able to run on different types of devices such as desktop PC, notebooks, tablets and smartphones. After making our game features clear, we also performed an analysis of existing favourite game engines. After the discussion we decided to build our game from the ground. We targeted Windows platform, since its Universal Application model enables targeting of all the types of devices that we mentioned.

While developing the game from the ground, we focused a lot of our effort on creating a 3D renderer unit which provides rendering of simple scenes with objects and animated characters. As a result we have a simple renderer which can be reused in other projects for Windows Store applications. Then we created two communication libraries. One provides bluetooth communication for cross-device communication on various devices with no need of local network or internet. The

other provides HTTP communication between a client and a server applications. Then we implemented the game itself featuring the game mechanics introduced in our game design. Lastly we created a server application that provides data storage for players to backup and to synchronize their game data across their devices. The server also hosts battles and offers online playing.

The overall process of game development is a demanding one. The result of this fact is that the Ninshu Arts might not look as nice and beautiful as first-rate games. This is mainly caused by the lack of artwork, drawings, new models and animations, since they provide the effects that catch the eye. In addition, most of the server application testing was done in a simulated environment on a local network. The game itself was released to beta testing a while ago, but the game server still needs to undergo large scale testing on a real server. Nevertheless, the game is in a stable state and it provides a solid ground to build new things on. Ninshu Arts is just a starting point of a much larger project.

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

[43] Spacewar! (video game) article on Wikipedia - https://en.wikipedia.org/wiki/Spacewar_(video_game), Last checked 2015-11-28

[44] Super Mario 64 article on Wikipedia - https://en.wikipedia.org/wiki/Super_Mario_64, Last checked 2015-11-28

[45] Tomb Rider article on Wikipedia – https://en.wikipedia.org/wiki/Tomb_Raider, Last checked 2015-11-28

[46] Paradox Engine official web page - http://paradox3d.net/, Last checked 2015-11-30

[47] Wave Engine official web page - https://waveengine.net/, Last checked 2015-11-30

[48] Xamarin official web page - https://xamarin.com/, Last checked 2015-11-30

[49] Microsoft ASP.NET development community - http://www.asp.net/, Last checked 2015-11-30

[50] Microsoft SQL server article on Wikipedia - https://en.wikipedia.org/wiki/Microsoft_SQL_Server, Last checked 2015-11-30

[51] Web API article on Wikipedia - https://en.wikipedia.org/wiki/Web_API, Last checked on 2015-11-30

[52] Web Sockets article on Wikipedia - https://en.wikipedia.org/wiki/WebSocket, Last checked on 2015-11-30

[53] ASP.NET Identity - http://www.asp.net/identity, Last checked on 2015-12-02

[54] Entity framework article on Wikipedia - https://en.wikipedia.org/wiki/Entity_Framework, Last checked on 2015-12-02

[55] Android official pages - https://www.android.com/, Last checked on 2015-12-04

[56] Apple official pages - http://www.apple.com/, Last checked on 2015-12-04

List of Tables

Table 1. – Game engines and supported target platforms and whether they are free to use or royalties are involved, (page 17)

Table 2. – Tested devices with their supported feature levels, shader model and shader profiles, (page 108)