Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Hasbi Adenan

A development environment for interactive installations combining gesture recognition and human character animation

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Hasbi Adenan

A development environment for interactive installations combining gesture recognition and human character animation

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Birgit Wendholt
Zweitgutachter : Prof. Dr. Philipp Jenke

Hasbi Adenan

Thema der Arbeit

Eine Entwicklungsumgebung für interaktive Installationen kombiniert mit Gestenerkennung und menschlicher Charakter-Animation

Stichworte

Computer Animation, Computergrafiken, interaktive Installation, Skelet Animation, Skinning

Kurzzusammenfassung

Der Zweck für diese Arbeit ist eine Umgebung für interaktive Applikationen zu entwickeln. Bei Benutzung der entwickelten Umgebung kann man einen menschlichen Charakter aus Bildern modellieren und mit ihm eine Interaktive Applikation zu entwickeln. Die entwickelte Applikation hat zwei Funktionalitäten: Imitation von Menschen und Interaktion durch Gesten-Erkennung und Charakter Animation. Mit dieser Entwicklungsumgebung braucht der Entwickler keine Programmiererfahrung.

Hasbi Adenan

Title of the paper

A development environment for interactive installations combining gesture recognition and human character animation

Keywords

Computer Animation, Computer Graphic, Interactive Installation, Skeletal Animation, Skinning

Abstract

The aim of this thesis is to develop a new environment for creating interactive installation using character animation. The developer of the environment should be able to create virtual characters from image input that could imitate participant movement and interact with participants. The interactive application with those functionalities could be developed without having any prior programming knowledge.

Contents

1 INTRODUCTION 6

1.1 GOALS 7
1.2 OVERVIEW 8

2 RELATED WORK 9

2.1 IMITATION 9
2.1.1 Talk to the Virtual Hands 9
2.1.2 Puppet Parade 10
2.2 INTERACTION 12
2.2.1 Virtual Superheroes 12
2.2.2 IMoSA 13
2.3 CONCLUSION 14

3 THEORETICAL BACKGROUND 15

3.1 COMPUTER ANIMATION 15
3.1.1 Key-frame System and Interpolation Technique 16
Interpolation 18
3.2 MODELING AND ANIMATING HUMAN CHARACTERS 20
3.2.1 Motion Control over Body 20
Hierarchical Modeling 20
Skeleton Definition 26
3.2.2 Kinematic Methods for Skeleton Animation 27
Forward Kinematic 28
Inverse Kinematics 28
3.2.3 Creation of Human Shapes 31
Polygonal Representation 33
Texturing 34
Skinning 35
3.3 MOTION CAPTURE 37
3.4 CONCLUSION 40

4 METHODOLOGY 42

4.1 FUNCTIONAL REQUIREMENTS 42
Character Modeling 42
Imitation 42

Interaction ... 43
4.2 NON-FUNCTIONAL REQUIREMENTS ... 43
Performance ... 43
Usability ... 43
4.3 SYSTEM DESIGN AND SOLUTIONS ... 43
Skeleton Component ... 43
Shape Component ... 44
Motion Capture Component ... 44
Gesture Component ... 44
Animation Component ... 45
Conclusion ... 45
4.4 TOOL AND LIBRARIES ... 47
4.4.1 Unity ... 47
Workflow and Primary Objects in Unity (GameObject, Component, and Assets) ... 48
GameObject and Component ... 49
Assets ... 49
Parenting system ... 50
Scripting in Unity ... 50
Creating Mesh in Unity ... 55
Animation System ... 57
Conclusion ... 59
4.4.2 Libraries ... 60
4.4.3 Conclusion ... 61
4.5 DESIGN ... 62
4.5.1 Component Overview ... 62
4.5.2 System Implementation ... 65
CharacterController ... 65
PolygonController ... 69
Gesture ... 73
4.5.3 Implementation Environment ... 81
4.6 CONCLUSION ... 82

5 CONCLUSION ... 83

6 APPENDIX ... 85

6.1 CONFIGURING UNITY ... 85
6.2 CHARACTER MODELING ... 86
6.2.1 Creating Skeleton ... 86
6.2.2 Creating Shape ... 87
6.3 IMITATION ... 91
6.4 INTERACTIVE ... 92
6.4.1 Creating template gesture ... 92
6.4.2 Creating animation ... 96
6.4.3 Attach gesture to animation ... 98
6.4.4 Gesture recognition ... 99

BIBLIOGRAPHY ... 100

TABLE OF FIGURES ... 105

1 Introduction

Interactive art installation has become one of the most interesting new forms of art nowadays. This art form emphasizes the involvement of the audience to achieve its purpose. Communications between human and machine are not limited only to keyboard input or mouse click, but also include natural human gestures and voices. These natural ways to interact with technologies make it easier for people to acquaint themselves with the interface, and operate machines and devices. This kind of interface is also known as Natural User Interface (NUI) where natural human movements (such as gestures) are used for input. Camera and computer vision techniques are used to recognize and read gestures of people [Nobl09], and by using the combination of movements of body parts, they enable the user to experience a greater degree of freedom in terms of user experience and object manipulation. For example, these include using two hands to stretch an object or make it bigger in the virtual world. Such movements are easier and more natural than when using a mouse or a keyboard, hence easy to learn and enhance user experience.

Even though a system can perceive human behavior through NUI, it should be able to react intelligently and give feedback to the audience accordingly. No matter how much a system can "read," it would be nothing if the output is not satisfying or sufficient. The output media in interactive installation should give intelligent reactions to the audience so that he/she knows what kinds of action provoke a response [Krue77]. One medium output that can be used for creating interesting reaction based on participant is by using a virtual character. Using a virtual character, an interactive installation can imitates the movement of participant or create an interactive scenario between participant and character.

Interaction with virtual characters with natural user interface has gained increasing attention over the past few years. With rapid developments in hardware and software technology, such interactive art is on high demand for commercial and research purposes.

Unfortunately, developing an interactive application from scratch usually requires understanding of specific and complex systems. Hence, there is a need for a general tool for developing this kind of interactive art that lightens the burden of developing installation for artists and developers without giving up the freedom of creativity.

1.1 Goals

This thesis seeks to introduce a new environment for developing interactive installations using character animation, bringing together the NUI aspect and interactive animation. The developed environment should have the following requirements:

1. Character modeling based on images
2. Imitation of participants' movements with virtual characters
3. Interactive application with gesture recognition and animation of virtual characters

Character modeling requires systems to be able to model and create virtual characters for interactive application. This functionality requires a developer to create virtual characters based on character textures from the developer. The character created from this system will move into the 2D world.

Imitation of participant’s movement requires a system’s capabilities to read and recognize relevant anatomical features of a participant and to transform it into movement of virtual character. The animated character is in human shape and should move according to participant’s movement. However the detail aspects of human shape like finger or face expression are not considered in this thesis.

Interactive virtual character involves recognizing input gestures from participants and creating animation sequences in response. When developing an application with this environment, the developer could define "template" gestures for recognizing input, create character animation sequences as output response, and finally connect between “template” gestures and animation sequences. Later in the finished application, this information will be used for creating interactive scenarios between the participant and the virtual character, where the gesture from the participant will be “recognized” and triggers the animation output.

The solution provided within this thesis will utilize existing, suitable tools and algorithms in order to implement the requirements.

1.2 Overview

This thesis will be divided into five chapters. After the introduction, the second chapter will briefly discuss the projects that have similar functionality to the developed system. This chapter will be further divided into two sections—imitation and interaction.

In the third chapter, this thesis will examine the theoretical background for the developed environment. This chapter will focus on three aspects—the foundation of computer animation, character modeling and motion capture systems.

The fourth chapter of this thesis concentrates on the implementation of the developed environment. In this chapter the functional and non-functional requirements will be described and the system solutions to the requirements will be proposed. This chapter will then go on to examine the tools and external libraries for creating the solutions. Combining external libraries and selected tools, the desired environment will be created.

Finally, in the chapter Conclusion, the evaluation of the overall system will be presented. This chapter will also discuss the possibilities of extending the functionality of the developed system.

2 Related Work

This chapter describes previous works related to aspects and requirements mentioned in the introduction. The following projects in the first subchapter are installations that use imitation functionality. Using information from motion capture data, projects such as *Talk to the Virtual Hands* and *Puppet Parade* create character imitation of human body parts.

The second subchapter mainly focuses on projects with interaction. Projects *Virtual Superheroes* and *IMoSa* present examples of interaction with the system with the help of motion capture. This chapter will be concluded with a discussion of the advantages and disadvantages of each system and how some aspects of the related studies will be used in this thesis.

2.1 Imitation

2.1.1 Talk to the Virtual Hands

The focus of the experiment from Dodds et al. [DoMB11] is to study non-verbal communication using virtual reality (VR) as a medium. In this experiment, participants wore a head-mounted display and were given a self-animated avatar that follows their movements. The participants would see the virtual world and their avatar from a first/third-person perspective through the head-mounted display.

In one of the experiments, the participants play a describer-guesser game[1]. This task is divided into two sessions. The first session is where the participants can play the game with self-animated avatars (avatars follows the participants movement) and verbal communication. The other session is to play the game where the avatars are static and the

[1] One have to describes a word without saying the word itself and the other have to guess the word

participants can only communicate verbally. The experiment compares the participant's performances in both sessions.

Figure 2.1 virtual world avatar follows gesture from participants (left) is how the participant saw each other in VR world

Technical environment:

Body movements of each participant would be tracked by an optical tracking system and mapped onto the avatar. For tracking the movements, participants wore six markers that would be used for tracking their hands, feet, body, and head (Figure 2.1). The joint positions that are not tracked (such as elbow) are calculated through *inverse-kinematics* algorithm. A software package called *Virtools 4.1* was used for creating a VR world, while 16 *Vicon MX13* cameras were used to track the markers on the participants' bodies.

The result from this experiment is that participants perform better when the avatar is self-animated (imitate). In the VR world, the participants could communicate better with body gestures available.

2.1.2 Puppet Parade

Puppet parade [Desi11] is an interactive installation from Watson et al of *Design I/O*. This installation imitates the hand and arm movements for creating bird puppets on a screen. This installation premiered at the 2011 *Cinekid* festival in Amsterdam.

This installation projects large bird puppets on a wall that imitates the hand and arm movement of participant (puppeteers) (figure 2.2 left). The puppeteers can control the birds'

neck and mouth movement using arm and finger movement and interact with the audience (children).

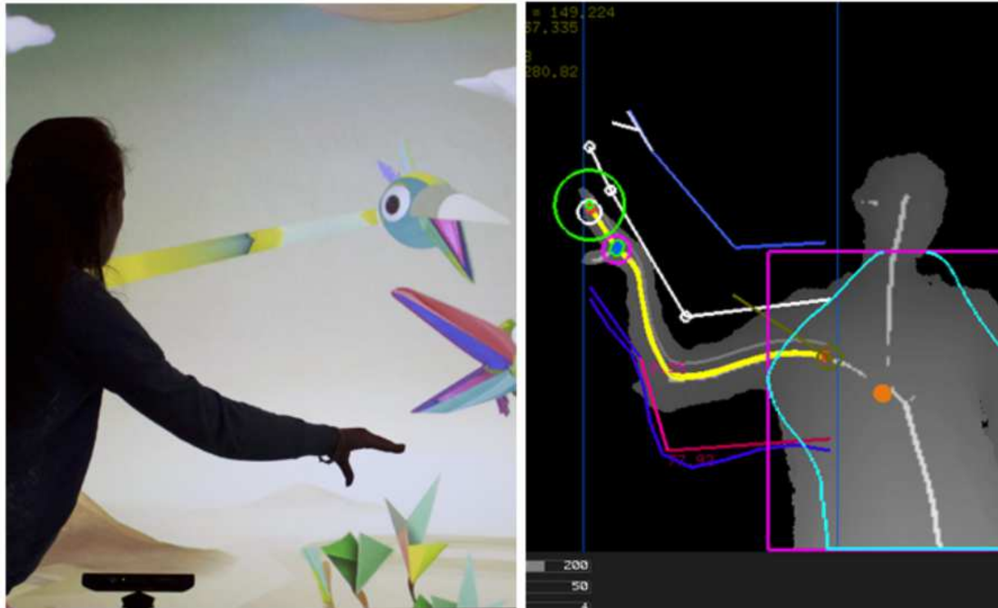

Figure 2.2 (left) Puppet Parade in runtime. (Right) skeleton point from thumb and the forefinger will be created for controlling bird-avatar’s beak.

