Developing a Workflow for Cross-platform 3D Apps using Game Engines

Linus Håkansson & Filip Larsson

Master's Thesis

Department of Design Sciences
Lund University
ISRN:LUTMDN/TMAT-5171-SE

EAT 2013

Developing a Workflow for Cross-platform 3D Apps using Game Engines

(Master Thesis)

Linus Håkansson
ic08lh1@student.lth.se

Filip Larsson
ic08fl7@student.lth.se

June 18, 2013

Master's thesis work carried out at
the Department of Design Sciences, Lund University.

Supervisor: Joakim Eriksson, joakim.eriksson@design.lth.se

Examiner Gerd Johansson, Gerd.Johansson@design.lth.se

Abstract

Cross-platform developing is not a new approach. However, considering it is common for developers to release an application exclusively for a single platform, the use of cross-platform developing is noticeably low. In this master's thesis, aspects regarding the development of a cross-platform 3D application is examined and discussed. A comparision between different motion capture systems used for character animations is presented along with a pipeline for the process of creating a character to be used in a mobile application. This thesis also provides guidelines and recommendations for independent game developers.

Keywords: Unity, Kinect, cross-platform, Qualisys, game

Acknowledgements

We would like to thank the Lund University Humanities Lab (Humlab) and especially Carolina Larsson and Stefan Lingren, for guidance with- and access to the Qualisys Motion Tracking System. We would also like to thank our supervisor Joakim Eriksson for guidance and support throughout the thesis. For helping us with iOS testing related issues, we would like to thank Mattias Wallergård. Finally, we would like to thank our girlfriends Linn and Emelie for their love and support throughout the thesis and for Emelie's contribution of art and design to the user interface.

Contents

1 Introduction 7
  1.1 Purpose and Problem Description 8
  1.2 Outline 8

2 State of the art 9
  2.1 Platforms 9
    2.1.1 iOS 9
    2.1.2 Android 10
  2.2 Game Engines 11
    2.2.1 Licencing 12
    2.2.2 Unity 14
  2.3 Animation 18
    2.3.1 Motion Capture 18
  2.4 Character Design 21

3 Design and Implementation of an Example Collection 27
  3.1 Game functionality 27
    3.1.1 Mini games 28
    3.1.2 Matchmaking and User Community 34
  3.2 Cross-platform development 36
    3.2.1 User interface 36
    3.2.2 Server architecture 37
    3.2.3 Deployment 39
    3.2.4 Performance 40
  3.3 Character Design 41
    3.3.1 Base modelling 41
    3.3.2 High polygon modelling 42
    3.3.3 Retopology 42
    3.3.4 UV mapping 43
    3.3.5 Texturing 44

3.3.6 Rigging and weights 44
3.3.7 Other Software 45
3.4 Animations 48
3.4.1 Motion Capture 48
3.4.2 Mecanim/Animation transitions 56

4 Conclusion and Discussion 61
4.1 Animation 61
4.1.1 Mecanim 63
4.2 Cross-platform Development 63
4.3 Character Design 65
4.4 Future Work 65
4.5 Final Conclusions 65

Bibliography 67

Chapter 1
Introduction

A common problem for developers today is the extra effort, knowledge and time needed to develop an application or a game that runs on several different platforms. This often results in a single platform release, which leads to the product only being able to reach out to a limited part of the target group. A common example of this phenomenon is to be found in the smartphone application market where the two biggest platforms Google Android and iOS hold 75 % respectively 17.3 % of the market (however these percentages do not represent the annual income of app sales) [1].

Cross-platform developing is not a new approach. However, considering it is common for developers to release an application exclusively for a single platform, the use of cross-platform developing is noticeably low. There is a big variety of cross-platform 3D engines available on the market such as Unreal Development Kit, Unity and ShiVa3D.

An important part of 3D game development is animations, and more specifically character animations. Two of the greatest difficulties when integrating the animations in a game engine are generating graphically smooth transitions between different animations and also the advanced scripting that is required for the different animation state transitions. However, Unity recently released Unity 4. Unity 4 comes with the new animation system called Mecanim, which has built in functions that are meant to simplify the tasks mentioned above.

When creating character animations in a third-party program, a common approach in both the video and the game industry is to record an actor's movement and apply it to the virtual character. There are different ways to record such movement. The more advanced techniques require a lot of expensive equipment, such as a passive or active marker system, which results in a higher quality. Another cheaper and more primitive option is to record the movement using a semi-passive imperceptible marker, for example with a Microsoft Xbox 360 Kinect.

Creating a 3D character for a game is a very time-consuming process, which

requires a lot of practice and skills using a lot of applications. For an independent (indie) game developer this can be a very expensive development part, since it requires a good designer working full time with the character.

It should be noted that material created in this thesis will only be used for educational purposes and are never to be used for any commercial reasons whatsoever.

1.1 Purpose and Problem Description

The main purpose of this thesis is to develop a network cross-platform 3D application using the game engine Unity. Another purpose is to provide guidelines and recommendations that could be used either by an educational institution or by indie game developers facing economical or efficiency related issues discussed in this thesis. This thesis will also investigate and answer the following questions, related to the problems described in the introduction:

- What pipeline is the fastest, most efficient and most economical to develop a fully rigged character?
- What difficulties occur and what differences exist that are related to cross-platform development in general and in Unity?
- How does a Microsoft Kinect compare to a Qualisys system for motion capture, when recording character animations?
- Is Unity's Mecanim a good tool for managing different animation transitions?

1.2 Outline

Chapter 2 gives a brief explanation of the major subjects which are discussed in this thesis, such as multiple platforms, game engines, motion capture techniques and character creation. How these subjects were dealt with and how the implementation of the application proceeded is explained along with presentation of results in Chapter 3. In Chapter 4, the questions asked in the problem description is answered and the implementation process of the application discussed, before the final conclusions are being made.

Chapter 2
State of the art

2.1 Platforms

A platform is an underlying computer system which allows an application to run. Windows and OS X are examples of platforms on personal computers. For mobile platforms, Android, iOS and Windows Phone are the main competitors on the market. This thesis will focus on the two mobile platforms Android and iOS.

2.1.1 iOS

iOS is a mobile operating system developed by Apple Inc. powering devices such as the iPhone, iPad and iPod touch which are also created by Apple. By creating both software and hardware Apple obtains a greater control over the performance of their devices, making sure the software works perfectly.

In order to develop applications for iOS, the iOS SDK (Software Development Kit), which was released in February 2008, is needed. The SDK itself is free to download, but in order to release applications for the iOS, enrolment in the iPhone Developer Program is needed. This will cost US$99 as an enrolment fee once a year. The SDK can only be installed on a Mac running Mac OS X[2]. Free development and testing is limited to an iOS emulator on the Mac. If the application is to be tested on a physical device, the enrolment in the iPhone Developer Program is required.

Some of the different devices which use iOS have few different resolutions, shown in Table 2.1.

There are big differences in performance between the old and the latest devices. The oldest devices are not supported in modern game engines such as Unity. More details about the performance of the different devices can be seen on http://www.iphonebenchmark.net/.

Table 2.1: Screen resolutions for different devices running on iOS

device	pixel resolution
iPhone/3g/3gs and iPod touch 1/2/3	320 x 480 pixels
iPhone 4/4s and iPod touch 4	640 x 960 pixels
iPhone 5 and iPod touch 5	640 x 1136 pixels
iPad /2/Mini	1024 x 748 pixels
iPad 3/4	2048 x 1536 pixels

Devices running on iOS only has one single physical button. In order to navigate in an application running on iOS, virtual buttons are needed. According to the iOS Human Interface Guidelines, a navigation bar enables navigation through different views and, optionally, management of the content of the views. In the navigation bar, a back button should be located in the top left corner. The back button should be labelled with the previous view's title. In the centre of the navigation bar there should be a title corresponding to the current view. The title should change to the new view's title if the view is changed. The navigation bar should only contain the back button, the current title of the view and a control that manages the view's content [3].

Figure 2.1: iOS navigation bar

2.1.2 Android

Android is an operating system designed for mobile devices. It was developed by Android Inc. and later bought by Google. Android is an open platform, making it possible for other companies to use and customise the operating system in order to suit the companies hardware. For creating applications to be run in Android, the programming language Java is mainly used.

Since there is a wide variety of hardware which uses the Android operating system as its platform, there is a lot of different resolutions running on different devices. The most common resolutions are listed in Table 2.2, however, there also exists other resolutions which are not present in the table.

Android applications can be developed on Windows, Mac OS X and Linux. Developing an Android application requires the Android SDK. The development of an application is free to try, but when publishing an application for Google Play, it will cost US$25 as a registration fee.

A big problem with Android is that with the open platform, a company can design a brand new phone but use cheap hardware. This will limit the number of supported phones by an application. Especially a modern 3D application must define which devices that are not supported by the application.

Table 2.2: Common screen resolutions for different devices running on Android

Name	pixel resolution
QVGA	240 x 320
HVGA	320 x 480
WVGA	480 x 800
qHD	540 x 960
HD	1280 x 720
FHD	1920 x 1080
WXGA	1280 x 800
WQXGA	2560 x 1600

An application running on Android can use either physical and/or virtual navigation buttons on the phone. Like for iOS, an Android application should contain a virtual back button, in Android called an up button. If there also is a physical back button on the device, this will result in possible navigation through both the up button in the application, and the device's physical back button. The up button should be visible if there exists a screen which is positioned one step above the current view in the navigational hierarchy. This will result in the starting page in the application not having any up button. The device's physical back button should be used to navigate in reverse chronological order. The up button will ensures that the user stays in the application, the physical back button however can be used to either close the application or to switch to another application. As for the navigation bar in iOS, Android's equivalence is called the action bar. The action bar should contain the icon of the application, together with the up button, in the left part of the bar. Next to the icon there should be a view control that allows the user to switch between views. If the application does not support different views, this space could be changed into a non-interactive component such as a label of application title. The action buttons are located next to the view control and enable users to perform different action on the application such as editing settings. If there is a lot of action buttons, an action overflow button will appear, containing the buttons which did not fit in the action bar [4].

Figure 2.2: Android Action Bar

2.2 Game Engines

A game engine is a system or framework used by a developer to create games or applications. Such a framework provides functionality for some of the most crucial and

common components in a typical game. Some of the most common functionality that a game engine provides are physics handling, user input, rendering, animation and scripting. The idea and purpose of a game engine is to prevent the developer from recreating standard components and functionality that have already been created in a previous, similar game. Instead, the developer is provided with a framework that allows the developer to put more focus on the creation of the game content. The game content, or the game assets, refers to the collection of scripts, textures, animations, sounds, models, etc. that will be used within the game engine to create the graphical and functional content of the game [5].

There is a wide variety of game engines available with different pros and cons, but due to the scope of this thesis, only game engines that support 3D, iOS and Android will be of interest. These requirements leave out some other popular game engines such as CryENGINE 3 and Torque 3D. The most common and popular game engines that fit into the description are Unreal Development Kit (UDK)/ Unreal Engine 3, ShiVa 3D and Unity. A very crucial and deal breaking factor to look at when deciding which game engine to use is the licencing. Since each game engine company uses different methods of licencing, this is not an easy process.

2.2.1 Licencing

UDK offers a completely free licence for educational and non-commercial use. Here, the term non-commercial refers to whether a released game is making revenue or not. This means that a game that is completely free and without any commercial aspects such as advertising or in-game content, is allowed to be released with the free UDK licence. The platforms that this free licence covers are limited to Apple iOS and Windows. If the UDK will be used to develop and release games or applications for commercial purposes, a commercial licence is available for US$99 per year. This commercial licence is not paid per-seat, instead, the licence could be used either by a single developer or a team of developers from the same company. There are no restrictions regarding the number of different games that are developed and released under the licence. However, the company has to pay the creators of UDK, Epic Games, 25% royalty on the revenue of the games or applications developed with UDK. This royalty only apply to revenues that exceed US$50000. If, for example, a company earns US$60000 from a game developed with UDK, that company must pay US$2500 to Epic Games because of the 25% royalty applied on the US$10000. For larger companies with more experienced and professional developers, there is an option to licence Unreal Engine 3. The differences between the UDK and the Unreal Engine 3 is that, though the UDK uses the same features as the Unreal Engine 3, an Unreal Engine 3 licence includes the underlying source code to the engine. Additionally, a full Unreal Engine 3 licence can release games for Adobe Flash, Google Android, Microsoft Xbox 360, Mac OS, Sony PlayStation 3, Sony PlayStation Vita and Nintendo Wii U. The price for this licence is not officially known due to the owner of the licence being under a NDA (Non-Disclosure Agreement) with Epic Games [6] [7] [8] [9].

Unity has a different licence system to that of UDK. As for UDK, a free licence exists. This free licence does not include all the features of Unity, but games devel-

oped by the free licence can be sold without any per-title fee or royalties. However, companies that exceed an annual revenue of US$100000 will not be licenced with a free licence and need to purchase the full version of Unity. For developers that wish to take advantage of the full feature set of Unity, a licence called Unity Pro could be bought for US$1500. This licence applies only to one specific individual and is not allowed to be shared with other developers, regardless of them being in the same team or company. However, one licence is valid for two different computers, used by the individual. The free licence of Unity gives the developer the possibility to develop games and applications for PC, Mac, Linux, Adobe Flash, Microsoft Xbox 360, PlayStation 3, Nintendo Wii U, Google Native Client, Web Player, Apple iOS and Google Android. It should be noted that development for Wii U, Xbox 360 and PlayStation 3 requires approval from the specific platform holder, this is also the case for development with Unreal Engine 3. Two additional licences exists for Unity, iOS Pro and Android Pro. Each licence brings additional features to the development for the respective platform which neither the Unity free nor Unity Pro includes. Both licences are priced at US$1500 each and requires the Unity Pro licence. With that said, a developer who wishes to take full advantage of the features of development for Google Android, needs to purchase the Unity Pro and Android Pro licences for a total of US$3000. Additionally, Unity offers a so called Team Licence, priced at US$500. The Team Licence does not require Unity Pro. The licence gives the user access to the built in Unity Asset Cache Server and Unity Asset Server, giving the developer features of reducing asset import time and using team collaboration [10] [11] [12] [13] [14] [15].

The ShiVa3D Engine offers a free web edition of their ShiVa Editor for download. The web edition has very few restrictions when it comes to features of the game engine itself, however, it only allows for publishing the material to the web browser. For the other supported platforms Nintendo Wii, PC, iOS, Android, Palm WebOS and Blackberry, publishing is restricted to testing purposes only. If a developer wishes to publish for those platforms, a Basic licence is required. The Basic licence has a price of US$400 and is valid for one machine only. ShiVa3D also offers an advanced licence, which includes more features than the free and basic versions. These additional features include SVN (subversion) support, LOD (level of detail) and plug-in export. The Advanced licence is priced at US$2000 and is, like the Basic licence, only valid for one machine. Neither the Basic nor the Advanced licences have any publishing fees, royalties or revenue restrictions [16].

When looking at the different licences for Unity, UDK and ShiVa3D, a conclusion can be made about the minimum price one has to pay in order to be able to release commercial games for both the iOS and Android platforms. For Unity, if the assumption is made that the game will not revenue more than US$100000, the need is met with the free licence since it allows publishing for both the iOS and Android platforms. Additionally, Unity Technologies will not demand any royalties or other fees from the profit of possible sales. If we instead look at the UDK, it is a bit more complicated. A big problem here is that neither the free nor commercial licence allows games being published to Android. The solution would be to apply for a full Unreal Engine 3 licence but since that licence is meant for larger companies, it is out of the scope of this thesis. Finally, the ShiVa3D Engine require a Basic or

Advanced licence in order to allow publishing for the iOS and Android platforms. The cheapest solution here would be to purchase one Basic licence and, since the licence is bound to one machine only, share that machine amongst the development team. This solution would cost US$400.

The decision of which game engine to use in this thesis went to Unity based on three important factors. First and foremost, the US$400 price difference between Unity and ShiVa3D is a deal breaker due to the almost nonexistent budget of the project. Secondly, Unity has a larger community than ShiVa3D. Finally, Unity is a natural choice since the writers of this thesis does not have hands-on experience with UDK or ShiVa3D.

2.2.2 Unity

Unity, developed by Unity Technologies, is the most used mobile game engine by developers according to a survey published in an issue of the Game Developer Magazine. In the survey, 53.1% of the developers used Unity when releasing applications for iOS and Android [17]. A selection of games developed in Unity and released on both iOS and Android are: Temple Run 2 and Bad Piggies [18].

The most vital part of the Unity engine is the integrated editor (Figure 2.3). Here, the Game View functions as a WYSIWYG (What You See Is What You Get) preview of how the game will look and function. To manipulate objects in the scene, the Scene View window lets the user move, rotate and scale objects that are already in the scene. To insert objects into the scene, the user could drag and drop objects into the scene either from the Project View or the Hierarchy View. The Hierarchy View is a list of all objects and assets that are currently in the chosen scene. In order to modify or place objects into the scene, the assets must be accessible and placed inside the project folder, showed in the Project View. The last of the standard Views in Unity is the Inspector View. This view lists all the components that are currently attached to a selected asset, i.e. transform, animation and script components.

The Unity engine supports the use of three different programming languages namely C#, Boo and JavaScript. All languages are running on the Open Source .NET platform Mono, and can be written and debugged in Mono Develop, the IDE (Integrated Development Environment) that comes with Unity [19].

Mecanim

As of Unity version 4, which was released in November 2012, a powerful animation technology called Mecanim is included [20]. Mecanim gives the user functionality to setup and control animations for a character inside the Unity editor. There are two types of character types supported by Mecanim, namely humanoid and generic characters. A humanoid is a character that resembles the appearance of a human being whereas a generic character has no predefined bone structure, centre of mass or orientation. If a generic character is to be used with Mecanim, some of the elementary features provided by Mecanim will not be available. A simple example of a generic character is a dog. When a humanoid character is to be imported to Unity and used with Mecanim, Mecanim creates an Avatar. An Avatar is an interface

Figure 2.3: The Unity editor showing the Game, Scene, Hierarchy, Project and Inspector Views

which translates the characters bone structure to the bone structure understood by Mecanim. Figure 2.4 shows a successful translation mapping between the Mecanim bone structure and the actual bones in the character.