Technical environment:

In this installation, a Kinect camera will be used for getting the silhouette of the hand while the system will track the skeleton point from the depth image of Kinect. The system will calculate the arm skeleton points from this information (including thumb and forefinger for the puppets mouth movement). These skeleton points movement will be mapped onto the bird model for imitating the hand movements. The system also tracks the movement in 3D so the puppet can look towards /away the audience when the arm rotates towards/away from the Kinect.

The system is supported by the Kinect library and driver, but the imitation functionality is limited to the arm and finger imitation. The system uses the *Xbox Kinect* sensor and is developed in *openFrameworks* with *ofxKinect* (as its Kinect wrapper) and *libfreenect* (as Kinect driver).

2.2 Interaction

2.2.1 Virtual Superheroes

Virtual Superheroes is a project by Rosenberg et al. [RoBB13] of Stanford University. The aim of this project is to study participants' behavior after a flying experience in the VR world. Each participant interacts with their movements in the virtual world by moving their hands.

Technical environment:

In this experiment, participants will wear a head-mounted display, and three optical markers on the head and both hands. For detecting the markers, eight optical infrared cameras are used. The system will detect the position based on those three markers and respond to the movement in the virtual world. When the participants raise their hands higher than their heads, they will fly higher in the virtual city.

Figure 2.3 (left) virtual world which participant see through head mounted display. (Right) participant raises their arm for flying simulation

2.2.2 IMoSA

IMoSa (*Imitation Mechanisms and Motor Cognition for Social Embodied Agents*) [SaKo11] is a project by the Social Cognitive Systems Group of Bielefeld University. This project focuses on developing a motor cognition mechanism and to apply it to the behavior of a robot avatar called VINCE.

Technical environment:

The system works as follows: participants' hand movements will be tracked using a *time-of-flight* camera and a tracking software. After tracking the participants' hand movements, the system starts recognizing the gesture and the robot avatar reacts in response to the recognized movements. If the robot avatar recognizes a gesture, it will tell the participant the name of the gesture and then recreates the participant’s movement.

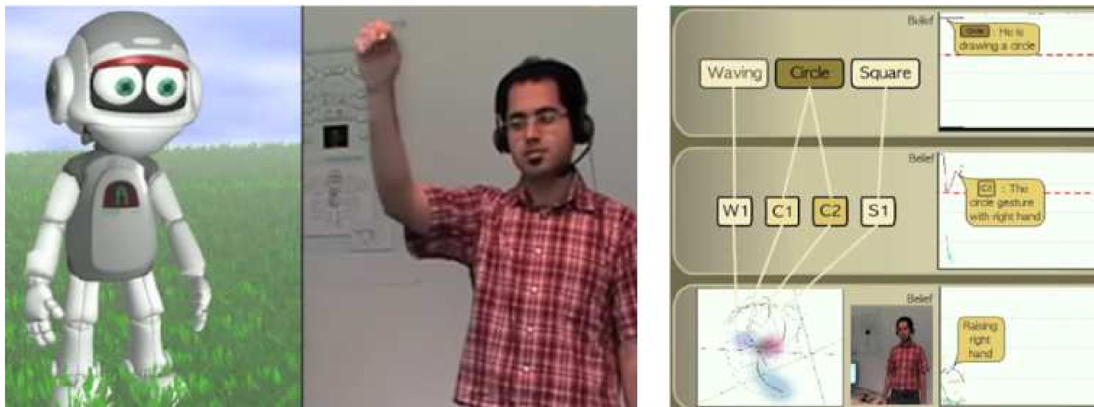

Figure 2.4 (left) participant tries to draw a circle hand-movement in IMoSa (right) the process of recognizing the hand movement

The process of recognizing gestures in this system occurs within a fixed time. Within this time, the participant will move their hand(s) in a loop and the system will try to match the participant’s hand-movement with template gestures. Based on the frequency of recognized template gestures in a fixed time, the system will decide which gesture does the participant’s movement belongs. If the gesture is not recognizable, the system provides possibilities to save it as a new template gesture.

2.3 Conclusion

There are many projects and interactive installations involving character imitation and interactive functionality. This chapter shown that for creating such interactive installation, a motion capture system is needed. The first part of this chapter shows an example of imitation functionality with marker-based [DoMB11] and marker-less motion capture [Desi11]. While marker-based system requires participant to wear multiple markers on their bodies, the marker-less motion capture doesn't require any markers. Both of the systems show basic requirements for creating imitation functionality: the system should be able to track the joint position from a participant and to map it onto the avatar for creating imitation in real time. The avatar will follows the participant movement based on the avatar skeleton.

The experience from Puppet Parade shows that interactive installations using marker-less motion capture is more flexible and suitable for interactive installation. Therefore, this thesis will use a marker-less motion capture system.

Section 2.2 focusses on projects with interaction functionality. The first project [RoBB13] shows an example of *physical interaction* in a virtual environment (interaction without interpreting gestures). As opposed to this, the project *IMoSa* [SaKo11] proposes a solution to interactions using gesture-recognizer algorithm. A gesture-based interaction is more suitable for this thesis since it gives the freedom of interpreting human behavior and reacting with appropriate response of an avatar

Creating character animation with imitation and interaction functionality will not only require motion capture system but also an animation system and a character modelling functionalities. The next chapter serves as the theoretical background for these functionalities.

3 Theoretical Background

This section will discuss the background theory that will be used in this thesis. The first two sections focus on character animation and character modelling: Section 3.1 explains how computer animation works and which functionality computer animation does have in relation to traditional animation. Section 3.2 describes the kinematic model of human characters followed by the basic of human character representation in computer animation. Section 3.3 describes techniques of motion capture that will be used for imitation and interaction requirement. The conclusion will show how the theory presented in this thesis should be applied in implementation.

3.1 Computer Animation

Animation is rapid display of sequential images that create an illusion of movement. A single image that is being used in animation is called *frame*. A traditional animation frame was created from hand-drawn images in the past. There are two ways to animate following traditional animation: *straight ahead* and *pose-to-pose*.

The term straight ahead means drawing frame by frame from the beginning to end of a sequence while in the pose-to-pose method, the animator decides the most important drawings in one movement sequence and draws them first then gives these main drawings (also known as *key frame*) to an assistant who will create intermediate drawings between these images (for example, see Figure 3.1). The process of filling in frames between the key frames is called *inbetweening*.

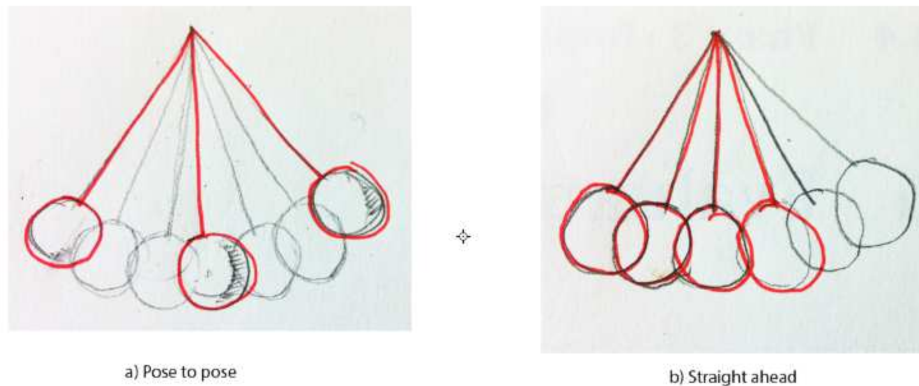

Figure 3.1: (a) In the pose-to-pose method the important drawings are first to be drawn (in red line) and intermediate drawings in inbetweening (in gray lines); (b) in the straight-ahead method the frames are drawn one by one from the start to end of a sequence.

3.1.1 Key-frame System and Interpolation Technique

The idea of creating movement between poses, as in the pose-to-pose method of hand-drawn animation, is also used in computer animation [Lass87]. Unlike traditional animation, computer animation does not use drawings or shapes to determine a key frame but uses values known as *key values*. Inbetweening happens when these values are interpolated between two key frames (also known as *interpolation*[2]).

Before key values and how interpolation works in computer animation is discussed, it is necessary to understand the basic representation of animation. Moving objects seen in computer animation are basically objects that are transformed from their original defining *space* (*object space*) into other intermediate spaces until they are mapped onto the monitor. Spaces are the environments of objects, and they are represented in a coordinate system. Object data is the representation of an object that is being animated and usually is defined by data points. In computer graphics these data points are referred to as *vertices* and used to define an object of animation.

[2] The terms interpolation in computer animation is basically inbetweening in traditional animation.

Object space is a coordinate system that is unique for each object. The data of an object is usually centered on the *origin,* which is also called *origin point* (0, 0, 0). The object that is defined by its vertices is transformed from object space through translations, rotations, and scales into *world space*. World space is the space in which objects are assembled in order to create an environment to be viewed by the *observer* (Figure 3.2) [Pare08].

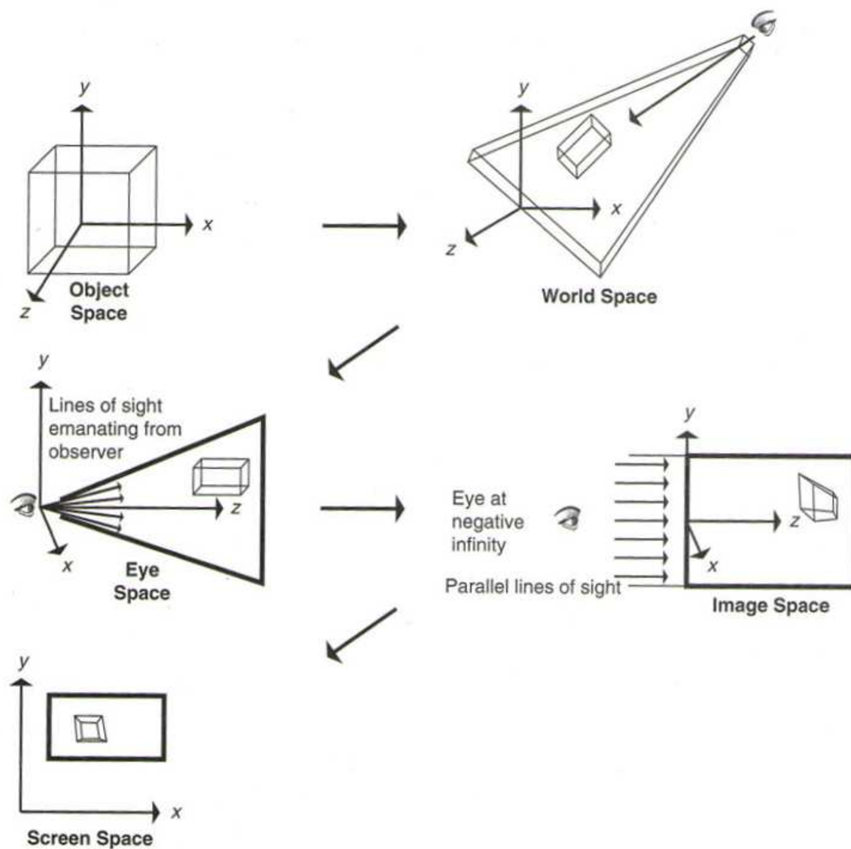

Figure 3.2: Transforming objects between spaces [Pare08] p. 45.