When a character has been successfully created with an Avatar, Mecanim allows the user to tell the game engine how and when different character animation clips should be played. For more information on animation clips, see Section 2.3.3. A powerful tool within Mecanim is the Animator component. In the Animator component, different states and layers are defined and controlled. A state in the Animator component corresponds to the specific set of actions and the environment of the character in the game. For example, a state defined as 'Idle' could correspond to a scenario in the game where the character is not moving at all. In this particular state, a looped animation clip of an idle motion could be applied to the state, forcing the character to play that animation while in the 'Idle' state. In order for the character to perform another action of movement, another state must be defined. For example, if the character is supposed to run, a state called 'Running' could be created, holding a clip of a looped running motion. Also, transitions between the two states must be defined, allowing the character to switch between the idle motion and a running motion. A transition is controlled by one or more conditions that have to be met in order to switch between the states. A condition is controlled by a user defined variable which in turn is controlled by a script. As for the example with the two states, the transition from the 'Idle' to 'Running' state could be triggered if the condition of the user defined variable 'Speed' is greater than a specific value. Figure 2.5 demonstrates an example of how a layer with different states and transitions will look like inside the Animator component.

Another powerful feature of the Animator component is the Blend Trees. A Blend Tree could be created inside a state to replace the current animation clip

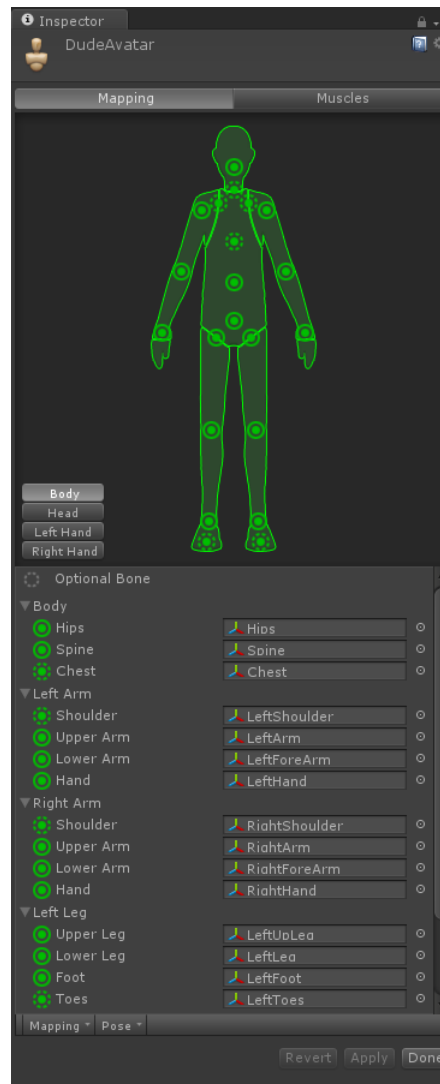

Figure 2.4: The Mecanim's Avatar Configuration, mapping the character's bone structure to Mecanim's bone structure.

in that state. For example, the 'Running' state's running motion clip could be replaced by a Blend Tree, see figure 2.6. The newly created Blend Tree could then include a multiple number of motion clips i.e., 'RunForward', 'RunLeft' and 'RunRight'. When the 'Running' state i active, the Blend Tree plays one of its motion clips depending on a user defined variable. In this example such a variable could represent the direction of the run in order to let the character run left, right or forward accordingly. If the value of the direction variable is in the middle between the threshold values of two motions, say 'RunLeft' and 'RunForward', the animations will blend into a dynamic animation which interpolates between the two motions. This feature results in smooth and dynamic transition motions between different animation clips [21] [22] [23] [24] [25] [26] [27] [28].

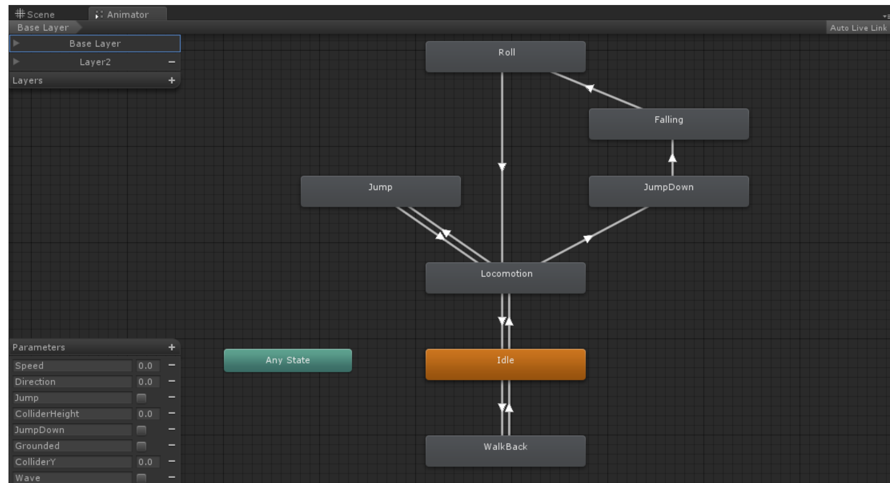

Figure 2.5: The Mecanim's Animator component, showing an example of different animation states and the transitions between them.

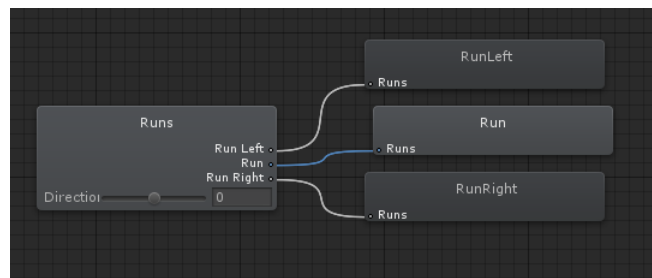

Figure 2.6: The Mecanim's Animator component, showing an example of a Blend Tree including three different animation clips.

Graphical User Interface

Graphical User Interface (GUI) is an interface that lets the user interact with the application through graphical objects like buttons and images. Unity offers implementation of such an interface through its own UnityGUI. With UnityGUI, the user is able to access the GUI class in order to create a Graphical User Interface. Examples of controls that are supported with the UnityGUI are labels, textures, buttons, textfields and pop-up windows. Though UnityGUI offers great features for developers, the UnityGUI is officially not recommended for use in mobile applications. The reason for this is because of how the UnityGUI functions. Whenever a GUI object is to be used or displayed on the screen, a script must call the OnGUI() function. This call will be made several times per frame hence requiring additional processor time and eventually decreasing the performance of the application [29] [30] [31] [32] [33]. With the purpose of solving this issue for mobile platforms, several third-party GUI frameworks exist. The three most popular ones are EZ GUI from Above and Beyond Software, Next-Gen UI (NGUI) developed by Tasharen Entertainment and prime[31]'s UIToolkit. With EZ GUI and NGUI being priced at US$199 and US$95

respectively, this thesis only focuses on the free UIToolkit framework [34] [35] [36].

The UIToolkit lets the user create interactive 2D-objects with one single draw call, instead of several per frame. In order to get textures working with UIToolkit and Unity, the free version of the software TexturePacker could be used. TexturePacker is a software which takes multiple textures and outputs a single, compressed, texture called a sprite sheet. Along with the sprite sheet, a data file called a sprite atlas is generated. The sprite atlas file contains information and coordinates of every texture that is currently present in the sprite sheet. There is a few performance related reasons behind packaging multiple textures into one single sprite sheet instead of importing each and every texture by itself into the game engine [37] [38].

2.3 Animation

A simple explanation of what an animation is could be given as "Animation is a sequence of images, with slight differences from one image to the next, that gives the impression of movement."

3D animation originates from 2D animation, but includes one more dimension. When animating in 3D, the developer will have to consider all of the different angles in order to have a good understanding of the animation, which makes the view-port in the 3D animation program very important. Time is the most important factor of animation. In animation, frames are used to measure time. The measurement of animation time is defined by the number of frames existing within a second of an animation. The standard number of frames in a second differs between geographical locations and the purpose of the animation. 24 frames per second (fps) is used in films, 30 frames per second is the standard in North American (NTSC) and 25 frames per second is the European standard (PAL). When creating 3D animations in a computer, a common term that is used is key frame, the name originates from key in 2D animations, where the key position was the picture which the lead animators drew. The assistant animators then drew the frames in between two key frames. In a modern animation software, the key frames are frames where the key positions of different objects are set. Frames between two key frames will be automatically predicted and generated [39].

2.3.1 Motion Capture

Motion capture, often referred to as MoCap, is the process of recording an entity's movement. MoCap has a variety of usage and system types but will, in this thesis, only be discussed in relation to computer animation and optical systems. When MoCap is used in the fields of film making or software development, a common process is to record the movement of a human actor, then using the recorded data in order to create animations to digital character models. Optical motion capture systems can be further divided into marker based and marker less systems.

In a marker-based optical motion capture system, markers are placed on the object which motions will be recorded. The markers are either reflective (passive)

or light emitting (active). Passive markers are made of reflective material and shaped as spheres. The size of the markers depends on the resolution of the cameras and what is to be recorded. Smaller markers are used for facial and hand captures and larger markers are used for captures of the rest of the body. The markers are placed directly on the subject's skin or clothes. Alternatively, the markers can be attached to a MoCap suit, making the marker setup phase faster and easier. Cameras in a passive system both have an emitter and a receiver in order to track the positions of the markers. In an active system, the markers will illuminate, either one at a time or all at the same time using different amplitudes and frequency modulations for each marker. Both marker systems are sensitive to different lighting conditions, especially a passive system using natural light. Marker-based systems require an environment of at least two cameras to detect the markers 3D-positions, however three or more cameras are preferred to achieve better accuracy. If a marker is covered by a body part, making the marker hidden from all cameras, there will be a loss of data from the recording. There are different editing techniques to fix this, but when too many markers are covered, or if the time which the marker has been hidden is too long, it will not be easy to fix the problems related to the loss of data [40].

Marker-less Motion Capture Systems use advanced computer vision to identify and track subjects without the need of special suits or markers. Instead, computers use advanced algorithms to track motion in real time. Marker less systems have a few advantages over marked-based systems, for example the environment setup is faster. It is also easier for children to use a marker-less system since no special suits or markers are needed. The difficulties, however, is in implementing accurate tracking algorithms that perform good enough results in real time usage [41].

The data recorded in a motion capture session is then used for the characters in a game. The recorded data will be applied to a rigged skeleton, which is the framework that moves and makes the character come alive. When applying the motion capture data to the skeleton, the easiest pose to work with is the T-pose. In the T-pose, the subject to be recorded will face forward to the camera with the arms straight out and with the palms down. The feet should be about shoulders' width apart. The body pose will now resemble a T. In a motion tracking application, the recorded data will be mapped to a skeleton. This process will be initialised on a frame where the subject is in a T-pose for easier mapping between the data and the skeleton. [40].

Kinect

The Microsoft Xbox 360 Kinect is a device that lets the users use their body as a controller. By identifying the positions of the user's body parts, the user's movements can be used to control a character in a game or to record animations.

The Kinect has an infra-red light projector and a receiver to measure the time for the light to reflect on an object. This will result in a depth map with the size of 320x240 pixels at 30 frames per second. The Kinect also has a regular RGB colour camera with the resolution of 640x480 pixels. The Kinect is an example of a marker less motion capture system [42]. The cost of a Kinect is US$99.

An external program is needed in order to record animation data from the Kinect.

Since this thesis investigates recordings made from both one and two Kinects sensors, only software supporting animation recording from two or more sensors are of interest. Two software that match such a requirement are nuiCapture Animate, developed by Cadavid Concepts, Inc and iPi Recorder, developed by iPiSoft LLC. nuiCapture Animate is priced at US$399 and iPi Recorder Basic Edition is priced at US$595. Both companies offers a trial version of their respective software with nuiCapture Animate trial version including a capture session limitation of 30 seconds and iPi Recorder being available for free (the program iPi Mocap Studio is needed to analyze the recorded data and comes with a 30 day trial including all features of the Basic Edition.) Neither of the trial licences are allowed for commercial purposes [43] [44][45] [46].

Qualisys

The Qualisys Motion Capture System system is a marker-based optical motion capture system. In this thesis, the Qualisys system at Lund University Humanities Lab, which consisted of eight cameras at 240Hz for tracking, used. The complete system was priced at approximately SEK 1 000 000. Things needed to make a recording in the lab are:

- A recording computer (always prefer the faster stationary Qualisys computer), with Qualisys Track Manager (QTM)
- Motion tracking cameras, including power cables and ethernet cables.
- Camera mounting equipment, tripods and/or poles.
- A data cable (Ethernet-to-parallel if stationary computer, otherwise the Ethernet-to-PCMCIA-card if laptop).
- Markers
- Calibration kit (calibration frame and a wand, sized depending on size of measurement volume).
- Double-sided tape for placing markers. A blanket to reduce glare from table or other equipment in the room.

The cameras should be placed in a way so that at least three of the cameras can see the intended markers all times.

A software called Qualisys Track Manager is used to record and export the data as a motion file to Autodesk MotionBuilder [47]. Before recording, a calibration of the system is needed in Qualisys Track Manager. This is done by using a calibration wand to make sure all the cameras are seeing the markers used in the recording session.

Figure 2.7: The Qualisys calibration wand

2.4 Character Design

When creating a character for a game there is often a predefined workflow or pipeline which includes a lot of steps. The pipelines can be very different, however there are usually similarities for the different steps. Some of the most common steps include concept design, base modelling, high poly modelling (sculpting), low poly modelling (retopology), unwrap, texturing and rigging/skinning. The company Zero Point Software A/S has created a game called Interstellar Marines [48]. On their website, they present the pipeline that is being used when creating characters for their games. This pipeline will be the inspiration of the workflow used in this thesis regarding the creation of a character. The first step of creating a character is to come up with a concept. A concept can be a painting or sketch, illustrating the desired appearance and theme of the finished character.

Figure 2.8: Conceptual art of a character used in the game Interstellar Marines

The next step is base modelling. In this step, the base model should consist of a rough model, with the proportions and shapes fairly correct while still keeping the geometry simple. This step is usually done in a 3D computer graphics software such as Blender and Autodesk 3ds Max. Blender is a free and open source software used for modelling and creating animated films, visual effects, interactive 3D applications or video games under the GNU General Public License [49]. 3ds Max is similar to

Blender, but has more advanced features and is not free to use. 3ds Max is priced at €4485 [50] [51].

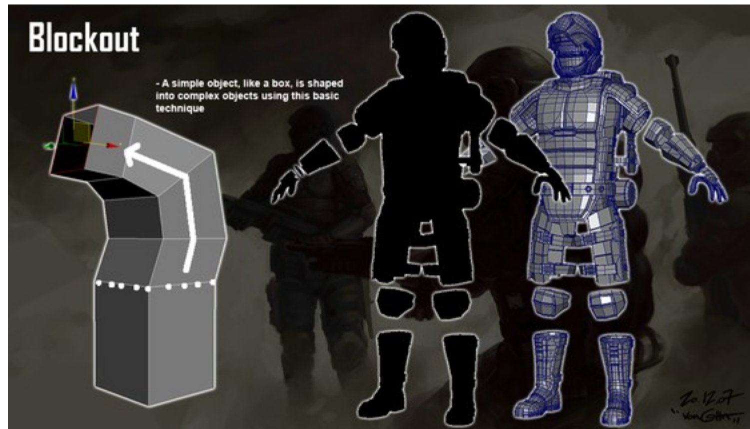

Figure 2.9: The base mesh used for the character in the Interstellar Marines game

After the base model has been created, a high poly model is to be made. This can be done by sculpting the base model to add more detail to the character, for instance wrinkles. Due to the introduction of high level details, this step can be very time-consuming. The sculpting process can be described as shaping a low polygon virtual clay mesh into a high polygon detailed clay mesh. This process is often done in Zbrush, Sculptris, Mudbox or Blender. Both Zbrush and Sculptris are made by Pixologic. Sculptris is completely free, but Zbrush is priced at US$699 [52] [53] [54]. Mudbox is developed by Autodesk and has a price of €795 [55][56].

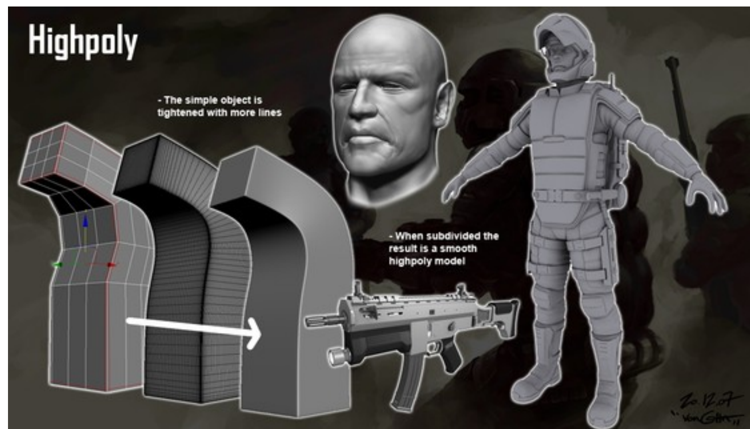

Figure 2.10: A high polygon model in the Interstellar Marines game.

The next step is to create a model which similar to the base mesh uses fewer polygons than the high polygon model. This is done by creating a type of low polygon shell on top of the high polygon model. The reason behind this is that the high polygon model will have too many vertices, resulting in performance issues when the application is running on a mobile platform. By reducing the level of

detail and number of polygons in the modelling program, the details applied in the previous step will be lost, hence reducing the quality of the character. However, a solution reducing the loss of quality will be presented in the steps to follow. In order to create the low polygon shell on top of the high polygon mesh, a new mesh is created. This step is often called retopology and can be executed in several different ways. There is a program called 3D-Coat that can do this process automatically, but the software comes with the price of US$349. Retopology can also be done in Blender, ZBrush, 3ds Max or Topogun, priced at US$100. If it is done manually, the new mesh's vertices are attached to the high polygon mesh. Then, the new mesh creates big polygon squares on top of the high polygon mesh's surface. The larger the created squares, the less detailed surface is made. On areas that need lower quality, such as the torso and back, bigger squares are created compared to squares that surface more detailed areas such as eyes and nose. When working with mobile platforms, this is one of the most crucial steps. Lower polygon count will result in a model more suitable for mobile devices, but with lower quality. The number of polygons and the quality will have to be adapted correctly for each project [57].

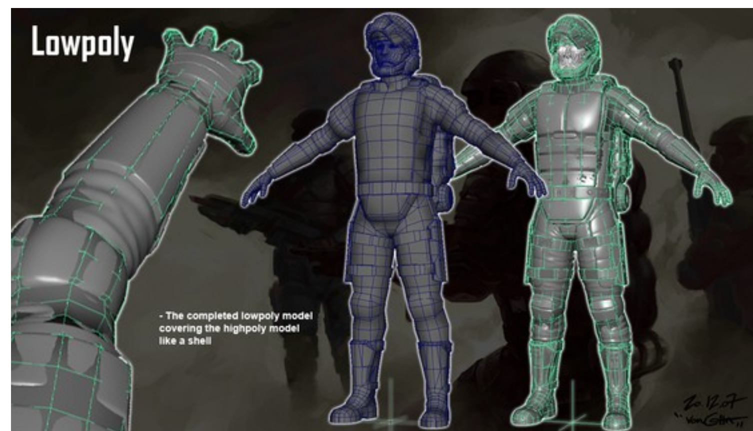

Figure 2.11: A retopologized mesh of the character used in the Interstellar Marines game

When the retopology process is done, an unwrap needs to be applied to the mesh. An UV map resembles the complete surface of the low polygon model and consists of a flat 2D image. The surface of the low polygon model can be split into several different parts which are all then assembled into one UV map. This map will be used as the base for the next steps when baking the colour map and the normal map.

Texturing is the next step. Here, both the high polygon and the low polygon models are used to create a normal map. A normal map is a method of faking bumps and dents in order to create more details without increasing the number of polygons.The normal map is created by baking the high polygon model's surface onto the low polygon model using the coordinates from the UV map. When importing and applying the normal map onto the low polygon character in the game engine, the low polygon character will almost contain the same amount of details as the high poly character, but with a vastly reduced number of polygons. Additionally, a colour map will be generated. A colour map contains the texture that has been

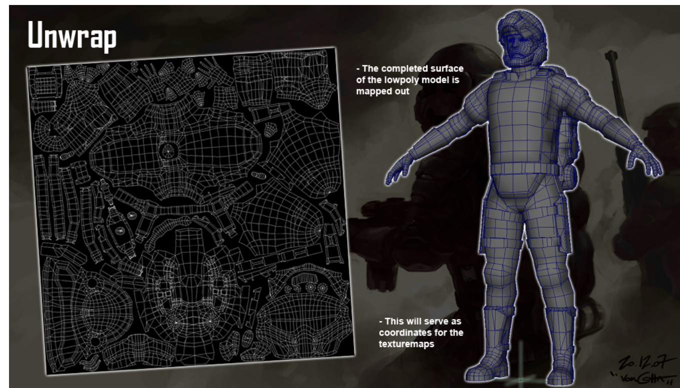

Figure 2.12: An UV map of the surface on the character used in the Interstellar Marines game.

painted on the character [58].

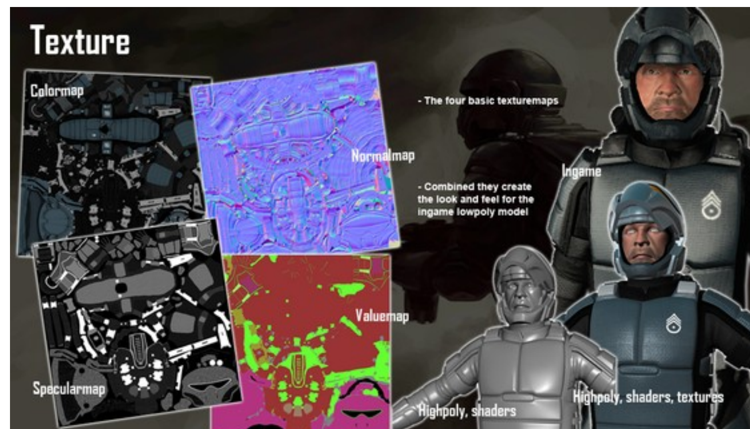

Figure 2.13: The texturing made in Interstellar Marines game, showing both the normalmap and the colourmap.

When the design and modelling of the character is complete, the character needs to be rigged and skinned in order to be able to use with animations inside a game engine. Character animation is the most advanced type of animation. Unlike a tree, a character not only needs to be able to move, it also must be able to express itself and show emotions. To rig the character, a skeleton built together by different bones needs to be added to the character. The bones need to be attached to the character using a skin modifier. This process is to ensure that the geometry surrounding the bones deforms properly when the skeleton moves. In 3ds Max, a skeleton could either be created by adding and connecting different bones to each other or by creating a Biped component, which is a predefined two legged skeleton. Similar approaches can be made in Blender with either a separate bone connection approach or with Blender's equivalence of Biped, the Rigify system. When a skeleton has been created and placed in order to fit with the proportions and anatomy of the character, the bones must be skinned to the character. In 3ds Max, a Skin modifier is used. With the modifier, each bone will have to be adjusted with envelopes and weights to connect the correct vertices to the bone for that specific body part [39].

Instead of making the character design process manually, there are tools to generate a fully rigged and skinned low polygon character with corresponding maps. Such a program is MakeHuman, which is an open source tool for making 3D characters. All of the content created with MakeHuman is licensed under the CC0 license. This will give artists a really high freedom of using the creations from MakeHuman in any way imagined. Another tool available is called Project Pinocchio and is created by Autodesk. The Project Pinocchio is similar to MakeHuman but does not have the same possibilities of making the character unique. With Project Pinocchio the appearance of a character is defined by editing parameters corresponding to the differences between two already defined characters. There are also other programs, but with limited licenses and heavy price tags. Some of those programs are DAZ Studio, made by DAZ Productions and Poser, made by Smith Micro Software [59] [60] [61] [62].

Chapter 3
Design and Implementation of an Example Collection

In order to illustrate the workflow, a collection of game examples were developed

3.1 Game functionality

The basic idea of the game example collection is to challenge other users in a tic-tac-toe based adventure game. When facing another user in a game, a tic-tac-toe board is created consisting of nine empty squares. Behind each square hides an unknown mini game. When the user who is drawn to make the first play on the board, decides to play one of the nine squares, the hidden mini game presents itself for the user in a new scene. The different mini games are explained in greater detail below. In a mini game, the user is first given instructions on how to play the game along with information regarding eventual high scores for that level. When the user is ready to play the mini game, a timer starts to count down. Before the time runs out, the user should collect as many points as possible in that mini game. Some of the mini games can also end prematurely if the user fails with a specific action. When a mini game has been completed, the square corresponding to the finished mini game will update with an icon showing the owner of the square along with the score that needs to be beat in order to overtake that square. The name of the mini game behind a square is only visible to a user which has already played that particular square on the corresponding board. A square could be overtaken by a user if the highest score for that square is beaten. The highest score for a square could also be updated if a user is already the owner of a square but wishes to improve the score that the opponent needs to beat. As with standard tic-tac-toe rules, a game is won by a user if he or she owns three squares in a row. An example of how the game board could look is presented in Figure 3.1. The initial placement along with the spawning of

objects in each mini game has a factor of randomness resulting in a uniquely looking mini game each time a mini game is played.

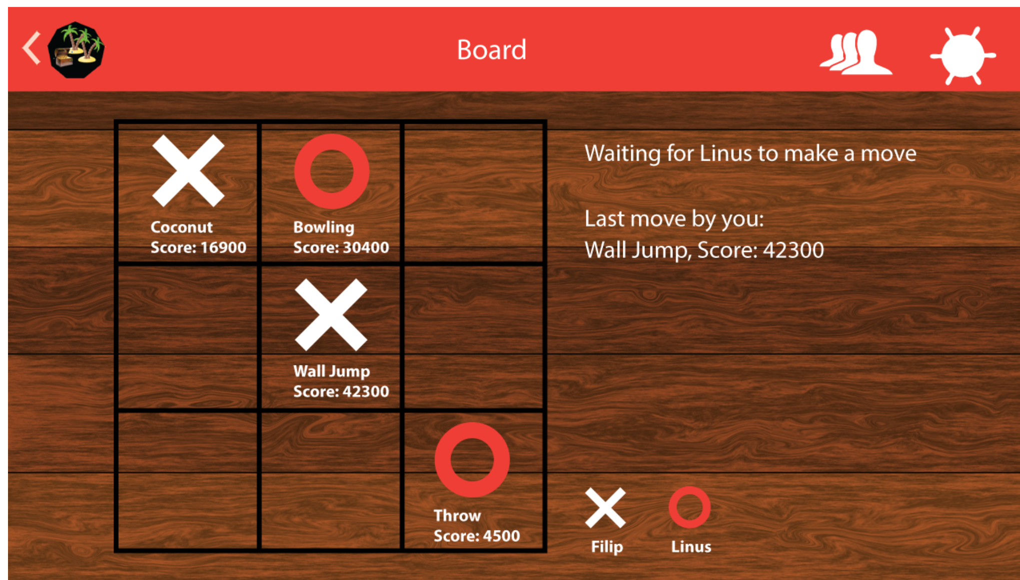

Figure 3.1: The tic-tac-toe board shown on an Android device.

3.1.1 Mini games

Though the tic-tac-toe game consisted of nine different squares resulting in nine different mini games, hi-fi prototypes were made for 13 different mini games. The mini games were then evaluated by a group of 12 participants. Each participant had to remove four of the mini games, in no particular order, which he or she did not wish to be included in the game. The result of the evaluation is shown in Table 3.1. Based on the evaluation, four of the mini games were removed. The mini games not developed further and instead removed from the game, were internally called Brick Breaker, Bomb Game, Boat Race and Obstacle Run.

Below are the presentation of the mini games that were selected after the evaluation. Each mini game is described in greater detail regarding the goals, controls and scoring system. The mini games Wall Jump, Throw Game and Rolling Stones are the most developed mini games in terms of features, assets and design. The Tightrope and Bowling games are also very close to completion. However the Coconut, Car, Raft Jump and Whack a Mole games require some more development before they are to be fully integrated into the game.

Wall Jump

The object of this game is to get as many points as possible by getting as far up in the sky as possible. The character moves up in the sky by jumping on platforms and springs spawned throughout the sky. Bonus points could also be collected by acquiring bubbles which are present on some of the platforms. The platforms have different properties. A standard platform has a fixed position in the sky and if the

Table 3.1: User evaluation results.

participant	game
participant 1	Brick Breaker, Bomb Game, Boat Race, Obstacle Run
participant 2	Bomb Game, Car, Boat Race, Wall Jump
participant 3	Raft Jump, Bomb Game, Obstacle Run, Whack A Mole
participant 4	Brick Breaker, Bomb Game, Whack A Mole, Obstacle Run
participant 5	Bomb Game, Obstacle Run, Boat Race, Brick Breaker
participant 6	Raft Jump, Bowling, Tightrope, Obstacle Run
participant 7	Brick Breaker, Boat Race, Throw, Obstacle Run
participant 8	Boat Race, Bomb Game, Obstacle Run, Coconut
participant 9	Coconut, Obstacle Run, Boat Race, Brick Breaker
participant 10	Brick Breaker, Boat Race, Raft Jump, Obstacle Run
participant 11	Obstacle Run, Bomb Game, Bowling, Brick Breaker
participant 12	Rolling Stones, Car, Boat Race, Obstacle Run

character lands on the platform it will give the character a vertical boost. Depending on a random factor along with the height of the character, some platforms with a moving behaviour might be spawned. These platforms will move back and forth on the horizontal axis between two fixed positions. There is also a platform with a fixed position that will be destroyed when jumped upon by the character. Finally, a fourth platform exists, also with a fixed position. This platform contains a spring which, if if jumped by the character, will give the character a vertical turbo boost. The mini game will end if the time limit is exceeded or if the character falls downwards for a long period of time.

A virtual joystick on the left side of the screen is used to move around in the vertical-axis. If the character is grounded, the character will use a strafe motion when moves. If the character is in the air, the character will move in a flying behaviour.

Figure 3.2: A screenshot from the Wall Jump mini game.

Throw

In this mini game, the objective is to hit as many birds and dolphins as possible by throwing balls at the objects. Points are given when either of the objects are hit, however, a dolphin is harder to hit than a bird therefore gives three times more points if hit. This mini game also features a score combination system. When an object is hit by the ball, the score given for hitting that object is multiplied with how many objects the user has hit in a row. If the ball misses an object, the streak is set to zero.

To throw a ball in the mini game, a screen swipe gesture is needed. The swipe gesture is made by putting down the finger in the bottom of the screen and then dragging it upwards in the desired angle. The velocity of the ball is calculated as the time taken between the beginning of the swipe gesture and when the the finger is released from the screen. The mini game will only finish when the time limit is exceeded.

Figure 3.3: A screenshot from the Throw mini game

Rolling Stones

The objective of this mini game is to catch coins which are bouncing down towards the character from a mountain. To increase the difficulty, large burning rocks are also bouncing down the mountain. The player needs to collect coins in order to acquire score but the user must also avoid getting hit by the rocks. The score in this mini game are given when a coin is collected. If the coins are collected in a streak, a multiplier will be added to the score of each coin collected with the same principles as for the Throw mini game. The mini game will finish if the time limit is exceeded or if the character is hit by a bouncing burning rock.

To move the character, a virtual joystick is used. Movement can only be done with a strafing motion along the horizontal axis.

Figure 3.4: A screenshot from the Rolling Stones mini game

Tightrope

The objective in this mini game is to collect as many coins as possible. The coins are placed on different platforms. By jumping between the platforms, coins are collected. The user must also avoid to lose the balance and fall into the water. The platforms will spawn with smaller and smaller sizes as the user progresses in the mini game. In this mini game, the user controls the movement of a ball that rolls forward with a constant speed. As for the combination systems of the Rolling Stones and Throw mini games, if coins are collected in a streak, a multiplier will be added. However if the user misses to collect a coin, the streak is reset. The mini game will finish if the time limit is exceeded or if the ball falls into the water.

In order for the ball to move sideways, the accelerometer on the device is used. The steering is done in the horizontal axis by tilting the device to the left or right. To jump from one platform to another, a tap on the screen will make the ball jump.

Figure 3.5: A screenshot from the Tightrope mini game

Bowling

This game's objective is to get as many points as possible by hitting pins with a bowling ball. There are three different objects that can spawn on the track. First there is set of pins. If the ball hits a pin it might fall depending on the collision speed and angle. Also, pins might fall by colliding with each other. The points are given when hitting a set if pins as in regular bowling with a strike working as a score multiplier. There is also holes in the ground being spawned on the track, if the ball falls down such a hole, the mini game will finish. Finally, there are speed boosts objects which will, on collision with the ball, boost the speed of the ball over a period of time. The mini game will finish if the time limit is exceeded or if the ball falls into a hole.

The controls of this mini game is the same as for the Tightrope mini game except the ability to jump is not available in this game

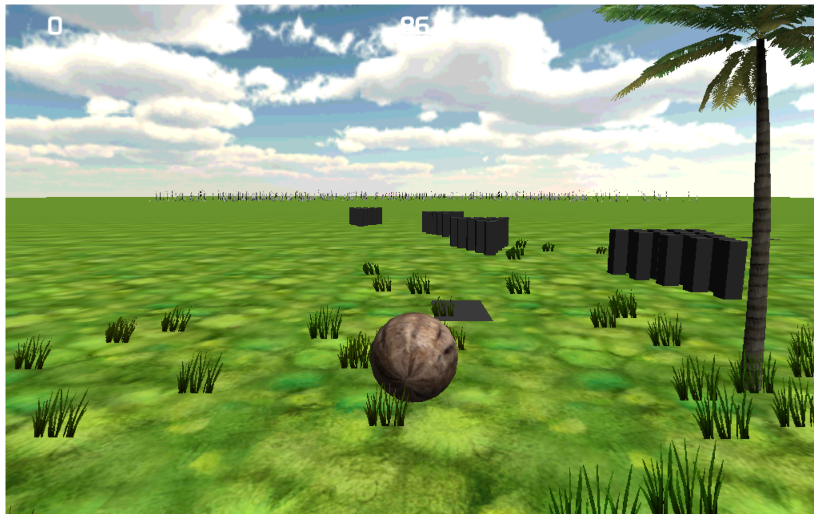

Figure 3.6: A screenshot from the Bowling mini game

Coconut

This game is about catching as many coconuts as possible. A coconut is caught if it falls into the basket carried by the character. When a coconut falls from the sky, it will bounce one time on the ground for the player to have an extra chance of catching it. Points are given when catching the coconut in the basket. If the coconut is caught before bouncing on the ground, the points will be higher than if there was a bounce before it was caught.

To move the character, a virtual joystick is used. Movement can only be done with a strafing motion along the horizontal axis.

Car

In this game the user should drive a car around a randomly created track. The track consists either of straight stretches, left turns and right turns.

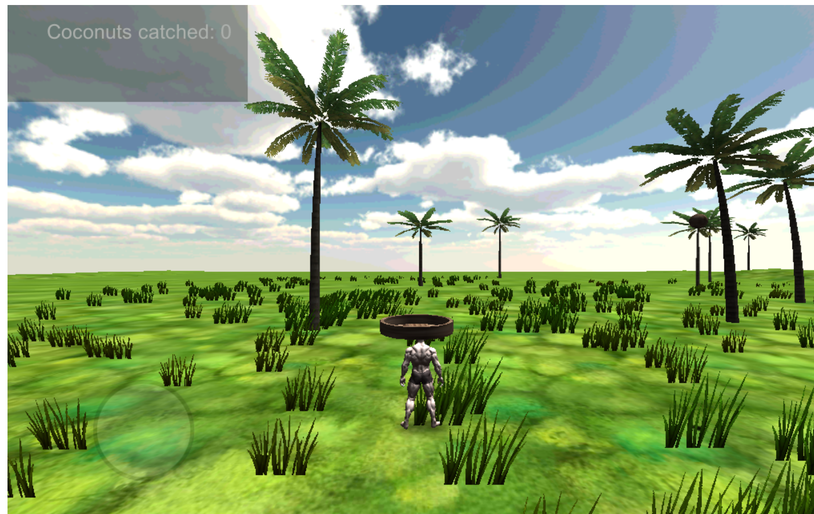

Figure 3.7: A screenshot from the Coconut mini game

To steer the car, the virtual joystick is used in order to turn left or right. There is also a throttle button and a break button which will power and break the car.

Figure 3.8: A screenshot from the Car mini game

Raft Jump

The objective of this game is to collect as many coins as possible. The coins are located on floating rafts. In order to collect the coins, the character must jump upon the rafts and catch the coins. The rafts float with different velocities and have different sizes. The number of coins on each platform is also based on a random factor. If a platform is hit by another platform, the platform with the lowest velocity will sink into the water along with eventual coins on it. The mini game will finish if the time limit is exceeded or if the character falls into the water.

A virtual joystick is used in order to move the character in the x and z-axis. Also, a jump button exists to make the player jump between platforms.

Figure 3.9: A screenshot from the Raft Jump mini game

Whack a Mole

The objective of this game is to whack as many moles as possible by hitting them with a bat. The moles raises from their respective holes in random intervals. Some moles stay up for a longer period of time than others, before going back down in the ground again.