Eye space has a coordinate system like that of the world space, except it considers the observer's position as the origin. Objects in world space will be transformed so that their positions become relative to the observer. The next transformation is from eye space to

image space. This transformation can create either *perspective projection* or *orthographic projection*. Orthographic projection keeps the object at its original size, but perspective projection creates a 3D perspective for the object and could create deformation. Figure 3.2 shows an example of transformation into the perspective projection within eye space. Finally, the object will be clipped and mapped on the screen in *screen space* [Pare08].

Interpolation

An animation following the previous process of bringing the object from object space to screen space is created by altering object transformations in one space over time. It can be created by modifying the objects' position or orientation in world space over time, or by modifying the object scale and its display attributes (such as color or transparency) over time. An animation can also be made by changing the observer's orientation and position over time—both of which create object movement[3]. These (transformation) attributes are *values* used in the interpolation function as parameters to create animation, and they are also modified over time.

In key-frame systems, the animator[4] usually has to determine the position, orientation, and scale transformation of an object in the so-called *key values* while the computer interpolates the intermediate frames. The interpolation technique is used to create new values in these intermediate frames.

As an example, we have two points that serve as key frames: A and B. The key frame A is located in position (0, 0) by frame 0, while the key frame B is located in position (10, 0) by frame 5 (shown in Figure 3.2).

---

[3] In reality, the object can have no animation. However, it creates illusion of movement by changing the viewer's movement.

[4] In traditional animation, animators are those people who draw frames. But in computer animation, animators are those artists who use programs to create animation.

Figure 3.3: Example of two key frames with its value (position).

Interpolation technique will be used in this example to create intermediate values for frame 1 to 4. If the desired animation is a movement from *key value* (0, 0) to *key value* (10, 0) along a straight line, then the proper interpolation technique is conducted with the help of a linear interpolation function. When there are two coordinates, as described in (x0, y0) and (x1, y1), then *linear interpolant* is the straight line between the two given point. Figure 3.3 shows the result from interpolation (drawn in black point).

Figure 3.4: Key frames with their interpolated values.

The previous example shows an interpolation by using one parameter (x-coordinate) of the object being interpolated, as the y-coordinate is constant. A linear interpolation function is chosen because of its simplicity; but what if the point has to move along a curved line, or the point has to be accelerated to arrive in frame 5 with the desired designated speed? More parameters would then be involved in the interpolation function since the complexity is higher.

A tool, which will be introduced later, supports the animation of the key frame system where the user can freely change the function and control the animated object's movement.

3.2 Modeling and Animating Human Characters

This section explains how animation and data representation of a human character is created in computer animation. This starts from how to control the motion of a body with hierarchical modeling to abstraction of skeleton by using such structures. Afterward, the kinematic method for animating such a model will be discussed. This section also addresses how the human shape representing the body of a character can be created.

3.2.1 Motion Control over Body

Hierarchical Modeling

While describing object motion, sometimes it is easier to describe an object in relation to another object. For example, it is easier to describe the movement of the moon relative to the earth that rotates around the sun than describing the moon's movement relative to the sun. This model of objects are the types of *hierarchical modeling.*

Hierarchical Modeling defines a tree-like structure that gives location constraints for every child relative to its parent [Pare08]. The data structure of hierarchical modeling consists of *nodes* and *arcs* (Figure 3.5). Objects of animation will be arranged in tree nodes, and transformations from node and arcs will be applied to them.

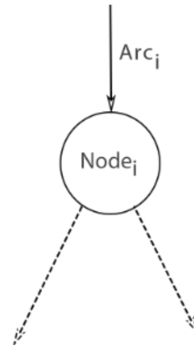

Figure 3.5: Arc and Node, $Node_i$ is leading node of Arc_i [Pare08].

A node of tree structure contains information necessary to define object data. Other than the object data, a node also contains transformations (of the object) that are applied within the local coordinate system (object space). The node transformation N_i is applied so that its point of rotation is at the origin, i.e., (0, 0) or (0, 0, 0) [Pare08]. For example, in Figure 3.6, a rectangle is specified with its bottom left corner at the origin in the local coordinate space. If it has to rotate around the rectangle's center, then a node transformation (N_0) would be needed to move its center to the origin.

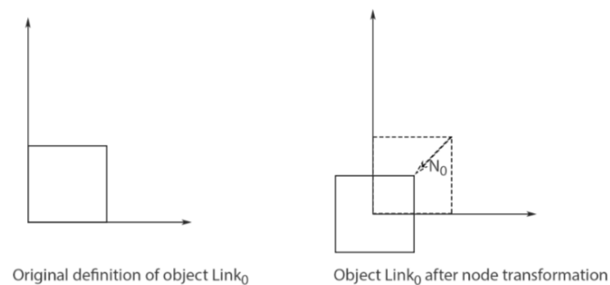

Figure 3.6: Node transformation within object space, making the rotation point in the middle of the rectangle.

Arcs hold two types of transformations. One constant transformation T_i defines rotation and translation of the child node relative to its parent node in a tree: this transformation determines the child's origin position relative to its parents. For example, if one node in a tree contains information about a human forearm, this transformation will be responsible

for placing the forearm object from the origin[5] to the default position at the elbow. Another transformation is variable transformation R_i that is applied in order to create the movement of an object. Variable transformation changes over time to create movement, while constant transformation keeps the objects in constrained position. In human character models the typical variable transformation is joint rotation.

In a tree structure there are three types of transformations: node transformation and arc transformations (constant and variable). Node transformation will be applied to transform the object's coordinate space, while arc transformation (both constant and variable) is applied to transform between the parent's coordinate space and child's coordinate space.

A model that is defined through the tree structure of a hierarchical model is shown in Figure 3.7. In this figure there are three objects arranged in a tree structure: $Link_0, Link_1$ and $Link_2$. $Link_0$ is a root object with constant transformation T_0(translation). An object $Link_1$ is defined as a child of $Link_0$ with its constant transformation T_1(translation) and variable transformation R_1(rotation), while object $Link_2$ is defined as a child of $Link_1$ with its constant transformation T_2(translation). In this example, the objects are physically connected from end to end. In robotic field, the object represented as the physical segment itself is referred to as a *link*. The object sequence that physically connects them is referred to as *linked appendages,* or in short, *linkages* and the connection between the objects through linkages is called *joint*. [Pare08] [Crai89].

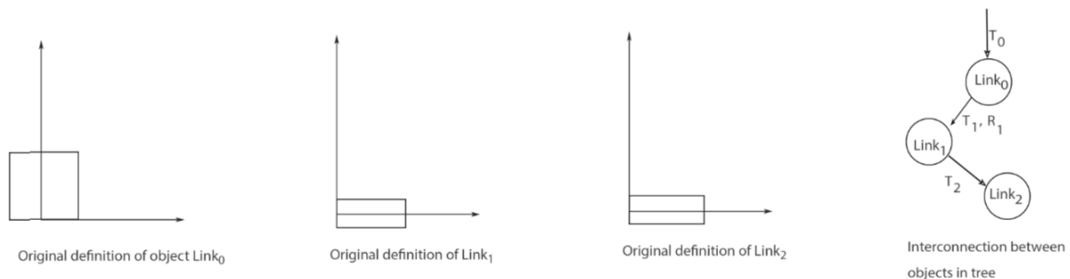

Figure 3.7: Example of the objects' tree structure in its original definition and their arrangement in tree structure.

[5] In the tree of a human body, the origin of the forearm is defined by the local coordinate of its parent, namely the upper arm.

Figure 3.7 shows objects (links) in their original definition. This information is contained in nodes in a tree structure. There is no node transformation in this example because the rotation point is defined (rotation point for $Link_0$ is the middle-bottom of rectangle, and for $Link_1$ and $Link_2$ the rotational point is defined in the middle-left side in the object space). All objects have already been defined in their natural default orientation relative to their parent objects and thus there is no need to add a rotational constant transformation.

Transforming a hierarchical model to world space is similar to performing a depth-first traversal of the tree. Figure 3.8 shows how the objects $Link_0$ and $Link_1$ get transformed into world space.

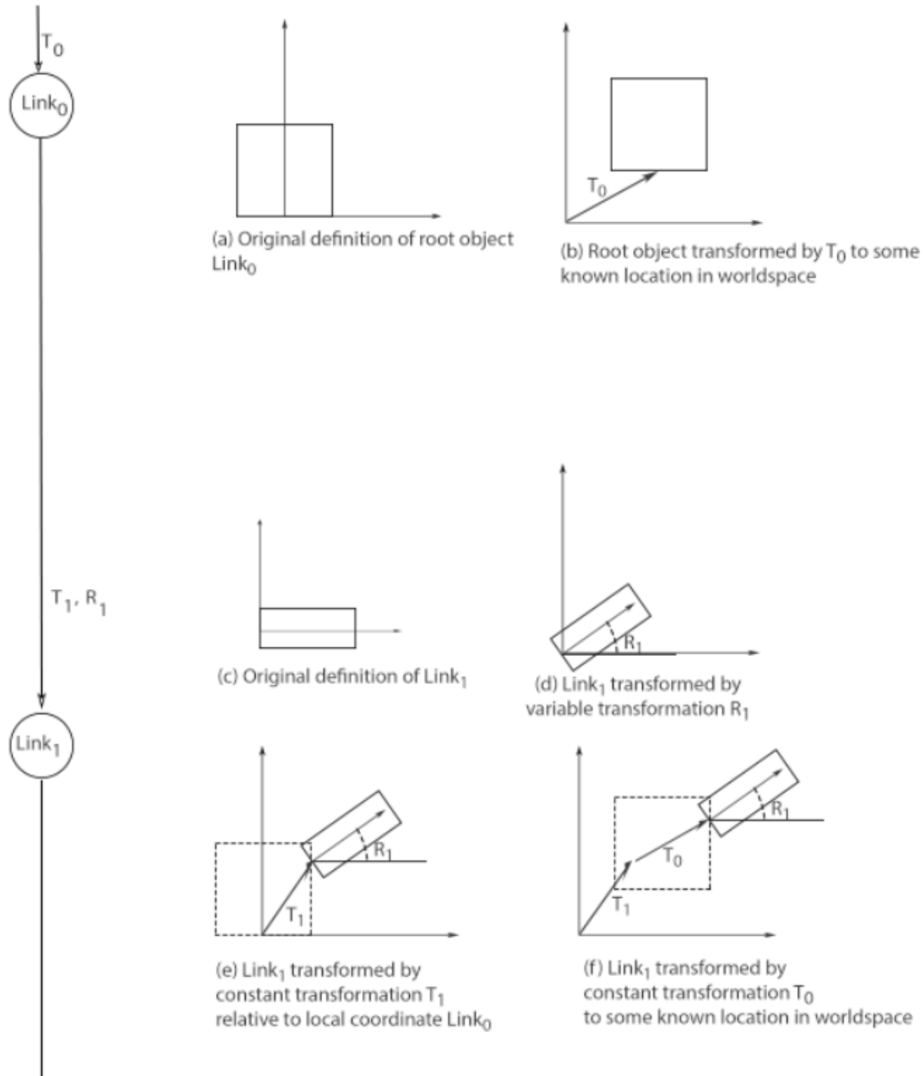

Figure 3.8: Transformation from the object space of the hierachical model into world space. (a) and (b) show normal transformation of root object $Link_0$ from object space to world space. (c)–(f) show the transformation from object $Link_1$ to world space.

The transformation for object $Link_0$ is only T_0. On the other hand, $Link_1$ needs three transformations in order to get into world space (R_1, T_1, T_0), as shown in Figure 3.8 (c)–(f).

Starting from the defining space in (c), the variable transformation R1 is applied[6]. After the variable transformation, the constant transformation T_1 is applied. Since T_1 is the transformation relative to Link0, T_1 brings $Link_1$ to the local coordinate system of $Link_0$. Finally, $Link_1$ is relocated to world space by applying the T_0 transformation. If the object data of $Link_1$ refers to vertices V_1 , then the location of vertices $V_1{}'$ in world space can be attained from:

$$V_1{}' = T_0 T_1 R_1 V_1$$

Object $Link_2$ is transformed similarly by applying all the transformation from the hierarchy to the root (V_2 is the vertex of $Link_2$):

$$V_2{}' = T_0 T_1 R_1 T_2 V_2$$

So the order of applying transformation for every object in $Node_i$ is: node transformation N_i, variable transformation R_i, and then constant transformation T_i of Arc_i [Pare08]. The end product after applying all transformations to all objects can be seen in Figure 3.9.

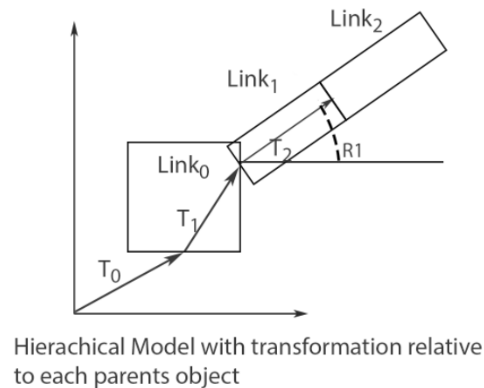

Figure 3.9: hierarchical objects in world space after transformation.

[6] If there is a node transformation on $Link_1$, then node transformation will be applied before variable transformation [Pare08].

From the above example we can conclude that hierarchical models can be useful to simplify the description of object movement. When making animation of a linkage system, as in figure 3.9, we can take the rotational parameter (variable transformation) of $Link_1$ as the interpolation parameter without having to worry whether object $Link_2$ will always be attached to the joint. The constant transformation that is applied from arcs and nodes makes sure that the child movement is restricted to its parent in a tree structure.

Skeleton Definition

As in the previous example of linkage systems (Figure 3.9), hierarchical structures are a common type to construct multi-joint models. Because of the constraints and the connectivity of the model, the animator does not need to make sure that the links of the model stay attached to one another. Such a hierarchical structure is useful for modeling an *articulated figure* (see Figure 3.10). An *articulated figure* is a model that has objects connected from end to end while forming a multi-segment jointed chain. In articulated figures the configurations of the joints are modified to create the movement of links, and the process is known as *articulation*.

Animated characters in computer animation are mostly developed as articulated bodies being represented as skeletons. A *skeleton* is a connected set of links and joints [BaSm79]. A *joint* is the intersection between two links, which is a skeleton point that describes where the limbs (in human figure) can move. The angle between two links is called *joint angle* [Pare08]. In a tree structure of a hierarchical model, the links (or limbs of human) are represented as nodes and the joints are represented as arcs (Figure 3.10).

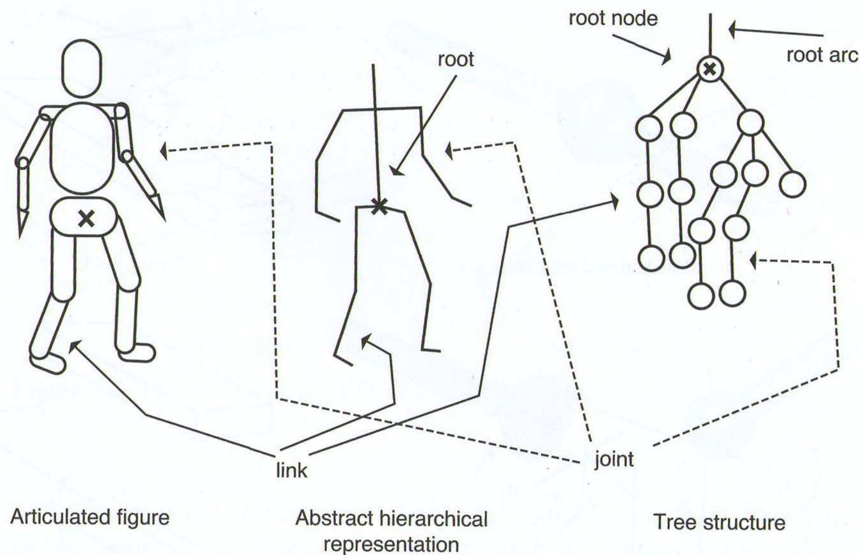

Figure 3.10: Tree structure representing the hierarchical structure of an articulated figure[Pare08] p. 192.

Since in an articulated figure the links should stay attached to one another, which happens to be the only variable transformation that can be used to articulate the figure is rotation of joint angle. Animation or movement can be made by changing the rotation, with the other transformations enforcing constraints. In a tree structure of a human character, node transformations make sure that the rotational point of every limb is exactly at the end of a limb, while constant transformations make sure that the tip of the limb is always at the joint.

3.2.2 Kinematic Methods for Skeleton Animation

In the field of robotics there are mainly two ways to describe the kinematic motion of articulated bodies such as human figures: *forward kinematics* and *inverse kinematics*. These two methods are the way to position an articulated body from one pose to another. For example, in case of positioning an arm of a virtual human, forward kinematics requires the animator to specify rotational joint parameters (shoulder, elbow, and wrist) in order to get the position of objects in world space, while the inverse kinematics method takes the desired hand position in world space and as a result gives the rotational joint.

Forward Kinematic

Forward Kinematics involves finding the end position of the model by rotating the joints. Figure 3.11 uses the linkage system from the previous example (Figure 3.9). The links are connected end to end, and the animator specifies R_1 and R_2 as the joint angle of $Link_1$ and $Link_2$. In a tree structure of hierarchical modeling, these two transformations can be applied directly as a variable transformation of the arcs leading to node $Link_1$ and node $Link_2$.

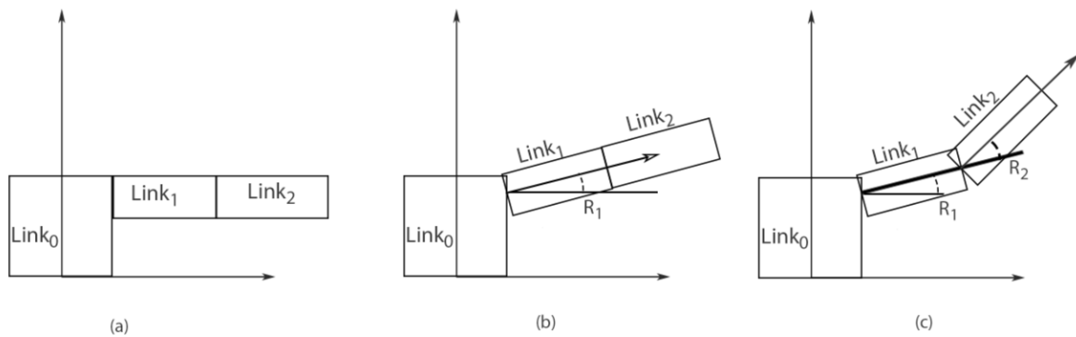

Figure 3.11: Example of forward kinematics specification of joint rotation.

Forward kinematics uses a straightforward method: the position of an object in world space (global coordinate system) is achieved by providing joint angles as input. One of the drawbacks of forward kinematics is that the process of specifying joint angles of articulated bodies can be tedious for animators and involves trial-and-error method to get the exact desired position. Another approach to positioning the articulated bodies is *inverse kinematics*

Inverse Kinematics

Inverse Kinematics permits direct specification of the desired *end-effector* positions and rotations (end point in Figure 3.13). In robotics [Crai89], the *end-effector* is the device at the robotic arm designed to interact with the environment, but in this discussion it is referred to as the position of the free end of a linkage chain. In inverse kinematics, the values of joint angles are calculated to attain the given desired position of the end-effector in world space.

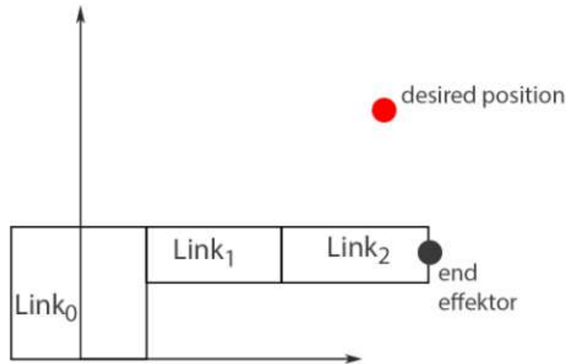

Figure 3.12: Example of linkages system with end-effector.

In Figure 3.12 (a), the desired position of the end-effector is defined by the red dot. In a simple linkage system, as shown in Figure 3.12, the joint configuration (orientation of each joint) could be attained by inspecting the model geometry and then using simple trigonometry. In order to move the linkages to the desired position, the system must first make sure that the desired location can be reached by the linkages. If the desired position has position (x, y) in world space, and objects $Link_1$ and $Link_2$ have length L_1 and L_2, the following precondition must be met to reach the desired position:

$$L_1 - L_2 \leq \sqrt{x^2 + y^2} \leq L_1 + L_2$$

After knowing that the goal can be reached, trigonometry could be used to determine the joint angles by using the *cosine rule:*

$$a^2 = b^2 + c^2 - 2bc \cos a$$

As a result, in Figures 3.14 (a) and (b), R_1,R_2, R_3, and R_4 could be calculated from the links' length and the desired position (L_1, L_2, x, y).

If there are many possible joint configurations for the desired end-effector's position, as in Figures 3.13 (a) and (b), then the situation is referred to as *underconstrained*. When there are no possible joint configurations for the desired position of the end-effector, the system is called *overconstrained* [Pare08].

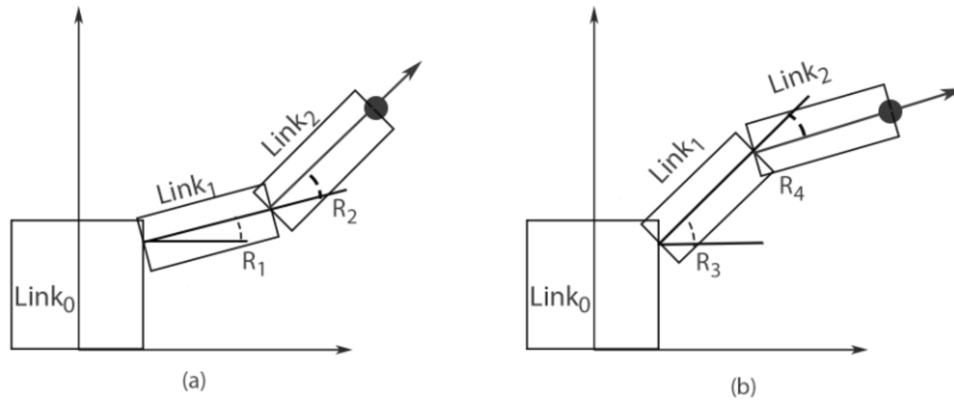

Figure 3.13: Sample sequence of positioning the end-effector of the linkage system by using inverse kinematics. Notice that there are two options, (b) and (c), as possible answers.

In case of a simple linkage system, as in the previous example (Figure 3.12), joint angles can be calculated analytically. However, the inverse kinematics solution to particular positions of complex linkages can become numerous and complicated. Another approach is needed when the number of linkages increases.[7] Once the joint angles are calculated, the figure can be animated by interpolating from the initial state to the final state (where the end-effector is in the desired position) and also by using the calculated joint value as variable transformation of the hierarchical structure (Figure 3.14).

---

[7] The *Jacobian* method, for example, uses the incremental approach that computes the joint angles toward the desired position.

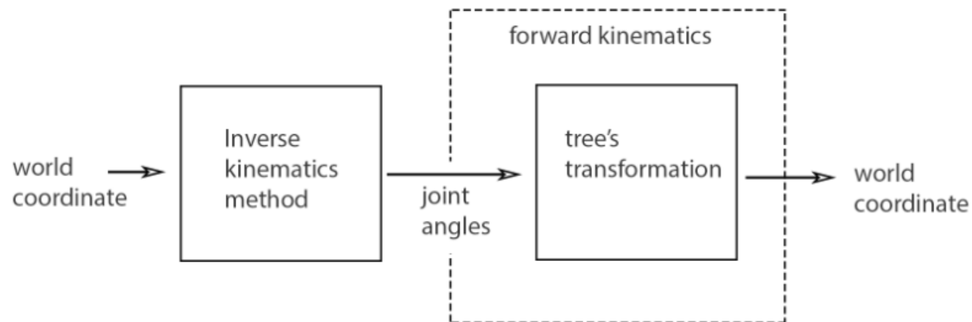

Figure 3.14: Representation of the inverse kinematics and forward kinematics of articulated bodies based on [KuZa06].