A virtual joystick is used in order to move the character in the x and z-axis. Also, a whack button exists to make the character unleash a hit with the bat.

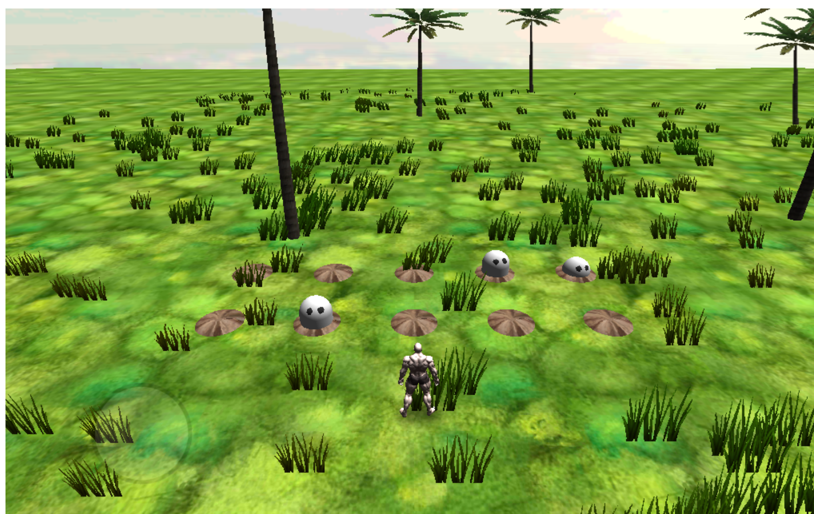

Figure 3.10: A screenshot from the Whack a Mole mini game

3.1.2 Matchmaking and User Community

When starting the game for the first time, the user is presented with a login screen which allows the user to enter his or her user name and password in order to login.

If the user wishes to register a new account, the user could do so by proceeding to the register forms. The login screen is showed in Figure 3.11.

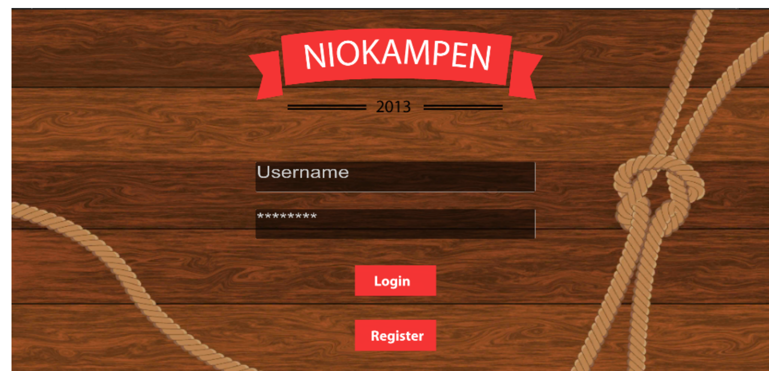

Figure 3.11: The login screen containing forms for user name and password along with buttons for login and registration of a new account. This screen is identical for both Android and iOS devices.

When the user is successfully logged in (whenever the application is started, the user will be automatically logged in if an account is already connected to the device and the remembered password is still valid) to the application, the main lobby shown in Figure 3.14 is presented. Here the user can choose whether to start a new game, access the board of a currently active or recently completed game or use the navigation bar. The games are listed in different frames depending on whether it is the user's turn, the opponent's turn or if the game is finished. The navigation bar is visible on most parts of the application and contains navigation for previous/back/logout and access to the friends screen and settings screen. If the user chooses to start a new game it will be presented with the screen shown in Figure 3.12. In this lobby the user is able to start a new game against a random opponent, accept eventual received challenges from friends or challenge a friend. The user is also encouraged with a button to add a new friend. The game's matchmaking system is heavily inspired by popular cross-platform multiplayer games such as Wordfeud [63]. If the user decides to play against a random opponent, the application searches for the user who has waited the longest period of time without being given an opponent and matches that user to the user most recently requesting a game. If there is no available opponent, the user's newly requested game will be put in a waiting list until the system finds a match. If the user instead decides to challenge a friend, the challenged friend receives a challenge request from the user. If the friend accepts the challenge request, the game will be accessible in the main game lobby for both users. The challenge requests will be presented to the user in the main lobby.

If the user navigates to the friends list, either by pressing the friends icon in the navigation bar or pressing the add new friend button in the start new game lobby, the user is presented with the screen shown in Figure 3.13. Here, eventual friend requests are shown along with a list of current friends. The user can either choose to accept eventual requests available in the request list or to challenge a friend as possible in the start new game lobby. Additionally, the user can add a friend by typing in the friend's user name and pressing the add friend button. If a user with

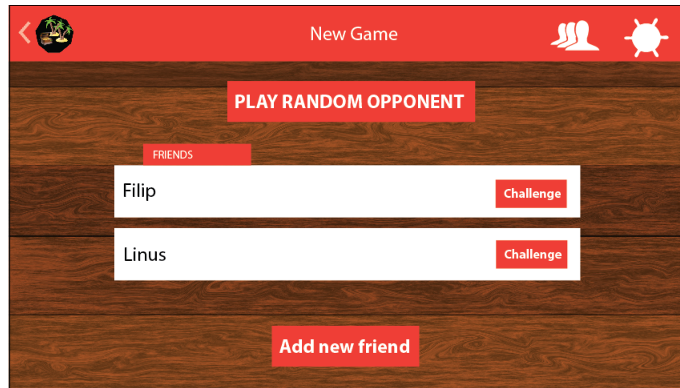

Figure 3.12: The start new game lobby shown on an Android device.

corresponding user name exists, a friend request will be sent to that user. If and when the request is accepted by the other user, the new friend will be present in the friends list.

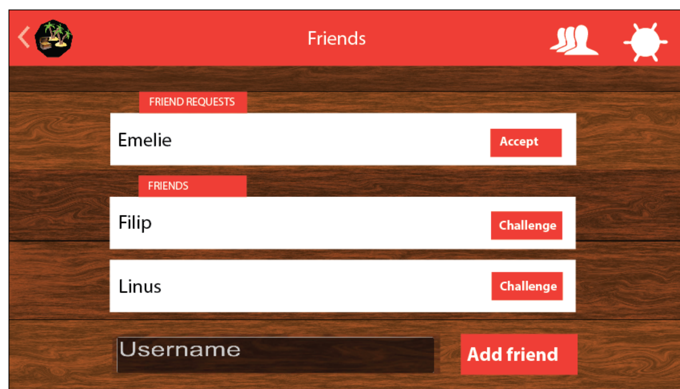

Figure 3.13: The start new game lobby.

Apart from participating in challenges with friends and random games against opponents, the game also includes a highscore system that lets the users compete with themselves and against all other users. Before playing a specific mini game the user is presented with instructions and highscores. Except for the score to beat in order to overtake the square belonging to the mini game, the user's personal best score for that level is presented along with the user name and score of the world's best score for that level.

3.2 Cross-platform development

3.2.1 User interface

Creating the user interface was the biggest cross-platform development related issue that was dealt with in this thesis. Normally when creating an Android application, the user interface is created in an IDE, i.e. Eclipse, and designed in a graphical editor

and/or with an (Extensible Markup Language) file. This is often the ideal solution because of the support of standard Android components, different types of layouts, multiple screen sizes and easy navigation. For iOS, the equivalent of designing user interfaces is the Interface Builder, integrated in the IDE XCode. Using these methods of designing separate user interfaces for iOS and Android would require some type of integration of the Unity application with the XCode and Eclipse IDE's. This would not comply well with the goals and purposes of the cross-platform development in this thesis because of the work needed to design two separate interfaces and applications. Instead, the user interfaces of the game was designed and developed inside Unity with the prime[31] UIToolkit framework. By using this method instead of creating separate interface solutions in the respective IDE for every platform, the amount of time and line of code needed to change something in the interface is reduced. With this solution, almost the same code is run, independent of the platform, in order to create the user interfaces. The guidelines and differences in the interfaces between the two platforms are explained in section Platforms. In order to comply with those guidelines, the application determines whether the device is running on the Android or iOS platform and adds specific user interface components to the generic interface based on the guidelines for that specific platform. An example of this could be seen in Figure 3.14 respectively Figure 3.15. Here Figure 3.14 shows how the main lobby in the game looks like when using an Android device. The same lobby is shown in Figure 3.15 but here on a device running iOS. In the latter figure, some graphics and navigation components differ from those shown on the Android devices in order to comply with the interface guidelines for iOS applications. Though some of the visual entities present are not exactly the same in both figures, most of the components shown in the example is identical and is built with the same code independently of the used device.

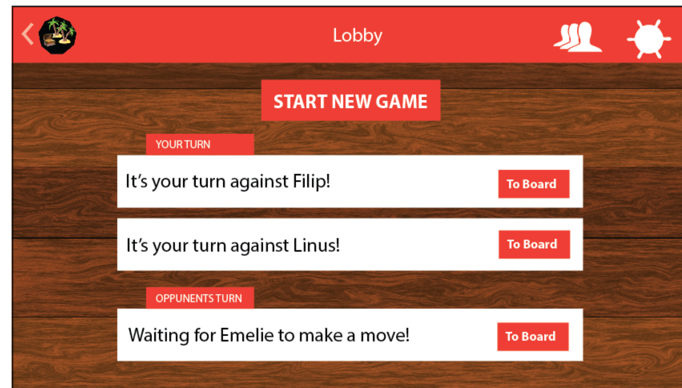

Figure 3.14: The main lobby of the game presented on a device running on Android.

3.2.2 Server architecture

Unity comes with support for networked multiplayer with methods like State Synchronisation and Remote Procedure Calls [64]. Except for Unity's network support, which is built upon the networking engine RakNet [65], there is a lot of networking

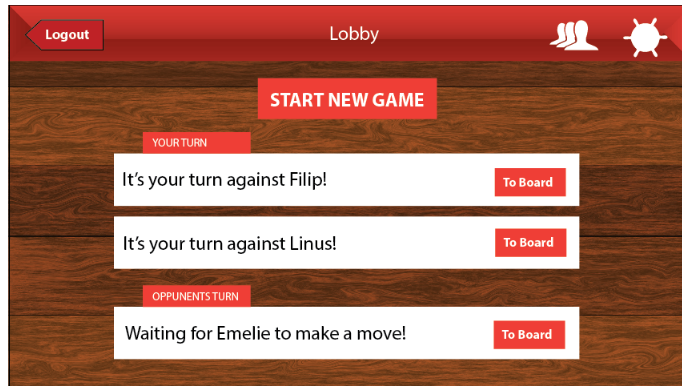

Figure 3.15: The main lobby of the game presented on a device running on iOS.

engines available that could be used with Unity i.e Photon Server [66] and Player.IO [67]. However, the game developed in this thesis does not use Unity's networking features or any other networking engines. The reason for this is that those networking features are mainly intended for active multiplayer applications where two or more players interact in real time. Since this is not the case in the game in this thesis, using those networking features would be excessive and inefficient. Instead, the game's networking is built mostly from scratch. The overview of the server architecture is presented in Figure 3.16. In the client game application, information is sent to a PHP Script located on the web host Binero [68]. The PHP Scripts interpret the information sent from the game application before either manipulating the database or retrieving information from it. Depending on whether the PHP Script performed a manipulative or retrieving action on the database, the PHP Script sends information to the game application. This information contains either whether the manipulation was successful or not, or relevant information retrieved from the database. Unlike real time multiplayer networking, the client-server communication is not in real time. Instead, information between the client and server is shared upon requests or in time intervals. For example, the states of the different squares on a board is updated once every ten seconds but also each time the user access the specific board. The server architecture in this game is completely platform independent and does not need to know whether a client is using an iOS or Android platform.

The server needs to handle features regarding user credentials, match making, high scores and match status. A list of networking and multiplayer features in the game is displayed in Table 3.2. The initiation of the database tables, along with the action on them through PHP scripts, is made with the relational database management system MySQL [69]. For managing and viewing the database, the web browser tool phpMyAdmin [70] was used.

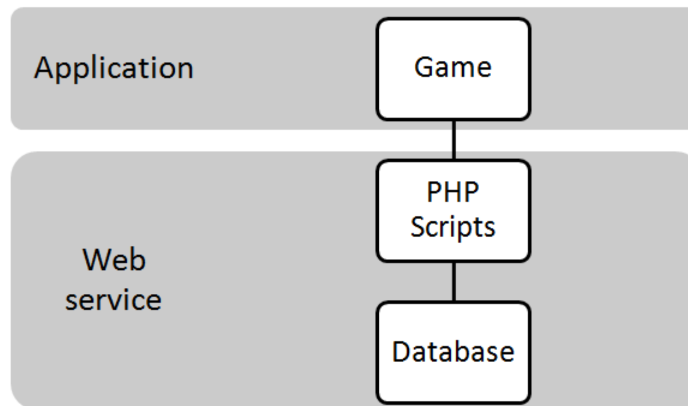

Figure 3.16: The Mecanim's Avatar Configuration, mapping the character's bone structure to Mecanim's bone structure.

Table 3.2: Networking and multiplayer features

type	feature
user	Create new user, return whether successful or not
user	Login, return whether successful or not
user	Add friend by name, return whether successful or not
user	Accept friend request
user	Challenge friend
user	Accept a challenge from a friend
user	Start new game against a random opponent
user	Retrieve board and square information
user	Play square, return whether allowed or not
matchmaking	Find random open game and if one exists, join it
matchmaking	If no games exist, create a new open game
matchmaking	When challenging a friend, create a new game
matchmaking	If accepting a challenge from friend, join that game
matchmaking	Return game information regarding next turn and last action
matchmaking	Return game information regarding next turn and last action
highscore	Return the user's high score for a specific level
highscore	Update the user's high score if improved
highscore	Return the world high score for a specific level
highscore	Update the world high score if improved
match	Update and show the status of a recently played square and present the score
match	Update user turns and match status when a square has been played

3.2.3 Deployment

Android

In order to develop on Android, the Android SDK was installed. When the installation was complete, the device that would be used was in need of drivers. On

Windows, the drivers sometimes gets installed automatically otherwise they will be needed to be downloaded manually. On Mac OSX there is no need of installing drivers. On the device, USB Debugging was needed to be turned on, which only can be enabled if the user is a developer on the device. When using an Android version from Android Jelly Bean, developer options are hidden and will only appear if the user taps on the build number in the settings menu six times. When the SDK and device had been set up, Unity needed to recognise the Android SDK location. This was done by specifying the SDK location in the Unity settings. When exporting the application to an Android device, a number of settings were to be changed. First of all, there must be a company name and a product name specified. Then an icon for the game should be added. Also, the orientation of the game was to be specified, whether or not the status bar should be hidden or visible had to be chosen, and the minimum Android API level was chosen. There was also a lot more specific device configurations like device filter, install location and graphics level. When all the settings were made and the device was plugged in through an USB cable, building the application resulted in the game starting on the device.

iOS

To be able to run the application on a device running iOS, a bit more setup was required. First of all a developer license was needed to be able to push the application onto the device. The operating system needs to be upgraded to the latest version, since the latest XCode version was only supported in the latest OSX. When the newest OSX was up and running it was time to download the latest iOS SDK from the iOS dev center. The latest iOS SDK also included the latest version of XCode. The next step was to connect the iOS device that would use the application and launch XCode. XCode will detect the plugged in device and the device should be selected to use for development. In XCode, an identifier code was not presented. In the iOS dev center, a new device was added, holding the identifier code and the corresponding device name. In the iPhone developer program portal, a certificate was created with the application id and the device identifier code. This certificate was imported and installed in XCode. Finally, after editing the settings inside Unity, the application was built with Unity, resulting in an XCode project. The XCode project was then opened inside XCode and built to the connected iOS device.

3.2.4 Performance

A problem with exporting to an Android device was the native Unity terrain. On devices with Tegra and Mali-400 GPUs (Graphics Processing Unit), the terrain got very pixelated as shown in Figure 3.17. On other devices with different GPUs using the same terrain, the terrain had the correct appearance as can be seen in Figure3.18. This hardware problem occurs because of the Tegra and Mali-400 GPU's having too low precision. To solve this issue, the camera in the game had to be set at a higher distance from the terrain to make the eventual pixelisation less visible to the user. Also, the resolution of the terrain texture was lowered.

When the native terrain asset in Unity had been replaced with a plane object

holding a texture instead, the fps increased a lot. The drawback of this was that the plane now used as a terrain did not look as good as the native terrain because of its flat layout.

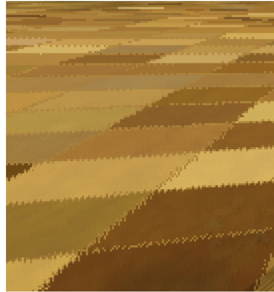

Figure 3.17: A part of the Unity terrain as seen on a device using a Tegra or Mali-400 GPU.

Figure 3.18: A part of the Unity terrain as seen on a device using a precise GPU.

3.3 Character Design

3.3.1 Base modelling

The base model was created in Blender. First of all, sketches of the concept was not made. Instead, reference images were found and used as a background in Blender [71]. After the background image had been inserted in the front view and the left view, a vertex was placed in the centre of the belly on the background image. From this vertex, a line was extruded up to the head. The extruded line contained several breakpoints. When the centre line was done, a mirror modifier was added in order to make the mesh symmetrical on both sides. When extruding from the neck out to one arm, the other arm was identically extruded. The extrude was made out to the arms and legs. When all the lines were completed, the skin modifier was added. This added a cuboid around the line. To make this smoother, a subdivision surface

modifier was added. With the modifiers placed, scaling the points from the lines in the beginning in x, y and z-axis were made. This resulted in the base mesh looking like Figure 3.19. In order to create a base hand, the end points on the arm was extruded to a hand and fingers. Again, the skin modifier and subdivision surface modifier was used. The same tools were used to extrude and create base feet.

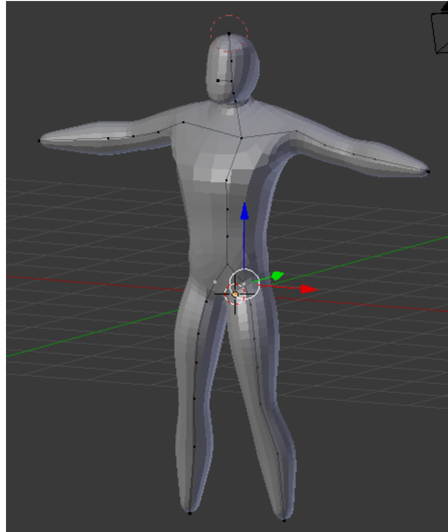

Figure 3.19: The base mesh created in Blender.