3.2.3 Creation of Human Shapes

After discussing the abstraction of a human skeleton model and how to animate it, we will discuss how the shape of human character will be represented. One possibility is to create a set of 2D objects that represent body parts and to attach them to the skeleton of the hierarchical model as links—for example, by using texture for head, foot, arms, among others, that is mapped into each 2D object. The drawbacks of this method: the shape is rigid and fixed; and there is no stretching or deformation of texture to represent flexibility of the human skin and muscle when moving (figure 3.15, first row). Deformation of texture can be achieved when texture is part of a deformable object.

If a texture defines the visual and color of a character (like the texture of human skin should have at least the color of human skin), then a deformable object is used to create a flexible body animation and define the shape of the body. Normally, the texture of the whole body will be mapped to a deformable object that follows the movement of a skeleton—for example, in the second row of figure 3.15, the texture of an arm is mapped onto one deformable object. When the skeleton objects are moved, the defining points of the deformable objects will also be moved; hence, the stretching of the objects creates flexibility for the figure (figure 3.15 uses the vertex as the defining point).

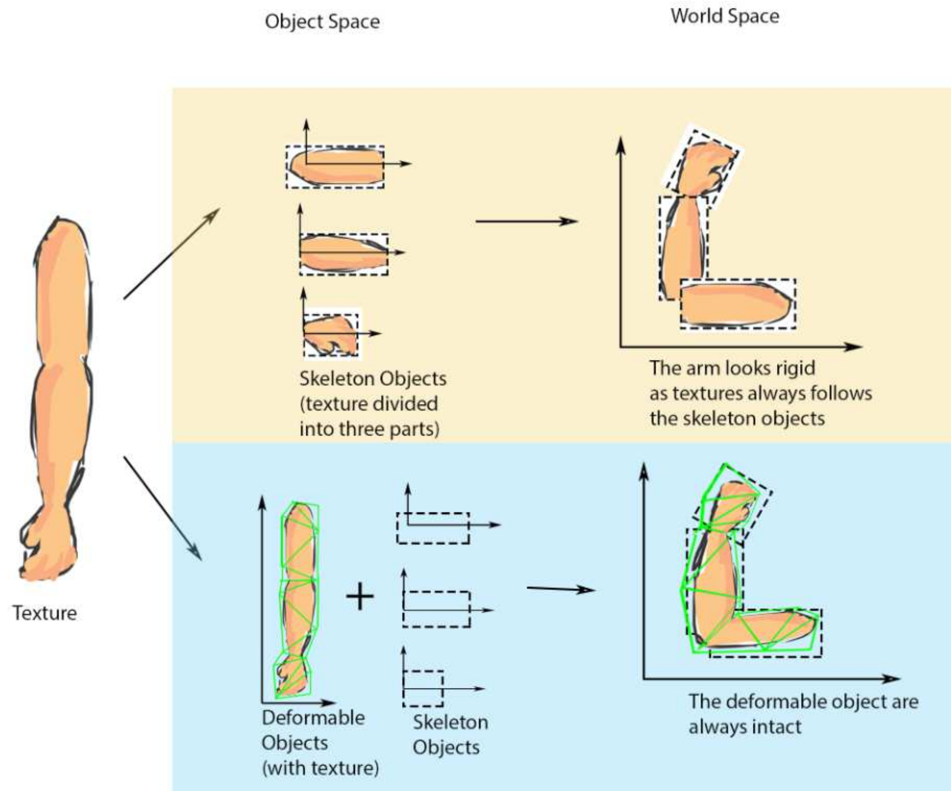

Figure 3.15: Example of using a texture for creating the shape of an arm. The two methods differ on how the texture is mapped onto the object. The first row divides the texture in their corresponding objects and uses it in the hierarchical structure as part of the object, while the second row maps texture into one deformable object and deforms it on the basis of the movement of links and joints.

The deforming of the object depends on the manner in which the deformable object is defined. There are mainly two ways to create a deformable object, and they are also the main modeling techniques used for creating a 3D model: polygons and patches [Pare08].[8] The difference between polygons and patches are basically in the defining method to create the object: polygons use points (vertices), straight lines and surfaces, while patches use curves

[8] There are several other methods for representing the virtual human figure, but they are not used much because of either lack of modeling tools or the method's complexity [Pare08].

defined from mathematical functions. Patches are better than polygons when smooth curves are required, while polygons are more flexible and quick to render than patches [Pare08]. For more detail on how patches work, please refer to [Salo06].

Polygonal Representation

Polygonal representation is one way of representing the human figure. A polygonal model (often called *polygonal mesh*) is a collection of vertices, edges, and faces [ToMa06]. A *vertex* defined as a single point in space and defined as the smallest sub-object of polygonal mesh. A vertex can contain information about color, opacity, lighting, normal vector, and texture coordinate. An *edge* is a straight line between two vertices. A *face* is the smallest part that can be rendered in polygon mesh comprising a closed set of edges (see Figure 3.16). Faces are mostly defined as triangle faces (with three vertices and three edges) to simplify rendering, but they may also be composed of other faces (such as *quad face* with four edges). A face in polygon mesh also has a *normal* to be able to tell the engine on which side to render the face. A face is normally drawn on the side defined by the normal (Figure 3.17).

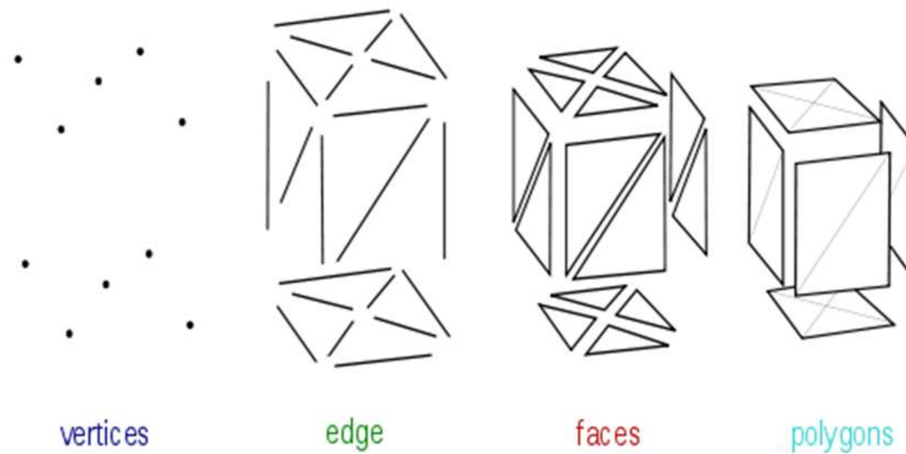

Figure 3.16: Vertex, edge, face, polygon, and surface in polygon mesh [Wiki00a].

In some other 3D applications the normal of the face can also be defined by *vertex normal*. Vertex normal is stored in the vertex and a face could obtain its face normal by calculating

the corresponding vertex's normal. Another use of vertex normal is to determine the light received at that vertex by rendering (Figure 3.18 [c] and [d]).[9]

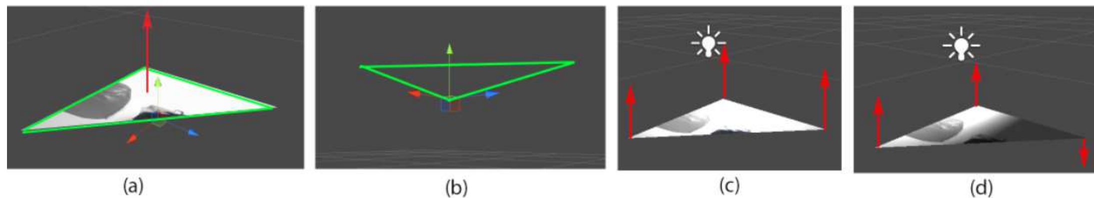

Figure 3.17: Polygonal mesh with normals (shown in red arrow) in (a) a face draws the texture based on its normal (b) shows the triangle from below (notice there is no texture in it). (c) And (d) show the vertex normal and how it influences the light it receives (the triangle is directly under the light).

Virtual character created from the environment developed in this thesis is based on an images and will be moved in 2D. The polygonal representation for the character is a simplified form of polygonal mesh (vertices has the same z coordinates).

Texturing

After defining the polygonal meshes the next step would be to map the texture onto the object. Whenever a polygonal mesh object uses texture as part of its surface, it needs mapping coordinates to tell the renderer how to apply the texture into the object.

---

[9] Since the tools to be used later in this thesis use no lighting, face normal, and are determined by faces (not by vertex), vertex normal will not be discussed further.

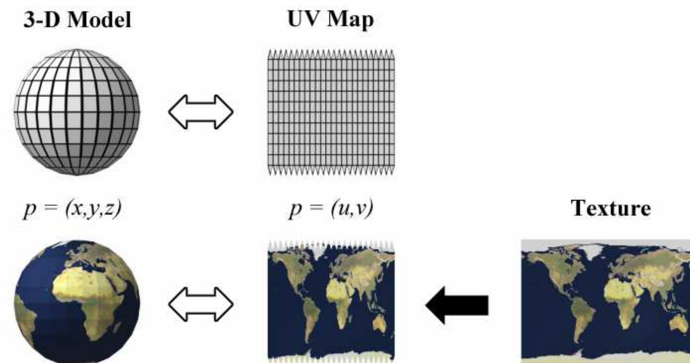

Figure 3.18: UV mapping from texture into the polygonal sphere object.[Wiki00b]

The process of rendering 2D representation into the 3D object surface is referred to as *UV mapping*. In UV mapping "u" represents one side of a texture and "v" represents the other (the coordinate of 3D is x, y, z; while 2D uses u, v in figure 3.18). Normally, the vertices of polygonal mesh hold the information regarding the position they have in the UV map, and the renderer will draw the faces based on the coordinate texture in the vertex.

Virtual character created from the environment developed in this thesis is based on an images and will be moved in 2D. Unlike the example of figure 3.18, the polygonal representation for the character is a simplified form of polygonal mesh (vertices has the same z coordinates).

Skinning

The vertex that holds the texture coordinate information makes sure that the renderer always correctly draws the texture. It also means that by modifying or transforming the vertex into another position, it will also stretch or deform the texture. In animation, this idea can also be used to show skin and muscle flexibility of the human character.

In order to create such an effect, the vertex of polygonal mesh needs to be attached to the position of joints [Owen99]. In skeletal animation this process of creating associations between visual representations with each skeletal points (or joints) of character is called *Skinning* [JaTw05]. In creating skeletal animation with skinning, every vertex in a single polygon has information about the skeleton point to which it is attached [Sori00]. In order to connect every vertex in polygonal mesh to a single skeleton point in a tree structure, the mesh vertices will go through two transformations: transformation from the mesh-defining

object space into world space; and then from world space into the local space of the corresponding skeleton point. Once a vertex of a mesh is in the skeleton point's local coordinate, the skeleton point's current transformation (be it translating or rotating) will also applied to the mesh's vertex [AkHH08].

Another information for vertex is that every vertex have an influence value, which is referred to as *bone weight* (or simply *weight*). It defines how much the transformation of skeleton point influences the transformation of a vertex. It differs in 3D applications, but mostly the weight value ranges from 0.0 to 1.0 (1.0 means the joint's transformation influences the whole vertex, while 0.0 means that the joint is not influencing the vertex). A vertex can have bone weight from multiple skeleton points.

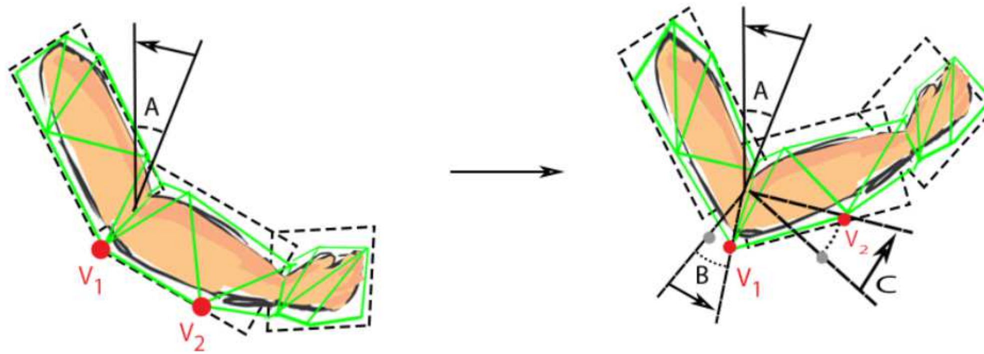

Figure 3.19: Deformable object before and after transformation: A is the rotation of a skeleton point, while B is the rotation of vertex v1 and C is the rotation of vertex v2. Vertices v1 and v2 are attached to the elbow rotation and thus follows its orientation.

For example, in figure 3.19, we have vertices V1 and V2 that associate with the elbow joint with weight value: 0.8 and 1.0 (shown in red dot). When rotation A is applied to the elbow, then rotation applied to vertex V1 is B = A * 0.8 and rotation applied to the vertex V2 is C = A * 1.0. These calculations also applied to other transformations such as translations and scales. Using bone weight on each vertex creates an illusion of stretching skin and muscles.

Although polygonal representation is easy to use and easy to manipulate, there are certain drawbacks. One of the drawbacks of polygonal representation becomes apparent when the polygonal model is used to create a smooth, curved surface (for example, a sphere): it can take hundreds or thousands of polygons to achieve this because an edge in polygonal mesh

is a straight line. In *patch* representation the same curved smooth surface could be attained easily.

3.3 Motion Capture

Motion capture (or *mocap*) is the process of tracking and recording the movement of people. It is used in many applications including animation for virtual character. Output data from motion capture can also be useful to analyze human movement in general. For example, motion capture data of an athlete's movement is useful to track their performance [MoGr01]. Motion capture methods mostly use a camera to track the movement of people. There are many areas of human parts that can be tracked by using the motion capture system, but the method is mainly used to record large body part movements, such as feet, limbs, arms etc. [10]

Motion capture uses several technologies, and it can be divided into two categories: *marker-based* and *marker-less*. The marker-based system requires the subject to wear an instrument (marker) for movement tracking, while the marker-less system uses no instrument on the subject.

The project by Rosenberg et al. [RoBB13] uses marker-based system for interacting in the virtual world. With marker attached to each hand, hand movements will determine the movement in the virtual world. The use of motion capture as an interactive tool is also found in [SaKo11]. Using a time-of-flight camera, this system can track hand movements without the aid of markers. The system will recognize gestures based on motion capture template data and attempt to learn the unrecognized gestures.

Creating imitation and interaction through virtual characters requires a motion capture system. Other than mapping the skeleton point position from the participant to the virtual character, this component is also required as an input when interacting with virtual characters (e.g. using gesture recognition). The next section will discuss the theoretical background needed for developing the desired environment.

[10] Motion capture is also used in small and more detailed human body parts, such as facial expression and hand gesture.

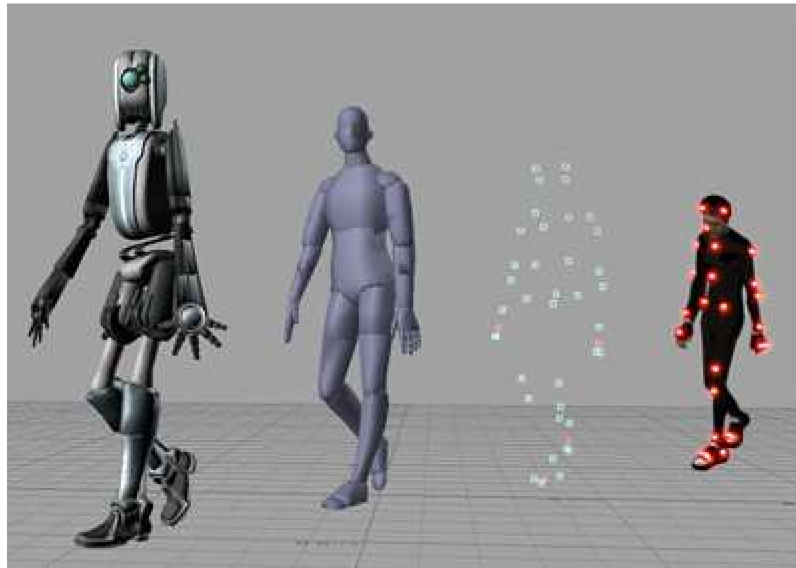

Figure 3.20 Motion capture by using LED infrared marker for animating a virtual character [Wiki00c]

In marker-based motion capture there are many variations of markers that can be used for recording human movement. One example of marker-based motion capture is the use of *active markers,* which are placed on objects (joints and limbs) and visible to the camera (figure 3.20). The markers illuminates infrared lights, which will be captured by the camera. This method needs usually more than three cameras for a full body motion capture [MoGr01]. After recording the movement, the system calculates the position of every marker from cameras and constructs 3D positions by using *triangulation* algorithm [HaZi05]. Having the 3D position, the movement can be mapped into the skeletal animation of a virtual character. The advantage of using this marker-based method is that a position can be recorded and displayed in real time. The drawbacks are: the marker-based method needs many (special) cameras and also restricts movements when using the instrument [Owen99].

The second method of motion capture technique is *marker-less* motion capture. Unlike the marker-based method, the marker-less system does not need special preparation (wearing a marker) for the subject in order to track the movement. As there is no marker involved, the marker-less system uses a tracking algorithm to identify the subject (in this case people). The algorithm usually uses a distinct feature of the subject to identify movement [YiJS06]. For example, identification of a human shape can start from identifying the shape of a human head or the color of human skin (color).

One example of the marker-less motion capture system is a camera with a depth sensor, such as *Microsoft Kinect* (figure 3.21 [a]), which is widely used because of low cost and portability. This kind of optical system does not require the object to wear optical markers.

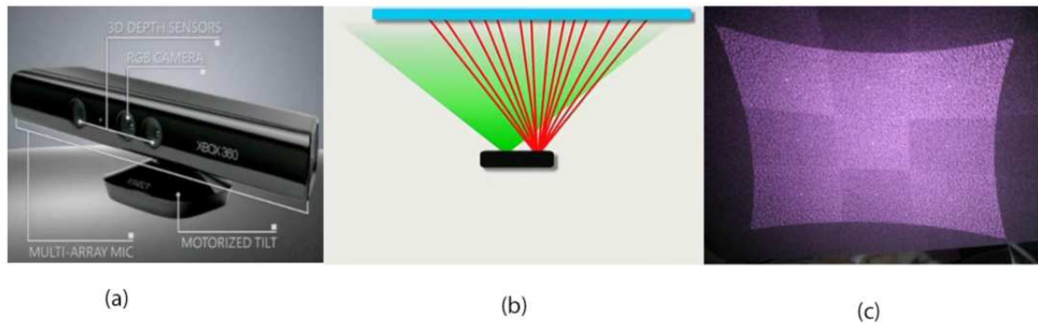

Figure 3.21: (a) Kinect camera and its sensor/projector; (b) the IR depth projectors emit light patterns detetcted by the infrared depth sensor; (c) the light dot pattern from infrared projectors.[Zhao14]

Kinect is a motion capture device from *Microsoft* and it was developed as *Project Natal* in 2010. This has an *infrared projector* and an *infrared sensor* for getting depth images (figure 3.21 [a]). The Infrared projector emits an invisible infrared structured light pattern, which is detected by the Infrared sensor (figure 3.21 [b]). The infrared sensor will pick up the distorted pattern of the infrared, and the camera calculates the depth image [Macc00] [ZhCS02].

Afterward, Kinect uses the data from depth image to create 3D skeletons. With the input of depth images they have a three-dimensional surface model of the human body. This is divided into distinct body parts with the help of an object recognition algorithm (color indicates part label), and the 3D location of each joint within body parts is generated (Figure 3.22) [SFCS13].

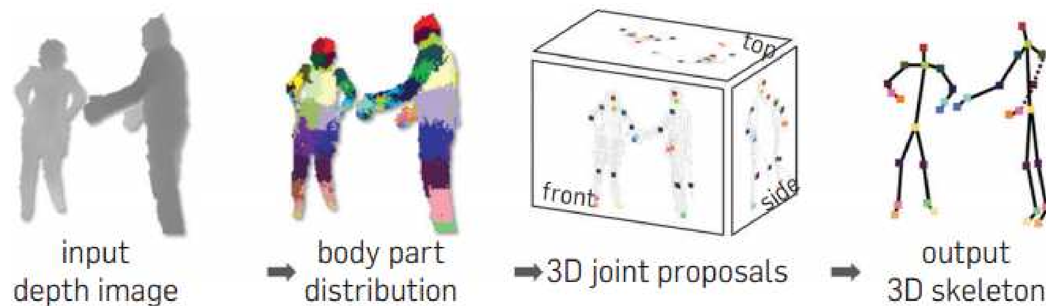

Figure 3.22: How Kinects get skeletal positions [SFCS13].

In general, the marker-less motion capture method is better than the marker-based system because no instrument is required for tracking the movement; however, the former method is slower because the model requires more time calculating a 3D skeleton. Because of the portability and low cost, this thesis use the Kinect camera as its motion capture system.

Motion capture is a powerful and useful method for record realistic movement. Though it will not entirely replace animation, it can be used to ease the workflow of the animator. Other than duplicating the movement of the object, motion capture can also be useful in other ways. In many interactive installations, for example, the motion capture technique can be used to create an fictional environment, as in VIDEOPLACE [Krue77] [Owen99].

3.4 Conclusion

This section has so far discussed how animation can be generated by computers. The key-frame system is a tool, derived from tradition animation, which can sufficiently describe motion by using interpolation between key frames. As in traditional animation, the animator will create key frames, and the computer will interpolate the values between the key values in key frames. An interpolation function between the key frames will be configured by the animator so as to have control over motion.

Having the means to describe motion with the help of a computer, one can now start to create a virtual human by starting from an abstraction of skeleton and bones. The use of a hierarchical structure that helps to create relative motion to describe the skeleton has been discussed. The skeleton of human character will be defined by a tree-structure that enforces transformation constraint from parent's bone into the children's bone. To create motion over

hierarchical models, two methods are available: forward kinematics and inverse kinematics. While forward kinematics gives total control over the end pose, forward kinematic could be tiresome for the animator in configuring specific locations at the end of the hierarchical linkage just by using variables from the rotational parameter of joint. On the other hand, inverse kinematics requires the animator to give an end-point position of the hierarchical linkage and then to calculate the best configuration of the rotational value of joints. The drawback of inverse kinematics: it could produce unwanted end pose in case of numerous configurations. Both methods will be used in this thesis. Simple inverse kinematics will be used in character modeling, while forward kinematics will be used to animate the character.

After defining the skeleton and its bones, the next step would be create a human shape. In order to avoid a rigid structure by just attaching texture onto bones, deformable object represented as polygonal mesh with texture mapped onto it will be used. Every vertex in polygonal mesh will be attached to a skeleton point through the skinning method and following the skeleton movements that stays intact. Changing or rotating the skeleton point will deform the polygonal mesh and create flexibility representing the human skin (or clothes).

Since the purpose of this thesis is not just animation of human character but also an interactive application, motion capture is required to track the participant's movements as well as to detect/recognize gestures. There are two methods: marker-less and marker-based. While marker-based motion capture is faster in terms of computing the position, marker-less motion capture is more suitable because it does not need the participant for preparation. Marker-less motion capture cameras, such as Kinect, which use the structured light method to create depth images is considered in this thesis also because of their portability and low cost.