3.3.2 High polygon modelling

When the base mesh was finished, the next step was to import the base mesh into Sculptris. Here, the head was sculpted using different tools such as crease, scale, draw, grab, smooth and pinch. Since the symmetry option was turned on, only half of the head needed sculpting. All details on the head, except for the eyeballs, was sculpted directly on the base mesh. The eyeballs were added in Scluptris by creating two new spheres which were placed in the eye sockets. Using Sculptris, the different body parts were sculpted in order to acquire accurate body proportions. The finished sculpted head can be seen in Figure 3.20.

3.3.3 Retopology

When finished with the sculpting process in Sculptris, the high polygon mesh was imported into Blender for retopologising. The imported high polygon mesh consisted of 174194 triangles, which had to be vastly reduced in order for the character to properly function on a mobile platform. First, a plug-in called Bsurfaces was activated in Blender. When Bsurfaces had been activated, a new mesh in form of a plane was added. The plane was then snapped to the high polygon mesh. From this plane, using the grease pencil inside Blender, Bsurfaces would fill in a space on the plane with quads. In the more detailed areas such as around the eyes, mouth, nose and ears, a lot more quads was drawn compared to on areas which did not have

Figure 3.20: The high polygon head, sculpted on the base mesh in Sculptris.

as much detail, for instance the skull. A lot of the quads were made by extracting other quads, since it was difficult to draw everywhere. When two quads met, a join of the vertices was needed to be done to make the quads stick together.

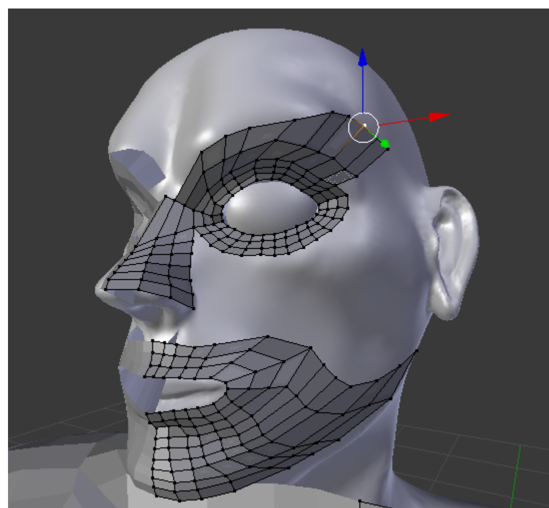

Figure 3.21: Retopologising in Blender using Bsurfaces

3.3.4 UV mapping

When the retopology process was completed, an UV map was created. In order to create the UV map, seams were needed to be added to the character in order to

split the flat image into different pieces. In Figure 3.22, the seams are marked in red, and resembles where the image will be cut. The seams were created along with the edges of the quads.

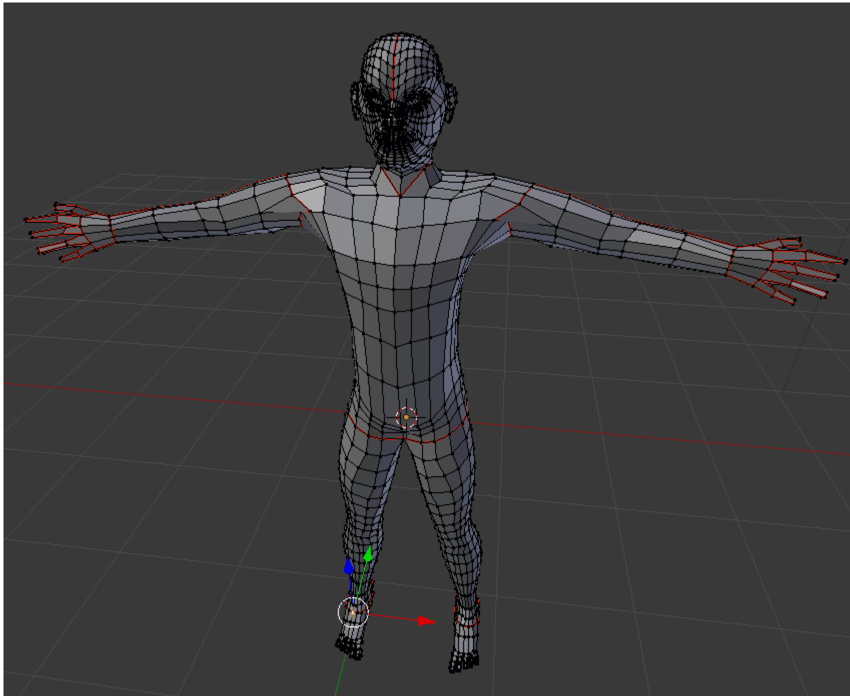

Figure 3.22: An UV mapping of the character. The seams are shown with a red colour.

When creating the normal map, both the low polygon mesh and the high polygon mesh were selected. Then, the Baking tool in Blender was used to create the normal map. This resulted in a normal map with sharp edges. The smooth edges tool was needed for the normal map to become smoother, which can be seen in figure 3.23.

3.3.5 Texturing

The texturing of the character was made by painting the character directly in the 3D view in Blender and generating the texture map based on the previously created UV map. The result can be seen in figure 3.24

3.3.6 Rigging and weights

When the character was created, there was just one final step before it could be imported into Unity and used with animations, namely rigging of the character. The whole rigging process was made in both 3ds Max as well as in Blender. The first attempt was made in Blender using the Rigify plug-in. The skeleton was first placed inside the character using manual fitting. Then, the generate button was pressed, making Rigify automatically fitting the skeleton inside the character and also skinning the character to the bones in the skeleton. The second attempt was

Figure 3.23: The baked normal map of the character.

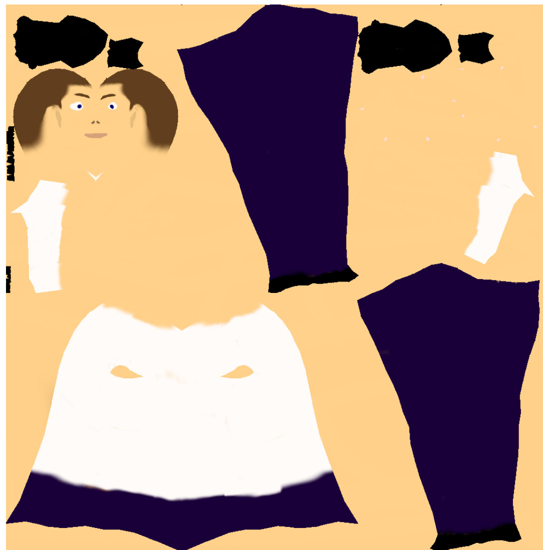

Figure 3.24: The baked texture map of the character.

made in 3ds Max. Here, the Biped tool was used. The biped skeleton could be seen next to the character model in Figure 3.25. The predefined skeleton was then manually fitted into the character before the Skin modifier was added. With the envelope tool in the Skin modifier, the vertices had to be adjusted so that they corresponded to the correct parts of the bones in the skeleton as shown in Figure 3.26.

3.3.7 Other Software

The Project Pinocchio was also tested to compare the manually created character with an automatically generated character. To use the Project Pinocchio, an Autodesk account was needed. The tool is only available online and is used in a web

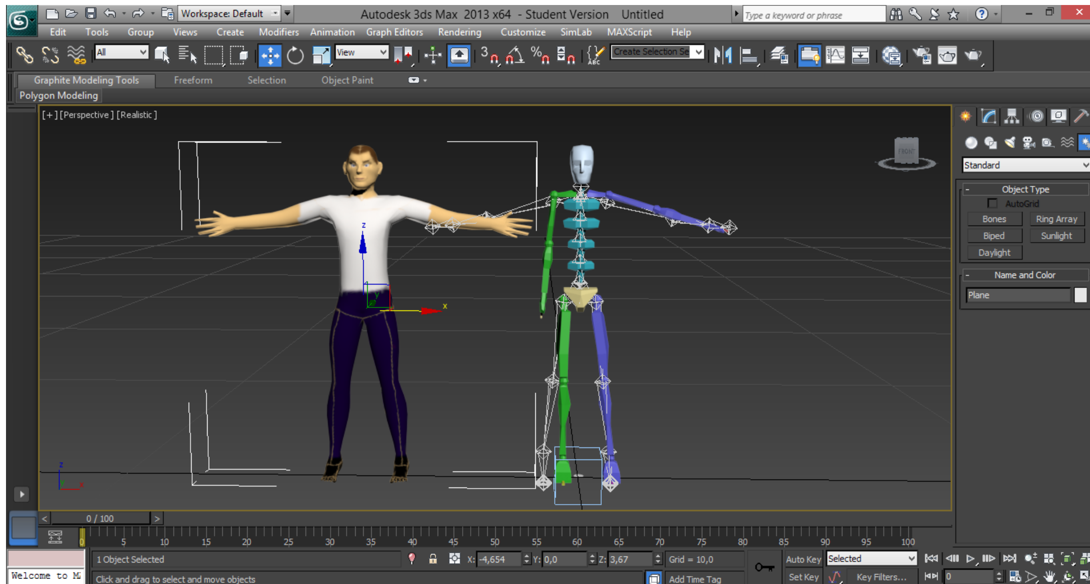

Figure 3.25: Fitting the skeleton inside the character

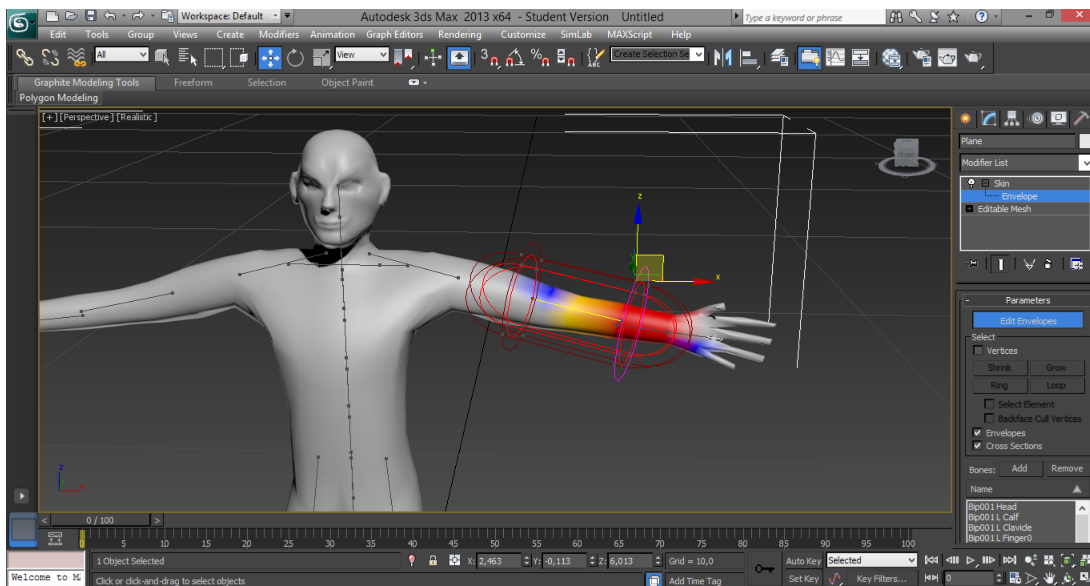

Figure 3.26: Changing the envelope on the arm

browser. The account can be created instantly and used to login. When logged in, a new character was created. First a reference character was selected. This reference character was selected from a list of about 20 previously created characters. When the reference character had been selected, a window with the option to change the appearance of the character, see figure 3.27 was presented. Here, parameters corresponding to the differences between the chosen reference character and another reference character was edited. When the body and face details were created, a hairstyle and hair colour was selected. The next step was to select clothes for the character. Here, a variety of shirts, pants and shoes could be picked, all with different styles and colours. When the character was completed and ready for export, the

height of the character, a low polygon resolution and a quad geometry was picked as options. Also, a normal map, a texture map and a specular map was chosen to be exported with the character. Finally, the export process was completed resulting in a FBX file ready for use within Unity and Mecanim. Figure 3.28 shows a comparison of the manually created character and the character created with Project Pinocchio.

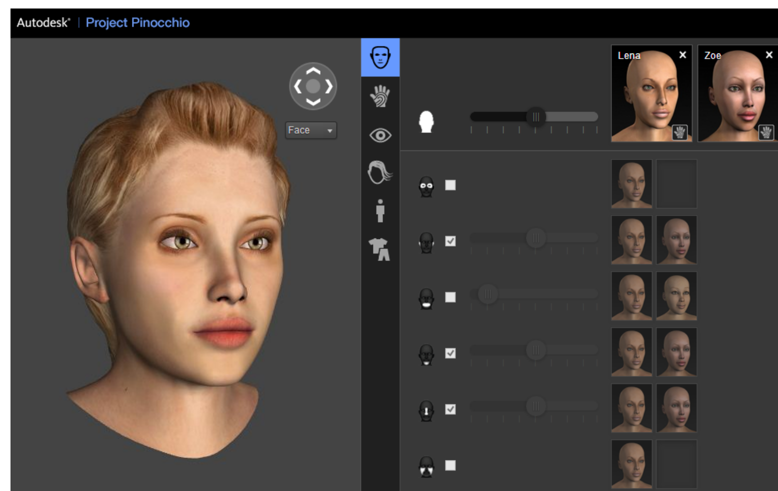

Figure 3.27: The editor in Project Pinocchio

Figure 3.28: Comparison of the manually created character to the left with the character created in Project Pinocchio to the right in the Unity editor

3.4 Animations

3.4.1 Motion Capture

This thesis deals with two different motion capture devices and techniques which are described in greater depth in Section 2.3.1, namely Microsoft Xbox 360 Kinect and Qualisys Motion Capture System. Here, the method and workflow of how the character animations were recorded and processed is described. The motion capture using Kinect was made with both one and two Kinect devices in order to compare the quality between both options.

Kinect

The first motion capture setup was using one Kinect device. First, the Microsoft Kinect SDK was downloaded and installed along with Microsoft .NET Framework 4.0. When using Windows 8, this was a required step for getting the Kinect devices to work with the iPi Soft programs iPi Recorder and iPi Mocap Studio. Then, the programs iPi Recorder and iPi Mocap Studios were downloaded and installed. A 30 days trial was activated. Figure 3.29 shows how the environment was setup for the motion capture session using a single Kinect device. The free software iPi Rec order lets the user record a depth and RGB (Red, Green, Blue) video of an actors performance. Figure 3.30 demonstrates how a live frame looks like when recording an actors performance with the iPi Recorder. An area in the frame that is covered in a dark blue colour tells that the Kinect sensor is capable of determine precise depth data from that area. A yellow area however tells that the Kinect sensor can only determine some or no depth data in that area. The purple area seen on the actor in the frame means that the depth data determination in this area is not as good as it is in front of the actor, however, it is still quite good compared to the red and yellow areas behind the actor.

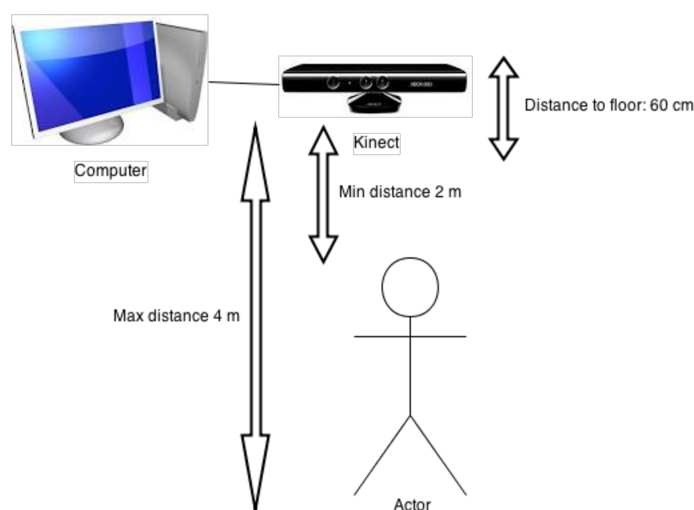

Figure 3.29: The setup of the environment using one Kinect.

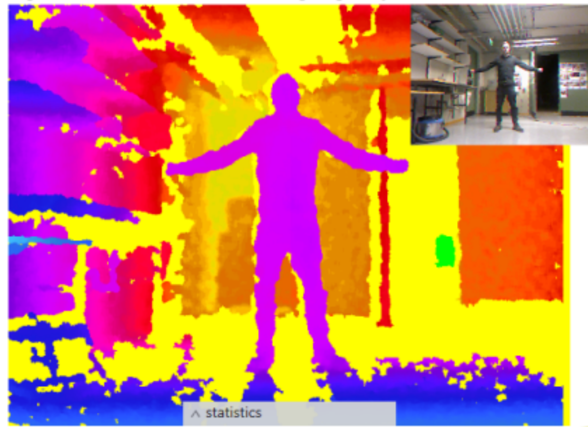

Figure 3.30: A frame in an iPi Recorder session showing the depth and RGB video.

When a session had been recorded with the iPi Recorder, it was saved for use in the iPi Mocap Studio. The iPi Mocap Studio analyzes the depth data recorded in order to track the actor's motion to a predefined skeleton. The predefined skeleton is in a standard T-Pose when initiated, therefore it is recommended to begin each session with the actor also in a T-pose. An example of how this can look is shown in Figure 3.31. In the figure, no actual motion tracking has been made yet since the purple actor depth data and the predefined skeleton is not aligned. Aligning these was done by moving and resizing the skeleton until it was fairly aligned with the depth data of the actor as shown in Figure 3.32. iPi Mocap Studio provides a feature called Refit Pose which automatically aligns the skeleton with the actor depth data, however this feature was best used after a fairly good manual aligning had been made. When a good aligning was made, the software started to automatically track the motion frame by frame, using the previous frames for prediction.

When a motion tracking session with iPi Mocap Studio was completed, the take was exported as a Biovision Hierarchy (BVH) file which is a popular character animation file format. The BVH file was then imported into the Autodesk MotionBuilder software as shown in Figure 3.33. After import, the imported animations was already playable inside the MotionBuilder software, however they were not ready for use with Unity's Mecanim. In order to achieve this, the skeleton shown in Figure 3.33 must be defined. To do this, the Skeleton button in Character Controls was pressed as shown in Figure 3.34. Then, the bones in the skeleton needs to be mapped to the definition of the character. This is simply made by selecting a bone of the skeleton and right-clicking on the corresponding part of the body of the character in the definition. A successfull mapping is shown in Figure 3.35. The final step that was done before exporting was characterising the skeleton. This was done by dragging the Character template from the Asset Browser on a bone of the skeleton and selecting the Biped option in the pop-up window that appears. This step is shown in Figure 3.36. Finally, the file was saved as a FBX file.