4 Methodology

This section discusses how the theory presented in Section 3 (theoretical background) will be applied to build an environment for creating interactive applications. Sections 4.1 and 4.2 discuss functional and non-functional requirements for the environment. Section 4.3 presents discussions on how the system design will be built later. Section 4.4 describes the tool and libraries that will be used in this thesis, while Section 4.5 discusses the system architecture and component in detail.

In this section there are two main subjects: developer and participant. Developers are those who will use the developed environment of this thesis in order to create interactive applications. Participants are the ones who will use the applications made by developers.

4.1 Functional Requirements

The functional requirements for developing an environment can be divided into three major groups:

Character Modeling

From the developed environment the developer should be able to create characters that will be used for imitation and interaction of the developed application. The developer can import a graphic format to the system, and the system will process it in order to create a character model based on the graphic. Characters developed by the system would be two dimensional.

Imitation

Imitation means that applications produced by the environment developed in this thesis should have possibilities to create an interactive application that will imitate the movement of the participant. The character created by the developer through character modeling will imitate the movement of the participant. However, imitation of any detail body part of the participant, such as fingers and facial expression, has not been considered here.

Interaction

The developer should be able to create applications that interact with participants. In this application virtual characters created through character modeling will interact with the participant on the basis of the participant's gesture. The participants' gestures will be recognized from the system, and they will trigger an animation sequences of virtual character. To implement this requirement the developer need three functionalities: first, the developer should be able to record and save the gesture sequence as an input; second, the developer should be able to create and save the animation sequence of a virtual character; and third, the developer should be able to detect which gesture is a trigger for which animation sequence.

4.2 Non-functional requirements

Performance

Since our system is an environment to build interactive applications, the processing from recognizing an input to providing the output should be done as quickly as possible. The output should be calculated fast so that the participants do not recognize delays and feel as if they are interacting with animated character.

Usability

The developed system should be easy to use and applicable—not only for programmers. Creating an interactive application with the developed system should be possible without the need to write code or having programming knowledge.

4.3 System Design and Solutions

In this section, the required components will be proposed in order to have functionalities described above.

Skeleton Component

For creating a virtual character that also moves, the skeleton of the character and its shape (skin) are needed. The skeleton will be used to control the movement of the character, while the shape will be the visual representation of a character.

In order to create the skeleton, there should be a component in the system that is responsible for modeling a human in a hierarchical structure (s 3.2.1). Other than creating the hierarchical structure, this component is also responsible for movement/animating over the skeleton by moving its rotational joint. Such a structure provides two advantages in modeling and animating character: it makes sure that bones (and its skin and muscle) are always intact and it simplifies the movement system by just rotating the joint of the skeleton (s 3.2.2). So in order to gain such a structure, the data model of the developed system should support the tree structure, where every object has its own local coordinate system and transformation of a parent in the tree structure influences transformation of its children. After having a way to create the skeleton, we need a component that responsible for creating the shape of the character.

Shape Component

As one of the functional requirements, the developer should be able to create character shapes from a graphic format. So in order to gain this functionality, a component for creating the shape is needed. In order to easily trace the shape of a character in graphic format, the graphic should have transparent background. The component will calculate the necessary information and create a polygonal mesh (s. 3.2.3), which will be attached to the skeleton from the skeleton-component.

Motion Capture Component

At this point, we already have components for creating and modeling virtual character. The requirement of character modeling is fulfilled, but not for imitation and interaction. For imitation and interaction requirements to be implemented, information from the motion capture component is needed (s. 3.3). In addition, interaction also requires an animation system.

In case of creating imitation, we will use a skeleton-component that already has the functionality to move the skeleton along the hierarchical structure. This component will take the required information from motion capture component and transforms it into the motion of the character. The 3D position of the participant will be transformed into a 2D position of the character.

Gesture Component

Gesture component is the component for managing most interaction requirement. This component has three main functionalities: recording Gesture, recognizing saved gesture, and connecting between gesture and animation. In order to record and recognize a gesture, the

position of the skeleton joint from the motion capture component is required. Using the motion capture component, the gesture component will take a 3D position of the developer in every frame and saves it as template gesture. Later, this template gesture will be used to identify a similar gesture of the participant using external library. Another functionality of this component is to connect the gesture with matching animation. If a gesture from the participant is recognized as the similar template gesture, it should *trigger* the matching animation sequence from the animation component.

Animation Component

Virtual character response will be represented in animation sequences. Creating animation sequences for a character requires a key-frame system. The developer that uses the animation system from this component will be able to create animation sequences based on the skeleton from the skeleton component (s. 3.2.1 and s. 3.1.1).

Conclusion

Figure 4.1 show how components work together: shape component will provide the visual representation by creating skin based on images, while skeleton-component responsible for moving the character skin.

For imitation functionality later in the application in the runtime, the motion capture will be used for getting participant's skeleton position and those information will be delivered to the skeleton component for imitating the movement.

For interaction functionality, more preparation is needed: when developing an interactive application using this environment, developer should create a template gesture using the gesture component functionality. Creating a template gesture require access to the motion capture component. Additionally, a developer also needs to create character animation sequences as response using the animation component. After mapping each gesture to each animation sequence, the application can be built.

Later in the application, the gesture component will recognize input from the motion capture component based on template gestures and trigger animation from the animation component. The animation component will execute the animation sequences for the skeleton component.

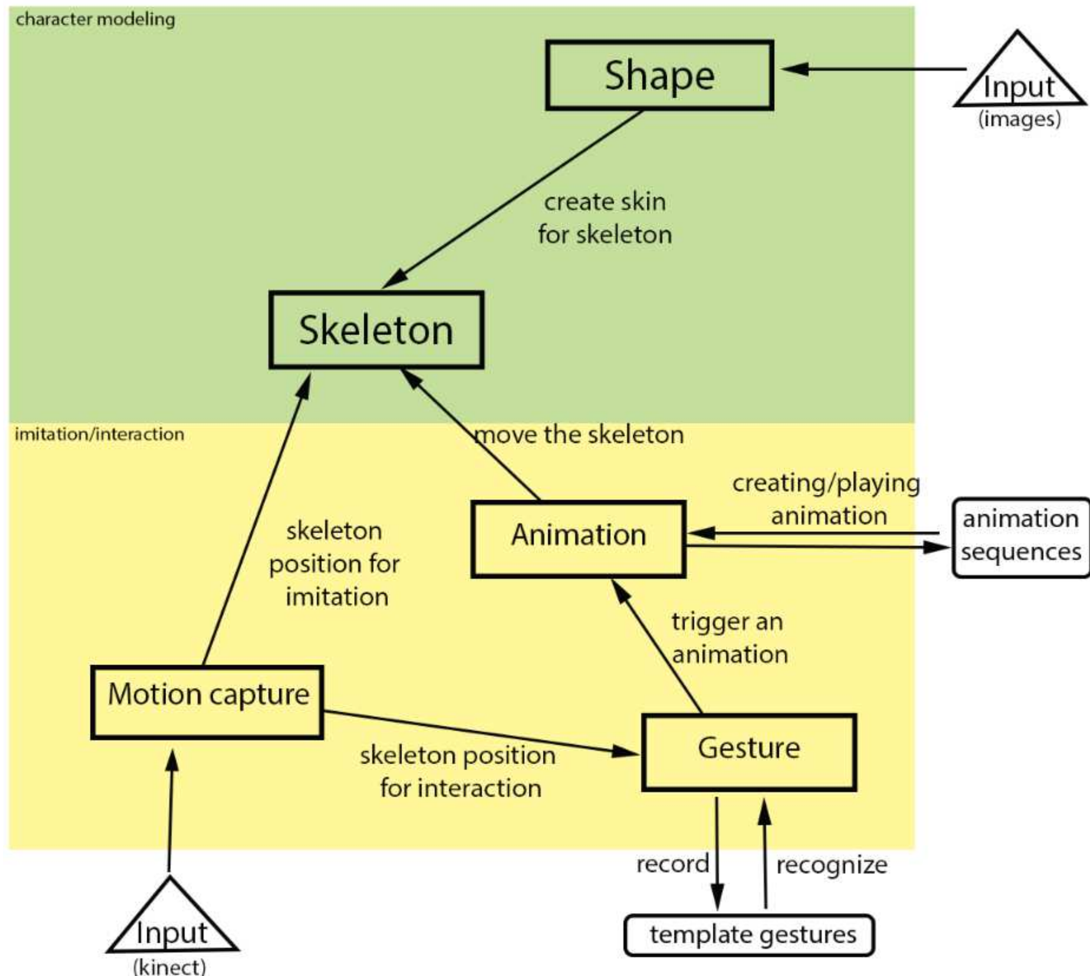

Figure 4.1 Component overview

The solution to designing those functionalities will be divided into two parts: one part is where the existing functionality from the external libraries and tool will be used in order to create the developed system; and another part is how this thesis brings the pieces of the libraries and tool together while extending its functionalities in order to create the desired environment.

Some functionalities, which have just been described, such as the animation system and the hierarchical structure are already available as integrated functionality of *Unity* (tool used in this thesis). This thesis will also use external libraries for functionalities that Unity does not

provide. The motion capture component is covered within the Kinect plugins for Unity, while gesture recognition functionalities and creation of the character shape will use external libraries. The next section serves as the introduction to the tool (Unity) and libraries that will later be built under this thesis.

4.4 Tool and Libraries

This section will discuss the tool and libraries that will be used in this thesis. First, the tool Unity will be introduced and the general workflow idea developing an application through the system explained in Section 4.4.1. At the end of this section we will discuss the Unity functionality that does not cover the desired functionality and how it will be extended. Section 4.4.2 will describe libraries that will be integrated into the system. The reason for these libraries being chosen and the process of their implementation into the developed system will also be discussed. The end of this section overviews the whole functionalities from Unity and external libraries in relation to the system design and solutions from Section 4.3.

4.4.1 Unity

Unity is a *cross-platform game engine* for creating games in iPhone, Ps3, PC, and many more. An application in Unity could be developed in C#, Boo, and JavaScript [Unit00a]. Unity has been chosen as the base engine because of its built-in functionalities and supports: first, its data structure in world space supports a hierarchical model called *parenting* so that implementation of the skeleton of virtual character (s. 3.2.2) could be applied without making big changes; second, Unity is expandable. It means that functionalities not provided by Unity could be extended by using *scripting*. Through scripting the user could also built an extended editor or external libraries. Unity also supports the creation and definition of polygonal mesh representation. Lastly, Unity has its own animation system called *Mecanim* that supports the key frame system [Unit00b].

In order to understand better how to develop a system in Unity, the following section will discuss the basic component and workflow of Unity. Afterward, the next section will describes and discuss Unity’s parenting system, scripting for extending functionalities, the creation polygonal mesh and lastly the animation system in Unity.