This thesis also examines the use of two Microsoft Xbox 360 Kinect devices for recording with iPi Recorder. Some of the steps are very similar to the case when using only one Kinect but additionally the process requires a calibration step. The

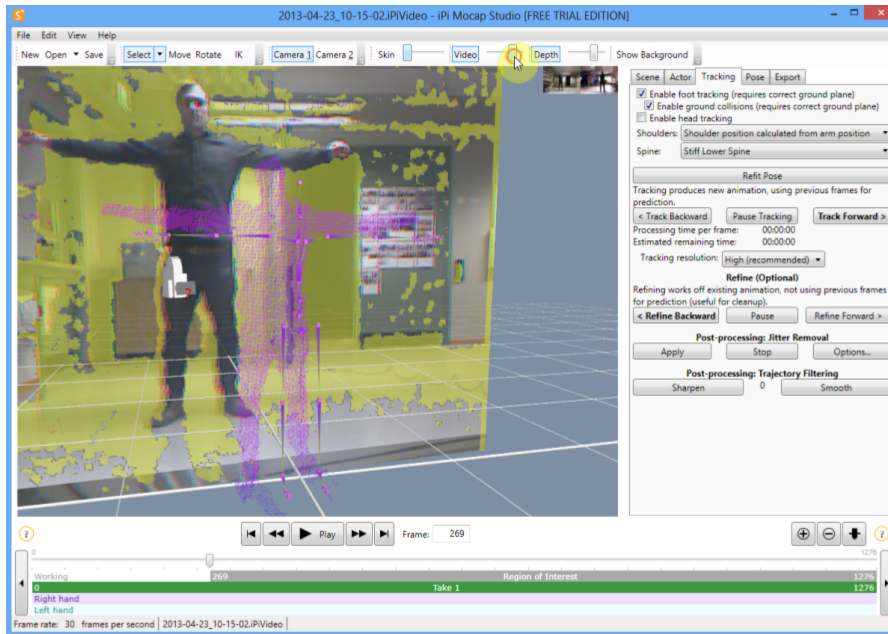

Figure 3.31: A frame in iPi Mocap Studio showing the RGB video (back), purple actor depth data (middle) and the pre-defined skeleton (front).

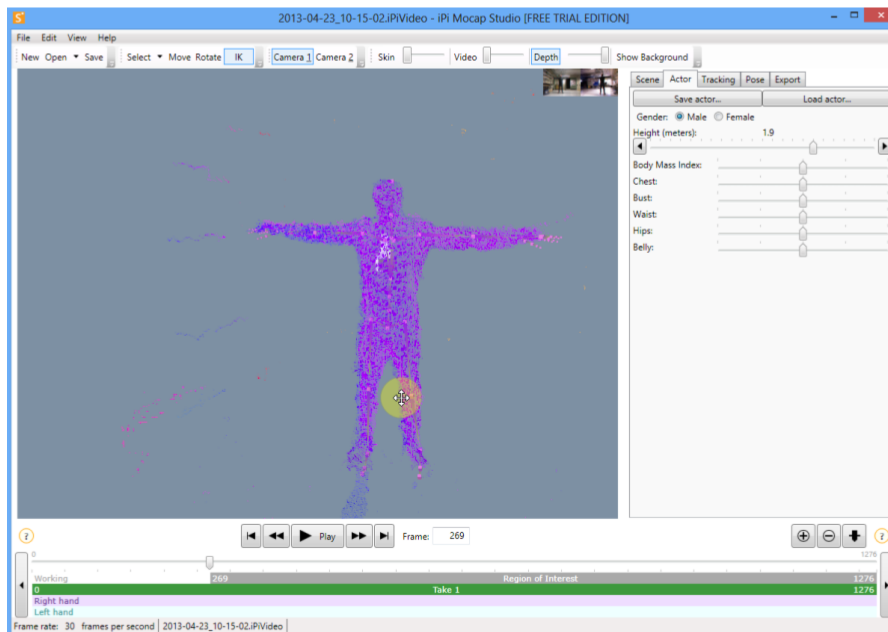

Figure 3.32: A frame in iPi Mocap Studio showing the actor depth data and the skeleton aligned together manually.

calibration step is needed for the software in order to calculate the distance and angle between the both devices. The environment of the setup using two Kinect devices is shown in Figure 3.37. The calibration was made by recording a special calibration session in the iPi Recorder. In order to let the software calculate the angle and distance parameters between the Kinect devices, the software looks to

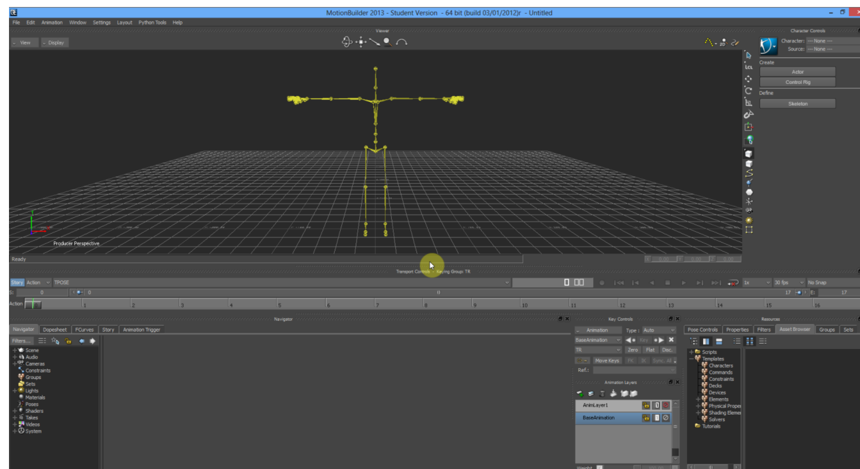

Figure 3.33: Autodeks MotionBuilder after importing one or several BVH files. The skeleton in the picture is automatically created and corresponds to the skeleton present in the iPi Mocap Studio.

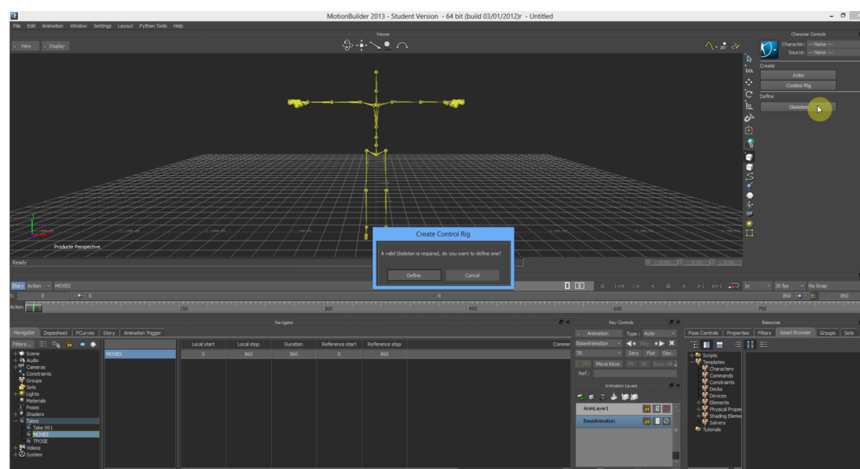

Figure 3.34: Creating a Control Rig using the Skeleton button in Character Controls.

identify a square like object which should be visible in both sensors. A side panel of a computer case was used as the square object in the session to let the software identify the object and, based on the object's motion and location in the space, determine the angle and distance between the Kinect devices. Figure 3.38 shows how a frame in the calibration session inside iPi Recorder looked. In the frame shown in the figure, the computer case panel is clearly visible in both Kinect sensors, allowing the software to easier identify the object. In the calibration session, the object was also waved in different directions and angles, allowing the software to make more precise calculations of the position of the object. The calibration recording was then saved for use in iPi Mocap Studio where a calibration project was created. In order for the iPi Mocap Studio to use the recorded calibration video, a region of interest, that covered a set of frames where the computer case panel was visible in both Kinect

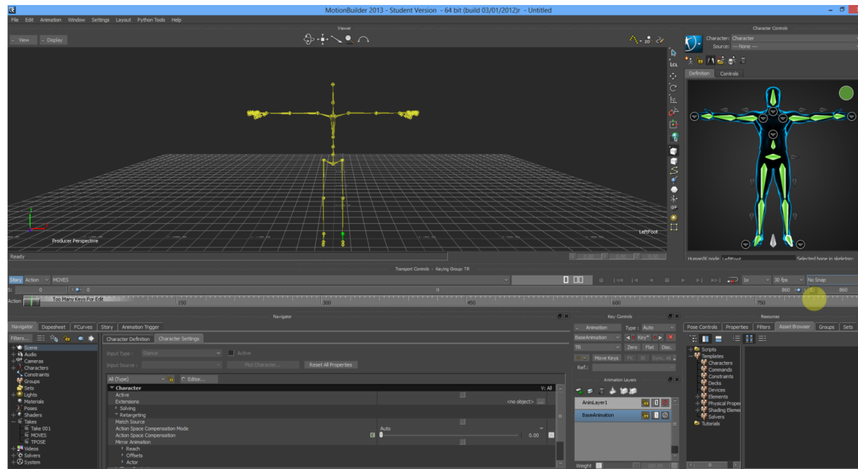

Figure 3.35: A successful mapping between the skeleton and character definition.

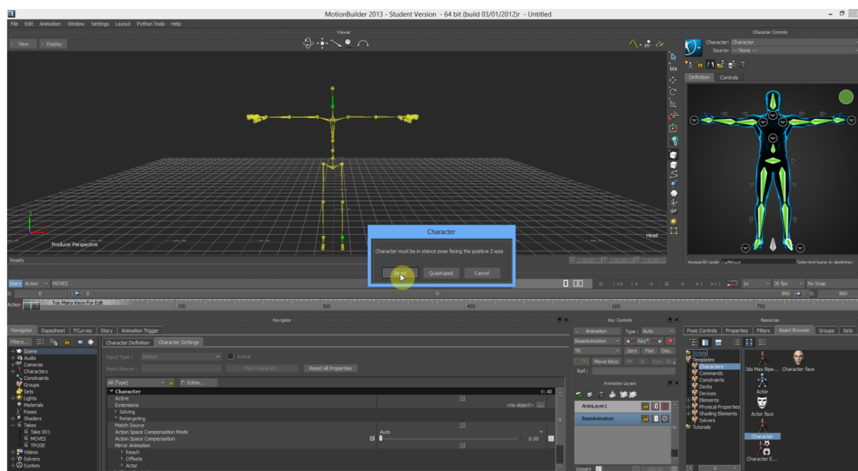

Figure 3.36: The characterisation using a biped character template.

sensors, was selected. Then the system started its calibration process. Figure 3.39 shows a frame in the calibration process with iPi Mocap Studio. In the figure, the green dots is automatically showed when possible for the sofware to determine the calibration object's corners and centre. The calibration was then saved as a scene file within iPi Mocap Studio.

When the calibration was done, as for with one Kinect device, the iPi Recorder was again used to record the actor's depth video session. A frame showing the actor performing a move in the recording session is shown in Figure 3.40. Note that there is now two depth videos compared to only one when using a single Kinect device. When the session was recorded, the video file was saved and then opened in iPi Mocap Studio. Here, an Action project was created using the calibration scene file that was saved in the calibration step. The next steps for skeleton tracking, export and characterisation in Autodesk MotionBuilder are identical to when using only one Kinect device.

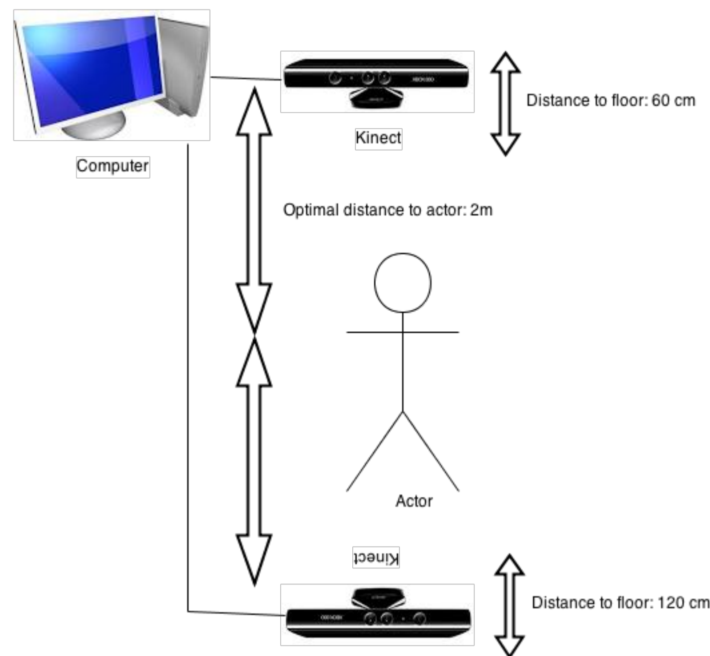

Figure 3.37: The environment setup using two Kinect devices with an angle of 180 degrees between the devices.

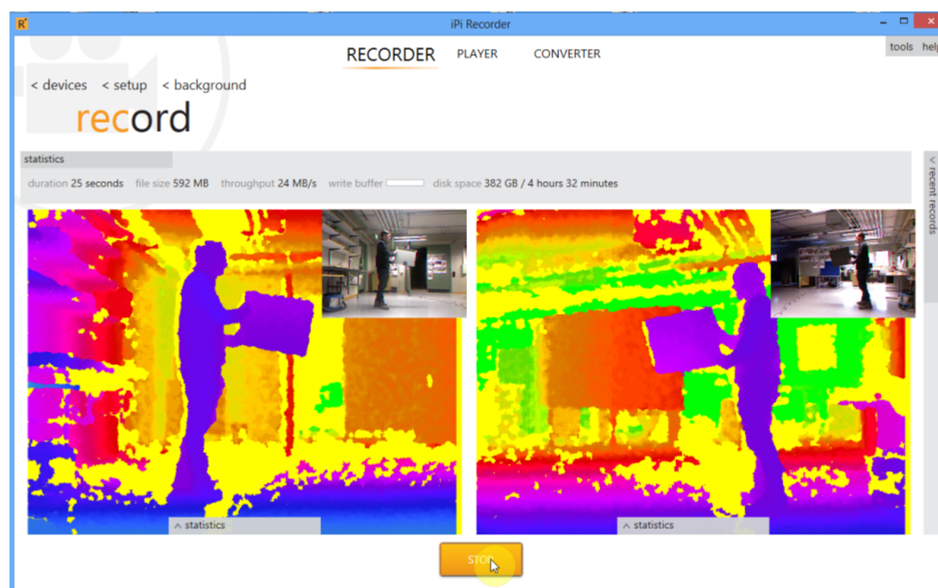

Figure 3.38: A frame in the recording of the calibration process.

Qualisys

The Qualisys Motion Capture System used in this thesis consisted of eight cameras. The position, angle and height of each camera can be seen in Figure 3.41. Before any recording was performed, Qualisys Track Manager, the software used to record and track the motion, needed a calibration step. The calibration was done by moving and rotating a calibration wand, explained and showed in Section 2.3.1. The actor had a

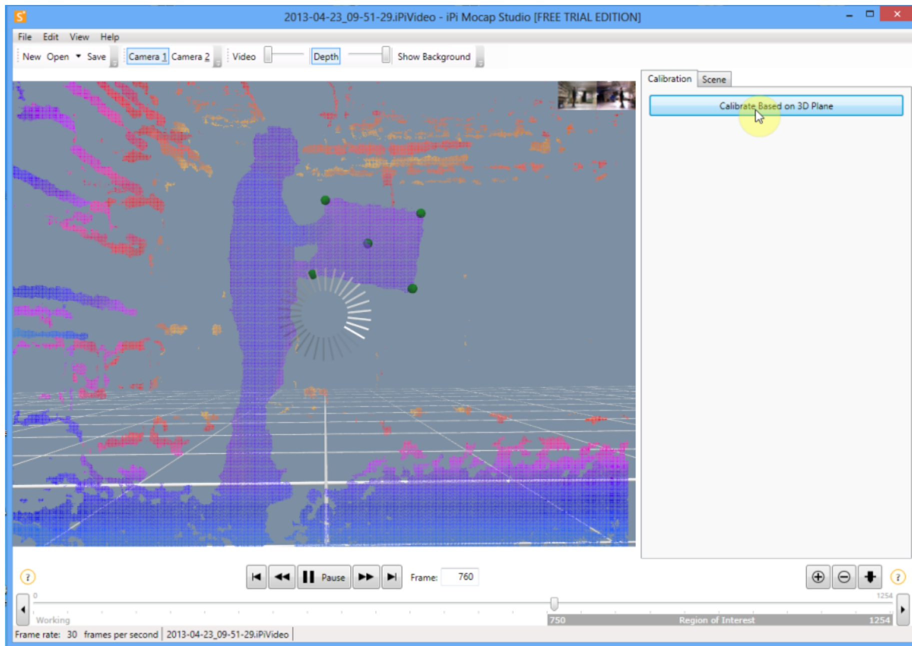

Figure 3.39: A frame in the iPi Mocap Studio processing the calibration data.

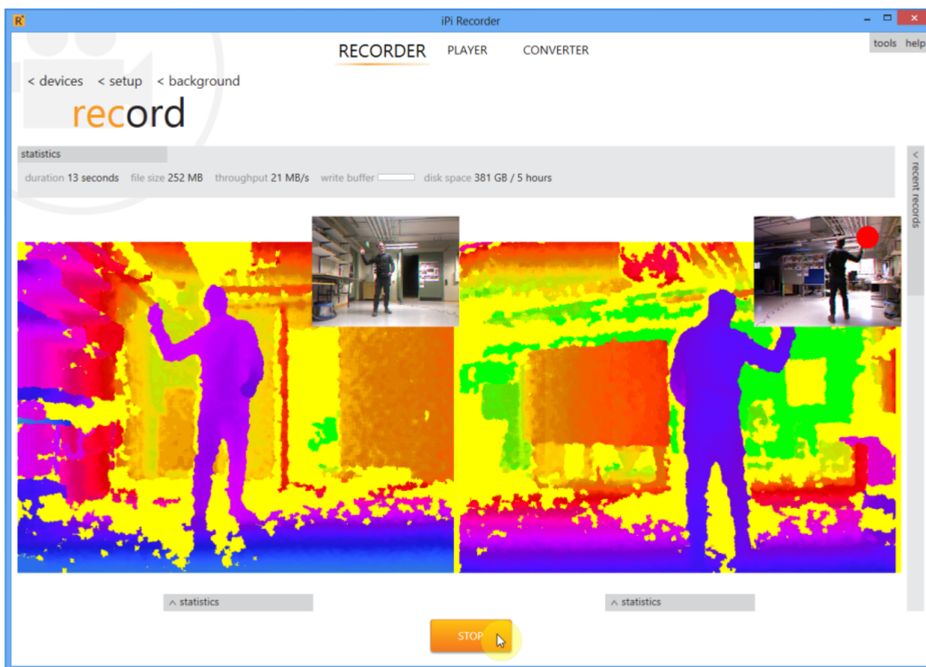

Figure 3.40: A frame of a record session in iPi Recorder using two Kinect devices.

marker placement that was made by following a marker placement guide [72]. Figure 3.42 shows the reference picture of the marker placement that was used when placing the different markers on the actor. When the recording sessions were completed, a skeleton tracking similar to the skeleton tracking when using the iPi Mocap Studio, was performed. Instead of aligning depth data with the bones of a skeleton as with

the case with the Kinect capture, the Qualisys system identified each marker in the frame and mapped them to a skeleton. The skeleton was not predefined as with the case with the one used in iPi Mocap Studio. Instead, the connection between markers were defined in order to create bones between the markers. For instance, the markers placed on the heel, ankle, innermost toe and outermost toe were connected in Qualisys Track Manager to resemble a foot. Quite often, the motion capture system lost track or identity of one or several markers in a frame. When a marker was interpreted as unknown, the user had to identify the marker manually. For instance, in Figure 3.44 the system was unsuccessful with determining the identity of the marker placed on the ankle of the actor, resulting in a lost bone connection between the foot and leg but also between the ankle, innermost toe and outermost toe. To correct this, the system needs to be told the true identity of the unidentified marker. In Figure 3.43, the marker has now been correctly identified and the correct bone connections are applied.

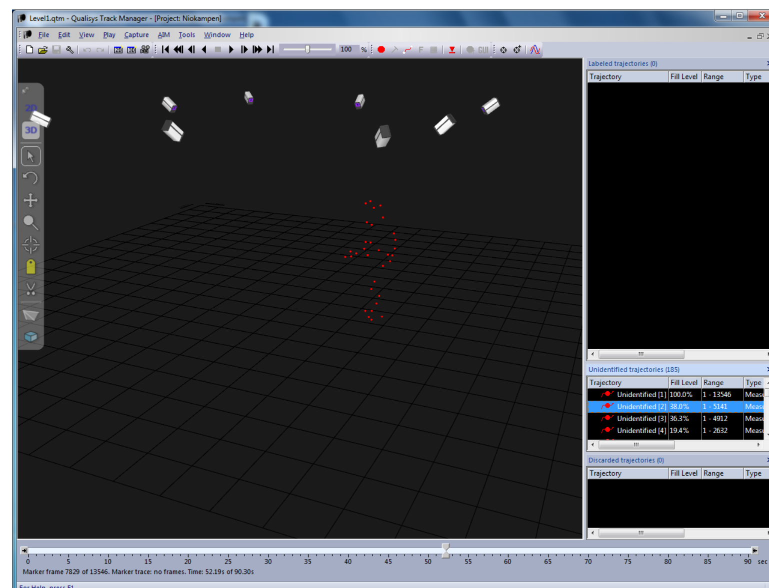

Figure 3.41: Qualisys Track Manager showing all of the eight cameras and the markers (red) placed on the actor.

When a session had been recorded and the actor's motion applied to the skeleton, the optical data was exported as a C3D file for use in Autodesk MotionBuilder. In MotionBuilder, the optical data was mapped onto an Actor using the Actor tool. The optical data applied to the Actor, along with the mapping between the Actor reference marker structure and the markers used in the recording is shown in Figure 3.45. The structure of the Actor is not compatible with Mecanim because of the Actor not being characterised. Instead, the rigged skeleton described in Section 3.3.6 was imported into MotionBuilder where it was characterised and had its bones defined in the Character Controls as shown in Figure 3.46. Then the characterised skeleton was modified so its Input Type was set to Actor. This had the effect of aligning the skeleton to the Actor. The next step was to plot all the animation takes to skeleton using the Actor as the reference. This was done by using the Bake (plot) Skeleton tool in the Character Controls. Now that the animation takes were

Figure 3.42: The reference picture illustrating how to place the markers on the actor.

Figure 3.43: A frame in the session where the Qualisys system is unsuccessful in identifying the identity of the marker placed on the heel of the actor.

applied to the Mecanim compatible skeleton, the Actor could be deleted from the MotionBuilder project. Finally, the project was saved as an FBX file for use in Unity.

3.4.2 Mecanim/Animation transitions

When a FBX animation file was imported into Unity, the type of animation had to be defined as a Humanoid in order for Mecanim to create a successful mapping of the bones. In Mecanim, each animation that was recorded in a session had to be manipulated before it was to be used within the game. First, a source take containing a full recording of a certain type of motions, running motions for instance,

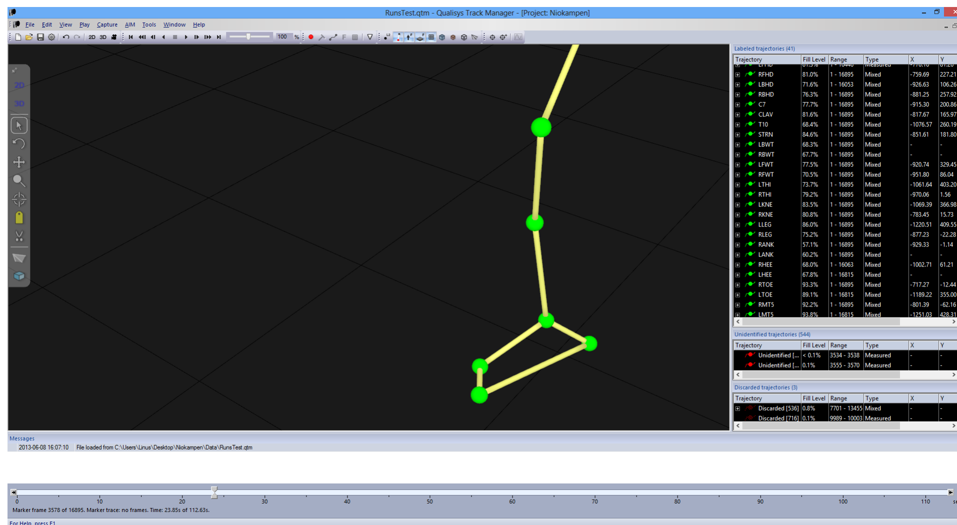

Figure 3.44: A frame in the session where all the markers are identified, resulting in a correct connection of the bones.

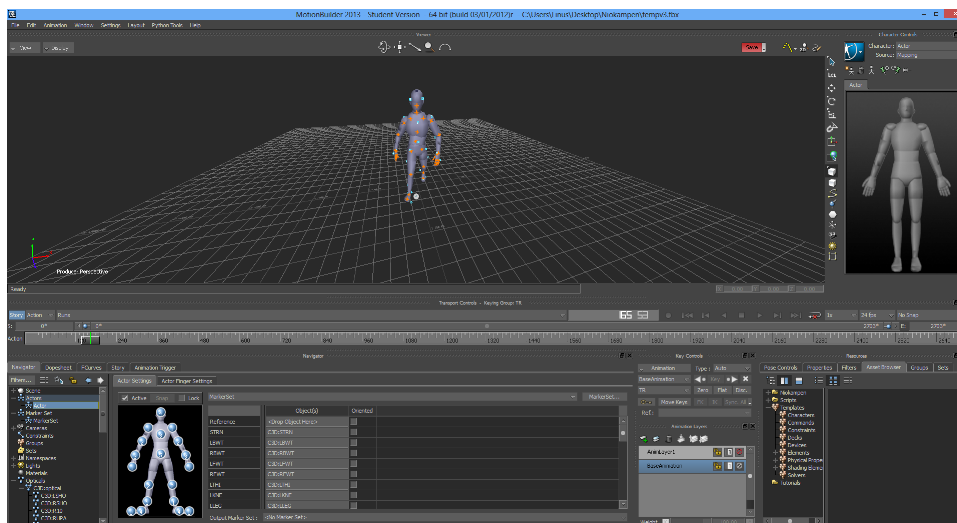

Figure 3.45: A frame in the session where all the markers are identified, resulting in a correct connection of the bones.

was selected. Based on that source take, different smaller clips were extracted, each one representing a specific motion to be applied to a character. An example of such a motion, that was extracted from the running motions source take, was the forward run motion. In Mecanim, a starting and ending frame which would represent the start and end point of the motion that was to be extracted, had to be chosen from the source take. When the start and end frames have been selected and if the animation is to be looped, Mecanim would tell whether or not the selected clip has a loop match based on three different parameters. A green loop match is shown when the rotation, y position and xz position of the character using the animation clip is similar for both the start and end points. Figure 3.47 shows a clip where the character runs forward, extracted from the source take holding all the run motions. In the figure, the top three loop matches lights are green, however the light representing the xz

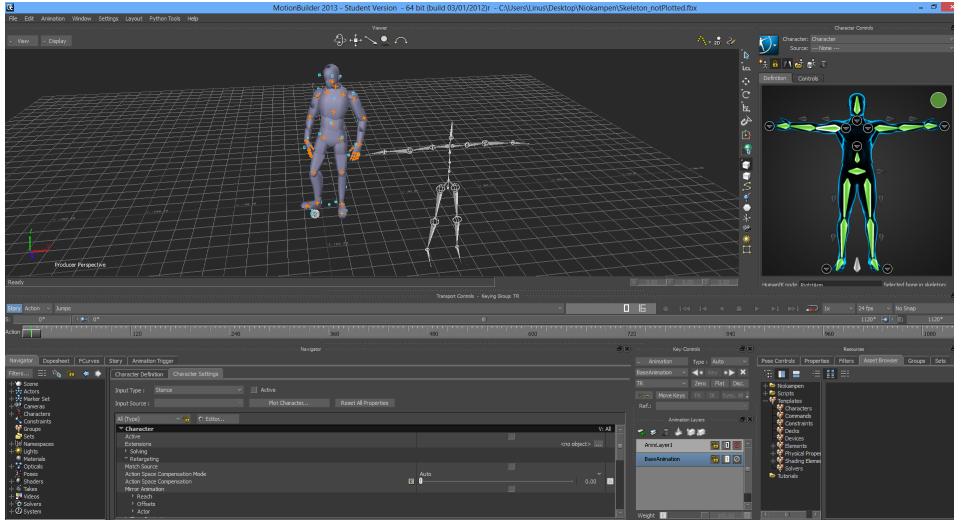

Figure 3.46: A frame in the session where all the markers are identified, resulting in a correct connection of the bones.

position is red. For a forward run motion this is a wanted behaviour due to the fact that the z position must be changed in order for the character to actually run forward in 3D space. This workflow had to be repeated for every single animation clip that was extracted from a source take.

For every level in the game, a different Animator component was created and placed on the character in the specific level. Figure 3.48 shows an example of how the states and transitions in the Animator component were created in the game. In the example, the Animator controls the character animations played in the Wall Jump mini game. The standard state of the character animation in this game is the Grounded state, active when the Grounded parameter is set to true and the magnitude parameter is close to zero. In the Grounded state, an idle motion animation clip is playing. From the Grounded state, the character will go into the Strafe state if the magnitude is greater than zero and if the character is still on the ground. If the character is not grounded, the state transitions into the InAir state where the character is performing a flying-like motion. From the InAir state, there might be transitions into the Falling, SmallJump and BigJumps states depending on what happens in the game. Different Animator components with the same principles was created for every mini game.

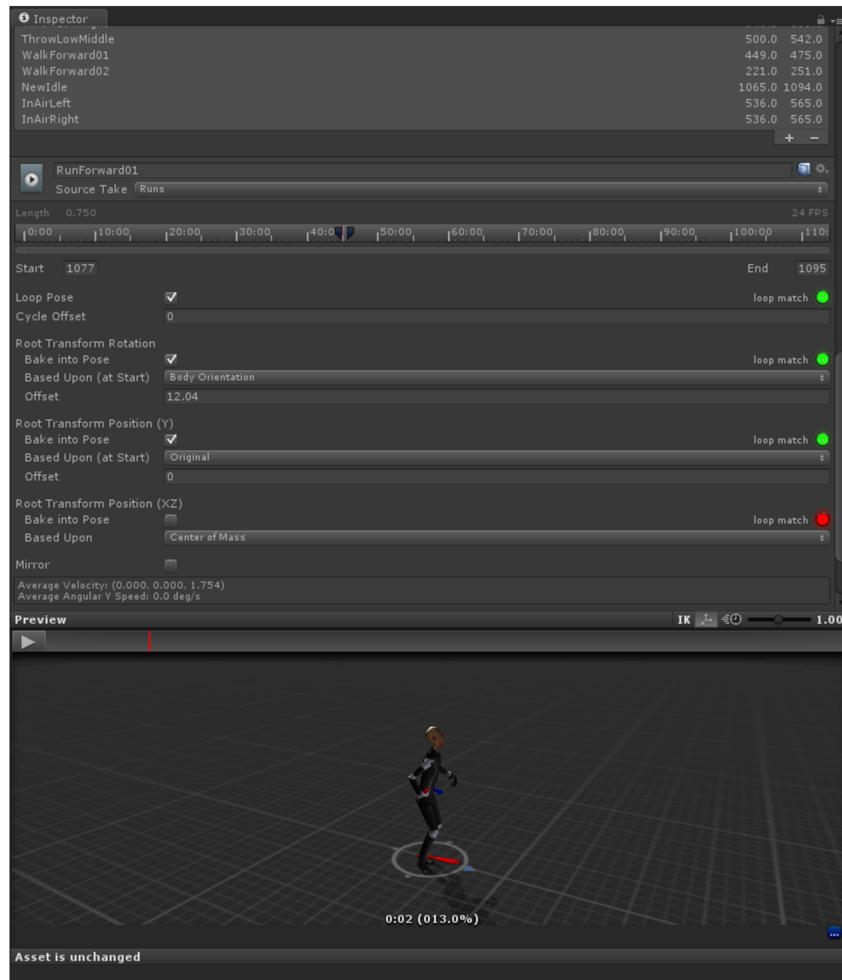

Figure 3.47: An animation clip showing a forward run motion. Notice the loop match lights, telling the user whether the pose is fitting for a loop or not.

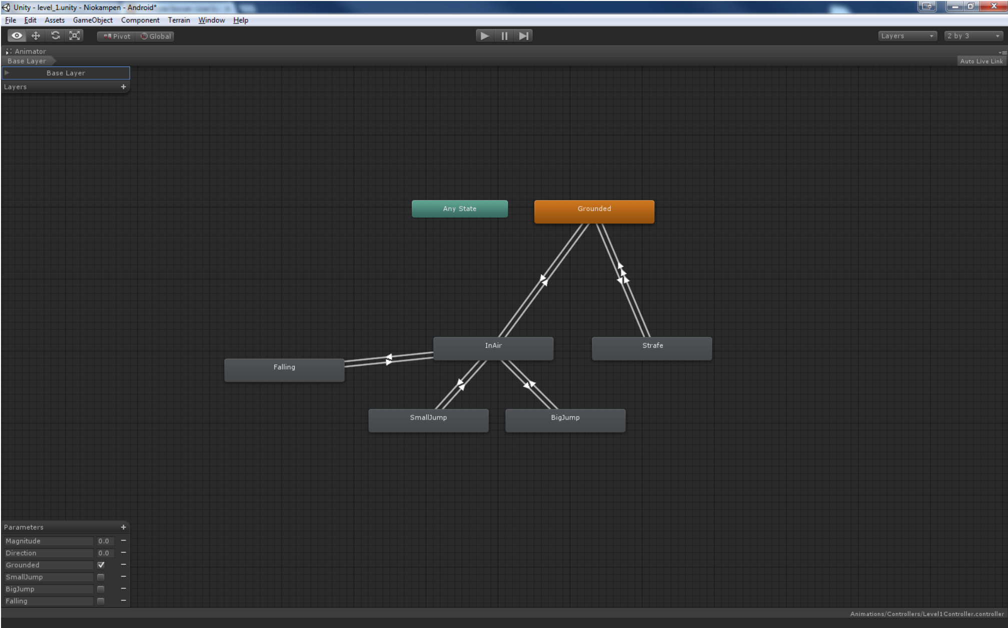

Figure 3.48: The Mecanim's Animator component showing the different states and transitions that were used in the Wall Jump mini game.

Chapter 4
Conclusion and Discussion

4.1 Animation

When using a single Kinect device as the motion capture system, the workflow was very easy and straightforward. The biggest problem with using only one Kinect device was the loss of depth data when one or several of the actor's body parts were hidden behind other body parts. When this happens, a single Kinect is unable to determine the position and motion of the hidden bones, resulting in very awkward looking animations. On the other hand, using only one Kinect device allows for an easier environment setup due to a small amount of required space.

Using two Kinects, facing each other with 180 degrees between them, almost eliminates the problem of depth data loss due to hidden bones. However there are cases when the problem is still present. If for example an actor faces one of the Kinects, putting the arms in an arm-cross in front of the body, the Kinect device behind the actor will not register any of the arms. The Kinect in front of the actor will have a hard time calculating the arm that is hidden by the other arm. With that being said, using two kinects greatly reduces such errors compared to when only using one kinect but it should be noted that such errors might still exists in a motion capture session using two devices. Also, the calibration steps needed when using two Kinects are quite time-consuming. In a situation where the environment is changed, for instance if one of the Kincets are moved, the calibration process must be done all over again in order for the system to adapt to the new environment. When using two Kinects, there are more requirements on the environment in terms of size and structure of the room. When only one device is used, a smaller part of the room needs to be freed from space compared to an environment using two Kinects. On the other hand, the area where the actor's movement is fairly accurately registered, is greater when using two devices. Due to the dual depth videos provided by two Kinect devices, the system makes fewer guesses of predicted bone positions than

when using a single device, resulting in a an animation with greater quality and less jitter.

The Qualisys Motion Capture System used in this thesis puts even more requirements on the environment. Firstly, the size of the room needs to be bigger than the size of a room needed when using one or two Kinects. Secondly, managing and storing the entities used in the system could be a big issue when there is a portability factor, mainly due to its many cameras. The calibration process is not as time-consuming compared to the calibration used when using two Kinects. However, other parts of the process is more time-consuming compared to using Kinects. Depending on the quality of the recorded data itself, the tracking process could be more time-consuming than the iPi Mocap Studio's tracking process when using Kinects. The steps in MotionBuilder needed to make the animation ready for use with Unity and Mecanim are more and complexer for the Qualisys system compared to when using Kinects. This is because of the steps involving the Actor Tool and the Actor to skeleton baking of the animations, not needed when using the Kinect workflow. The quality of the animation however is greatly improved when using the Qualisys system and the jitter is vastly reduced.

The question about how a Microsoft Kinect compare to a Qualisys system for motion capture , when recording character animations to be used in a mobile game application, can now be answered. This thesis finds that using a motion capture system with two Microsoft Xbox 360 Kinects is the most feasible approach for an indie game developer. This conclusion is based on two major factors. First and foremost there is the economical factor. The Qualisys system used in this thesis had a price tag of approximately SEK1 000 000 which is out of budget for many developers. This can be compared to the cost of recording animations with two Kinects, using the same workflow and software as in this thesis. That cost is approximately US$5000 based on the sum of prices for two Kinect devices (US$100 each) , the iPi Motion Capture Basic Edition (US$595) and the Autodesk MotionBuilder (US$4195). Secondly, though the difference in quality and jitter between animations recorded with Qualisys and Kinect devices are noticeable, it is not as of huge importance due to the environment in which the animations are being played. If the animations were to be used within a PC game, video game or in an animated movie, the quality of the animations recorded with the Kinects would probably not be of enough quality without hours of manual work with improving the animation quality. However, in the mobile platform environment, this thesis conclude that the animations recorded with two Kinect devices are, without any manual improvement of the quality, good enough for indie mobile games. Animation recording with a single Kinect device would, by using the workflow and software used in this thesis, cost approximately US$4600 which is about US$400 cheaper than when using two Kinects. This solution is only recommended for animation recordings using very simple motions and since using two Kinects is only eight percent more expensive, this thesis recommend using two Kinect devices over one whenever the budget allows it.

4.1.1 Mecanim

Unity's Mecanim tool was very handy because of its many functions. First of all, importing an FBX file that was exported from MotionBuilder using the workflow explained in this thesis, required no manual bone configuration because of the Mecanim automatically recognising and mapping the bones. Also, Mecanim provided very usable features when editing and managing the different animation clips. If an animation session, recording different runs, had been captured using one single take, Mecanim could use this whole take in order to create different animation clips from it. For instance, a separate clip consisting of a looped forward run motion could easily be extracted from the larger source take. Mecanim also provided good loop tools for creating a good loop match in an animation clip. The transition and blending tools were also very powerful. The use of Unity's Mecanim reduced the number of other animation software needed in the workflow. The only negative aspect was found when the FBX file was to be exported from Unity back into MotionBuilder. The reason behind this action was because of how the optical data obtained from the Qualisys capture was stored in the FBX file. When the optical data had been used in MotionBuilder in order to create the Avatar in Skeleton (see Section 3.4.1) it became redundant. However, the optical data was not deleted from the MotionBuilder project before the FBX export to Unity was made. This resulted in a very large FBX file due to the optical data size. When this was found, the FBX file had already been processed with Mecanim, extracting and looping a large number of animation clips from the original source takes. The extracted animation clips is stored in the FBX file but are not exported in a way so that MotionBuilder recognises it. This led to that if the optical data was to be deleted from the FBX file inside MotionBuilder, the new exported FBX file would not contain the extracted animation clips, resulting in a loss of many hours of work. The optical data was actually present and shown as a file inside the Unity explorer but unfortunately it could not be deleted.

This thesis conclude that Unity's Mecanim is a very useful and handy tool that contains all the features needed in this thesis in order to get the animation behaviour correct. The possibilities of extracting and looping different animation clips from a source take reduced the amount of necessary software, for instance bvhacker. However, the management of the different files inside the the FBX file could have been better implemented bearing in mind that the optical data could not be deleted.

4.2 Cross-platform Development

One of the greatest challenges relating to the cross-platform development in Unity was the issue with the user interface. As using Unity's native interface is officially not recommended for use with mobile platforms, it came to using an external plug-in to develop the interface. Due to economical reasons the decision went to prime[31]'s UIToolkit. The use of the UIToolkit was quite complex due to several factors. First and foremost, the support and community around UIToolkit is very limited and the documentation of the C# implementation was not very deep. The process of

getting the toolkit up and running was also quite complex. On the other hand, many features were very powerful and the workflow was straightforward when using the TexturePacker software to create sprite sheets. The positioning and parenting features was very good when creating a user interface layout. Also, the UIToolkit included more advanced functions that could be used by the interface objects in a more dynamic environment such as colour blending and position animations. Though the toolkit came with most of the common components used in a interface, the lack of text fields were a disappointment. Instead, the native text fields had to be used in the same environment as the UIToolkit. prime[31]'s UIToolkit is a good tool for creating interfaces for mobile platforms but depending on the budget of a project, a purchase of the EZ GUI or NGUI should be in consideration.

A great advantage when developing a cross-platform application with the Unity game engine was the option to create a generic user interface within Unity. This generic interface developed in this thesis determines whether the device running the application runs on iOS or Android. The generic components is first created and then based on the platform, the platform specific components are added to the interface. This approach removes the necessity to develop two separate interfaces within an iOS IDE respectively an Android IDE. Although the generic cross-platform user interface approach is very effective in terms of time and lines of code, it lacks the graphical interface tools available in IDEs such as XCode and Eclipse.

Unity provided good features to generate form data to post to web servers with the WWW and WWWForm classes. Those classes were used with ease in order to send and retrieve information between the Unity application and the PHP scripts located on the web server.

Though this thesis has not made deeper investigation in the ShiVa3D and UDK game engines, it can be concluded that economically, Unity is the best choice.

One of the most difficult aspects we experienced related to cross platform development was the device deployment and testing. There is such a wide variety of different devices, especially devices running on Android, with different hardware and screen resolutions. In this thesis, all of the different devices and screen solutions have not been tested. Also, since the user interface in the game is not fully adapted to fit and look good on every device and resolution, there is no guarantee that the application and especially the user interface, will look good on a particular device.

When testing the application on an Android device, the process was very straight forward with a single build deploying the application to a device connected via an USB cable. A more complex and time-consuming process had to be made when testing on an iOS device. For more details about this see Section 3.2.3.

The database used with the game was created and manipulated with MySQL through PHP scripts, which in turn were communicating with the Unity application. Because of this generic setup, the network solution was platform independent and no database related problems occurred in the thesis.

4.3 Character Design

To create a character without having a background of art design, character sculpting and character modelling is a tough task. The process involves many different steps and many different softwares. The pipelines and workflows used to create a game character is very different depending on which company that develops the character. However, the pipeline and different steps used in this thesis could be viewed upon as a guideline. This thesis found that creating a character from scratch without the appropriate background, competence and resources was extremely time-consuming. Therefore, it is instead recommended for indie developers who wishes to create their own characters to look at other alternatives. If outsourcing this specific task to another person or company is out of the budget, one can instead turn the attention to free character creation tools such as MakeHuman and Autodesk's Project Pinnocchio. This alternative comes with other limitations such as limitations regarding the appearance, clothing and the structure of the character. If, however, an own pipeline is to be used, this thesis concludes that using completely free softwares such as Blender throughout the whole pipeline is feasible and very economical.

A problem with Blender's Rigify tool was that the generated skeleton was not fully compatible with Mecanim. 3ds Max's Biped tool was more compatible with Mecanim, however the weighting system is not as good as it is with Blenders' Rigify.

4.4 Future Work

Since the game is not completely implemented, there is some work to do before the game is ready to be released. First and foremost, only five of the nine mini games are in a completed or close to be completed state. The other four games might need three or four weeks of implementation before being considered as completed. Also, some user related functions also need to be implemented before an eventual release. The most important one being a password reset function.

Before releasing the game, more usability tests needs to be carried out on the game in order to obtain more user input and detect more bugs. Tests focusing on a wider range of devices and screen resolutions must also be performed. Finally, if the game is to be released, content that has been produced using trial licences of different software must be replaced by other content. This includes the animations recorded with both the Kinect devices and the Qualisys system. Instead, there are free animation files available that need to be used instead.

4.5 Final Conclusions

This thesis has presented the development of a cross-platform, multiplayer, 3D game application. The thesis has not focused on the programming aspects of the game. Instead, issues regarding cross-platform development (in general and especially for Unity), animation techniques and character design workflow has been the main

focus. These aspects has also been examined from an economical point of view in order to provide recommendations and guidelines for indie game developers.

It is in this thesis concluded that Unity is a powerful and free game engine which lets the user develop cross-platform games with very few platform specific differences. A difference however is described regarding the implementation of user interfaces. In the thesis, a generic user interface was developed inside Unity using the free prime[31]'s UIToolkit plug-in. In order to comply with interface guidelines provided by the companies behind both platforms, some dynamic adjustments are being made dynamically on the generic interface depending on the running platform. Unity also provided good support for connecting to a cross-platform independent multiplayer network, using an external web server and database, which was also implemented in this thesis.

The thesis investigated and compared three different motion capture systems and techniques for use in recoding of character animations. It was concluded that using a setup consisting of two Microsoft Xbox 360 Kinect devices was a cheap and good enough solution for use in a environment not that dependant of animation quality, such as a mobile platform.

As for the character design, this thesis explains the pipeline and workflow used to develop a fully rigged human character that can be used in a 3D game. However, due to complexity and time needed for an individual to create a good character without having a character design background, some alternatives to creating a character from scratch is described. This thesis concludes that, if an indie development team does not have experience or knowledge regarding character creation, it is recommended if deemed suitable to use automatically created characters. For instance with the Project Pinnocchio software.

Bibliography

[1] John Koetsier. Windows phone jumps to third in global smartphone market share — and could be second faster than you think. http://venturebeat.com/2013/05/16/windows-phone-jumps-to-third-in-global-smartphone-market-share-and-could-be- May 2013.

[2] Apple Inc. ios developer program. https://developer.apple.com/programs/ios/, May 2013.

[3] Apple Inc. ios ui element usage guidelines. http://developer.apple.com/library/ios/#documentation/userexperience/conceptual/mobilehig/UIElementGuidelines/UIElementGuidelines.html, May 2013.

[4] Google. Android design patterns. http://developer.android.com/design/patterns/pure-android.html, May 2013.

[5] A. Thorn. Game Engine Design and Implementation. Foundations of game development. Jones & Bartlett Learning, 2011.

[6] Epic Games. Udk licensing faq. http://www.unrealengine.com/udk/licensing/licensing-faqs/, May 2013.

[7] Epic Games. Game developer licensing for unreal engine 3. http://www.unrealengine.com/en/licensing/, May 2013.

[8] Epic Games. Unreal engine game platforms. http://www.unrealengine.com/en/platforms/, May 2013.

[9] Epic Games. Udk commercial licence terms. http://www.unrealengine.com/udk/licensing/commercial_license_terms/, May 2013.

[10] Unity Technologies. Unity - store. https://store.unity3d.com/products, May 2013.

[11] Unity Technologies. Unity software license agreement. http://unity3d.com/company/legal/eula, May 2013.

[12] Unity Technologies. Frequently asked questions. http://unity3d.com/unity/faq, May 2013.

[13] Unity Technologies. Unity for mobile. http://unity3d.com/unity/multiplatform/, May 2013.

[14] Unity Technologies. Unity for console games. http://unity3d.com/unity/multiplatform/consoles, May 2013.

[15] Unity Technologies. Unity collaboration. http://unity3d.com/unity/collaboration/, May 2013.

[16] Stonetrip. Shiva editor. http://www.stonetrip.com/shiva-editor.html, May 2013.

[17] Gamasutra. Mobile game developer survey leans heavily toward ios, unity. http://www.gamasutra.com/view/news/169846/Mobile_game_developer_survey_leans_heavily_toward_iOS_Unity.php, May 2013.

[18] Unity Technologies. Gamelist. http://unity3d.com/gallery/made-with-unity/game-list, May 2013.

[19] Unity Technologies. Unity scripting. http://unity3d.com/unity/workflow/scripting, May 2013.

[20] Inc Marketwire. Unity 4.0 launches. http://www.marketwire.com/press-release/unity-40-launches-1726144.htm, May 2013.

[21] Unity Technologies. Unity animation. http://unity3d.com/unity/animation/, May 2013.

[22] Unity Technologies. Animator component and animator controller. http://docs.unity3d.com/Documentation/Manual/Animator.html, May 2013.

[23] Unity Technologies. Mecanim animation system. docs.unity3d.com/Documentation/Manual/MecanimAnimationSystem.html, May 2013.

[24] Unity Technologies. Retargeting of humanoid animations. http://docs.unity3d.com/Documentation/Manual/Retargeting.html, May 2013.

[25] Unity Technologies. Creating the avatar. http://docs.unity3d.com/Documentation/Manual/CreatingtheAvatar.html, May 2013.

[26] Unity Technologies. Generic animations in mecanim. http://docs.unity3d.com/Documentation/Manual/GenericAnimations.html, May 2013.

[27] Unity Technologies. Workning with humanoid animations. http://docs.unity3d.com/Documentation/Manual/AvatarCreationandSetup.html, May 2013.

[28] Unity Technologies. Configuring the avatar. http://docs.unity3d.com/Documentation/Manual/ConfiguringtheAvatar.html, May 2013.

[29] Unity Technologies. Gui scripting guide. http://docs.unity3d.com/Documentation/Components/GUIScriptingGuide.html, May 2013.

[30] Unity Technologies. Unity ios basics. http://docs.unity3d.com/Documentation/Manual/iphone-basic.html, May 2013.

[31] Unity Technologies. Gui basics. http://docs.unity3d.com/Documentation/Components/gui-Basics.html, May 2013.

[32] Unity Technologies. Gui. http://docs.unity3d.com/Documentation/ScriptReference/GUI.html, May 2013.

[33] Unity Technologies. Ongui(). http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.OnGUI.html, May 2013.

[34] Above and Beyond Software. Ez gui. http://www.anbsoft.com/middleware/ezgui/, May 2013.

[35] Tasharen Entertainment. Ngui: Next-gen ui. http://u3d.as/content/tasharen-entertainment/ngui-next-gen-ui/2vh, May 2013.

[36] prime[31]. Uitoolkit. https://github.com/prime31/UIToolkit#readme, May 2013.

[37] Andreas Löw. Increasing your game's performance. http://www.codeandweb.com/what-is-a-sprite-sheet-performance, May 2013.

[38] Andreas Löw. What is a sprite sheet? http://www.codeandweb.com/what-is-a-sprite-sheet, May 2013.

[39] Inc Autodesk and Autodesk. 3ds Max 9 Essentials: Autodesk Media and Entertainment Courseware. Autodesk media and entertainment courseware. Taylor & Francis, 2007.

[40] M. Kitagawa and B. Windsor. MoCap for artists: workflow and techniques for motion capture. Focal Press. Elsevier/Focal Press, 2008.

[41] Organic Motion. What is motion capture? http://www.organicmotion.com/products/openstage/motion-capture, May 2013.

[42] Computable Minds. Kinect: How works its 3d body tracking. http://www.computableminds.com/post/kinect/how-works/characteristics/microsoft/xbox-360/3d-body-tracking, May 2013.

[43] iPi Soft LLC. Software. http://ipisoft.com/store/, May 2013.

[44] iPi Soft LLC. End-user licence agreement. http://ipisoft.com/EULA_Studio.html, May 2013.

[45] Inc Cadavid Concepts. Purchase. http://nuicapture.com/purchase/, May 2013.

[46] Inc Cadavid Concepts. Trial. hhttp://nuicapture.com/download-trial/download-nuicapture-animate-stable-version/, May 2013.

[47] Humlab. General knowledge base for using the qualisys 240hz system for motion tracking. http://wiki.humlab.lu.se/dokuwiki/doku.php?id=public:motion_tracking#appendixtested_setups_and_lessons_learned, May 2013.

[48] Zero Point Software. Interstellar marines. http://www.interstellarmarines.com/, May 2013.

[49] Blender. Blender. http://www.blender.org/, May 2013.

[50] Autodesk. Autodesk 3ds max 2014 purchase. http://store.autodesk.eu/store/adsk/en_IE/pd/productID.269713600?mktvar004=ilt_wwm_emea_ie_nc___3dsmax2014___, May 2013.

[51] Autodesk. Autodesk 3ds max overview. http://www.autodesk.com/products/autodesk-3ds-max/overview, May 2013.

[52] Pixologic. Purchase zbrush. http://store.pixologic.com/ZBrush-4R5-Single-User-License/, May 2013.

[53] Pixologic. Zbrush. http://pixologic.com/zbrush/, May 2013.

[54] Pixologic. Sculptris. http://pixologic.com/sculptris/, May 2013.

[55] Autodesk. Autodesk mudbox overview. http://www.autodesk.com/products/mudbox/overview, May 2013.

[56] Autodesk. Autodesk store. http://store.autodesk.eu/DRHM/store, May 2013.

[57] T. Mullen. Mastering Blender. Wiley, 2012.

[58] Zero Point Software. Character modeling pipeline. http://www.interstellarmarines.com/articles/development/character-modeling-pipeline/, May 2013.

[59] MakeHuman. Makehuman. http://www.makehuman.org/, May 2013.

[60] Autodesk. Project pinocchio. http://projectpinocchio.autodesk.com/, May 2013.

[61] Daz 3D. Daz studio. http://www.daz3d.com/products/daz-studio/, May 2013.

[62] Smith Micro Software. Poser. http://poser.smithmicro.com/, May 2013.

[63] hbwares. Wordfeud. http://wordfeud.com/, May 2013.

[64] Unity Technologies. Networked multiplayer. http://docs.unity3d.com/Documentation/Manual/NetworkedMultiplayer.html, May 2013.

[65] Jenkins Software LLC. Raknet 4. http://www.jenkinssoftware.com/features.html, May 2013.

[66] Exit Games. Photon server. http://www.exitgames.com/Photon/Unity, May 2013.

[67] Yahoo! Player.io. http://playerio.com/, May 2013.

[68] Binero AB. Binero. http://www.binero.se/webbhotell, May 2013.

[69] Oracle Corporation. Mysql. http://www.mysql.com/, May 2013.

[70] The phpMyAdmin Project. phpmyadmin. http://www.phpmyadmin.net/home_page/index.php, May 2013.

[71] Dr Dan Saranga. Athletic male. http://www.the-blueprints.com/blueprints/humans/humans/32708/view, May 2013.

[72] Carnegie Mellon University. Marker placement guide. http://mocap.cs.cmu.edu/markerPlacementGuide.pdf, May 2013.