Workflow and Primary Objects in Unity (GameObject, Component, and Assets)

In this section, workflow and primary objects in Unity will be discussed. In Unity, an application's development occurs in two phases: testing phase, where an application is executed in runtime; and edit phase, where an application is edited and developed in Unity. In the following sections we will describe the testing phase as *Play-mode* and the edit phase as *Edit-mode*

Basically, the data model in Unity consists of three basic parts: GameObject, component and assets. The workflow in Unity include: preparing the asset (from external applications) or creating assets in Unity, creating GameObject, adding components based on their functionalities to GameObject, attaching the asset to the component as property. The developer could test the application in Play-mode, if needed, and iterate the process before building the application (Figure 4.2)

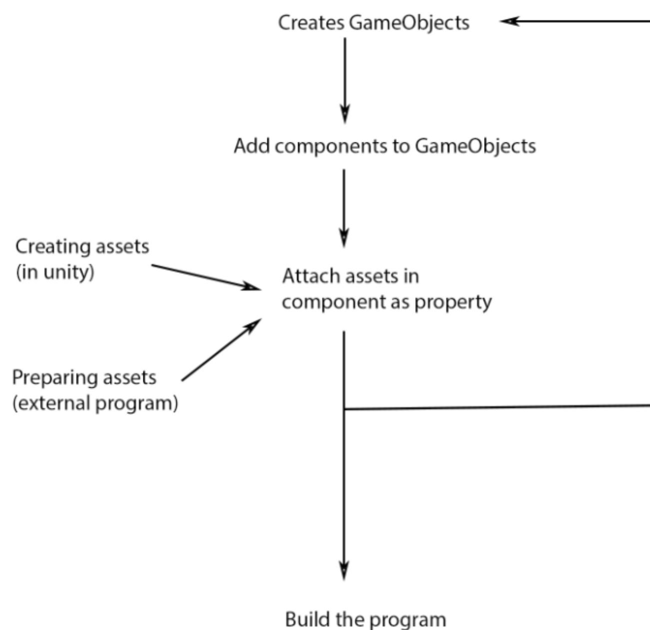

Figure 4.2: Workflow in Unity

GameObject and Component

Every object in the world space of Unity is defined as a *GameObject*. GameObject by itself does not have any functionality other than defining its position in world space. GameObject acts as a container for *Component* [Unit00c]. Components determine which functionality a GameObject has, and every component has unique property, which user can modify through an inspector view. For example, if a GameObject has light component, it emits light and can be used in world space as a light object, and light component has properties such as color and intensity (Figure 4.3).

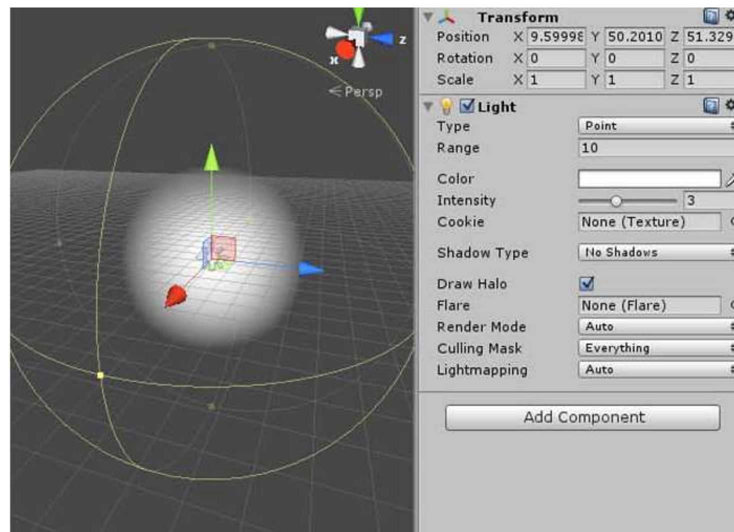

Figure 4.3: A GameObject with light component. The right window is the inspector view that shows the components of selecting a GameObject. The left window is the scene view, the world space of Unity.

A component of a selected GameObject can be viewed through the inspector view (Figure 4.3 right). The user can add new components of a GameObject through this view in order to add functionality of a GameObject. Another way to add or access a component is through code.

Assets

Other than GameObject and its Components, it is also important to mention the assets in Unity. Every data that could be useful for the components of a GameObject are referred to

as an *asset*. Asset could also be assigned or modified through code. The kinds of asset that will be used in this thesis are *scripts*, *polygonal meshes*, *material* (texture), *animations clip,* and *animator* (animation transition controller). Any code from external libraries/plugins will be recognized as scripts in Unity.

Parenting system

Unity use the concept called *parenting* [Unit00d]. By making a game object a child of other game object, the child will inherit the movement and rotation of its parent. Doing so also makes the child having its own local coordinate system relative to its parent (parent position is child's origin coordinate system) and does not work within global coordinate system. Parenting in Unity could be managed within one view editor called *hierarchy view* (Figure 4.4). In this window a user can manages, access, creates or groups game objects. The user can select one object and drag it onto another to group it together within one game object. The parenting system of Unity represents the hierarchical structure from Section 3.2.

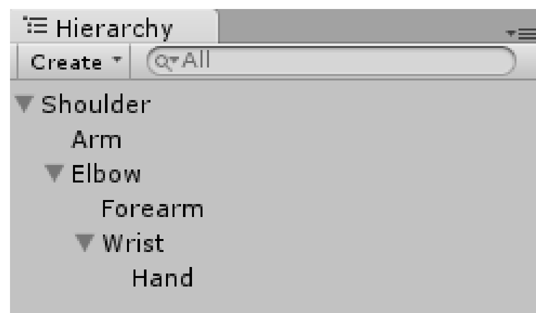

Figure 4.4: Hierarchy view creating hierarchical structure in arms

In this thesis, every joint of skeleton will be represented as an empty GameObject. Although GameObject itself does not have functionality, we could controls the movement of child GameObjects with the parenting system. For example, if we arrange GameObjects as in Figure 4.4, then the changing rotation of "Elbow"-GameObject will result in the changing position of "Forearm", "Wrist", and "Hand"-Gameobject.

Scripting in Unity

Unity also provides the user with a way to create customized functionality through a component called *script component*. Scripts can be used to create graphical effect, extending editor, artificial intelligence, or to control animation. It is also through scripting that the functionality of other external libraries in this thesis will be integrated and extended.

Extending Unity functionalities using scripting will use classes derived from MonoBehaviour, Editor, or EditorWindow. *MonoBehaviour* is a class of *UnityEngine*-namespace. Editor and EditorWindow are classes of *UnityEditor*-namespace (Figure 4.5). Classes derived from *MonoBehaviour* will be used for creating dynamic functionalities during Play-mode. Classes derived from Editor are used for building custom inspector views and custom editors on-scene view for a MonoBehaviour class. Classes derived from EditorWindow are used for creating custom windows. Both classes from UnityEditor-namespace will be used only for functionality during Edit-mode.

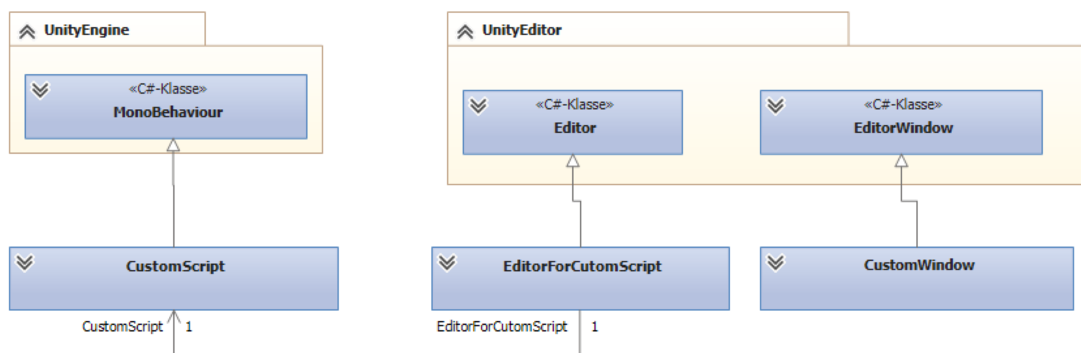

Figure 4.5 Inheritance class from Unity Engine and Unity Editor

MonoBehaviour

MonoBehaviour is a class from the UnityEngine-namespace, which is a base class for every script component in Unity. Most functionalities coming from the classes derived from *MonoBehaviour* will be used in runtime. It has many event-based functions such as *Update* and *Start*[11]. The Update function will be called for every frame in runtime, while Start will be called one time before Update function. Both functions will be used mainly for getting positions of participant in every frame (Update) and initializing library instance before using it (Start).

Another event-based function that will be used in this thesis is the *OnGUI* function. This function will be used for rendering and handling of GUI events. *OnGUI* of MonoBehaviour will

[11] They also had other event-based functions like *OnTriggerEnter* or *OnMouseEnter,* but most of them will not be used in this thesis.

be used for example to create a menu for gesture recording while in Play-mode (s.4.5.3 Gesture).

Other than controlling the game object from where it attached to (through event-based functions), a MonoBehaviour script can access other GameObject and their components. Unity provides a number of different ways to retrieve other GameObject. One possibility is by creating public variables. A public variable in script is visible in the editor *inspector view* (Figure 4.6, right), and through this window a public variable could be assigned while in Edit-mode.

Figure 4.6, shows example of a C# class *CustomScript*, which has been derived from *MonoBehaviour*. The right part of Figure 4.6 shows the inspector view from the script after it has been attached to GameObject and selected. The default inspector view for scripts shows the name of the script and its public variables.

Figure 4.6 CustomScript C# script (left) and default inspector view of this script (right) after it is attached into a GameObject

Editor

Editor is a class of UnityEditor-namespace. Classes inheriting from *Editor* will be used to create the custom inspector view for a *MonoBehaviour* script. With a custom *Editor*-class we can customize the inspector view appearance. Another use of *Editor*-class in this thesis is to create a custom editor for editing component properties directly on the scene view (and not on the inspector view).

Figure 4.7 Example of *EditorForCustomScript* C# script (left) and its custom inspector view (right) for *CustomScript*. Notice there is different with default view of script in figure 4.6

Figure 4.7 shows *EditorForCustomScript* as the *Editor*-class for *CustomScript*. To be an Editor for a specific class it needs to add the *CustomEditor* attribute. Once it has this attribute, if a GameObject with *CustomScript* component is selected in Edit-mode, Unity will call the *OnInspectorGUI* function from *EditorForCustomScript* (Figure 4.7 left side). The *OnInspectorGUI* function will override the default inspector view of this component with the customized one. We can add view component layouts inside this function, such as a button or an input box.

Another functionality of *Editor*-classes, which will be used in this thesis, is to create customized editors for scene views. To achieve this, *OnSceneGUI* function will be used. In the *OnSceneGUI* function, we can add controls for modifying Object properties, such as its position or rotation. By default, a GameObject scene-GUI control, when selected, is colored arrow-headed and square controls (Figure 4.8 right side). This control can be dragged within the scene view, in order to modify the position of GameObject.

Unity provides the *Handles* class for creating the custom GUI control on scene view. This class will be used for creating a custom handle within the OnSceneGUI function. Figure 4.8 shows an example of creating a free-moving handle from the *Handles* class. The function *Handles.FreeMoveHandle* will create a white dot handle on the left side of the current GameObject. The handles can be dragged to the scene view, and it will give its new position. In this example, the new position is assigned to the variable *newstandpoint* (Figure 4.8 left).

```
[CustomEditor(typeof(CustomScript))]
public class EditorForCustomScript : Editor {

    //onScene called per editing in scene
    void OnSceneGUI ()
    {
        CustomScript cs = target as CustomScript;

        Vector3 newstandpoint = Handles.FreeMoveHandle(
            cs.transform.position + Vector3.left, //pos
            Quaternion.identity, //rotation
            0.2f,    //size
            Vector3.one * 2f, //snap
            Handles.DotCap //capFunc
            );

    }
}
```

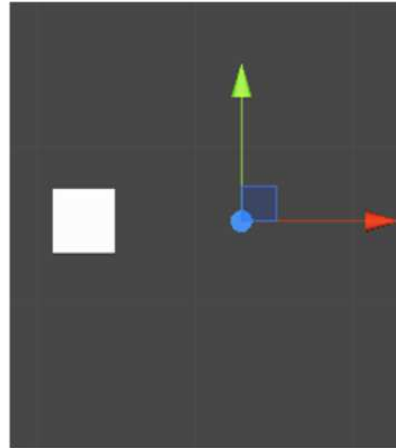

Figure 4.8 : Example of the OnSceneGUI function that creates the handle. First, the current CustomScript instance will be aquired through Editor.target. After assinging GUIstyle font color, a label will be written to the scene through the Handle.Label function.

The only use of creating custom handle in this thesis is only for creating handle for modifying bone rotation in skeleton in Edit-mode. Position of current handle will changes rotation of its parent bone rotation.

Although classes derived from Editor provides custom functionalities in its inspector view and scene view, *Editor*-classes is dependent on MonoBehaviour classes. From our example, the EditorForCustomScript class is an *Editor*-class for the CustomScript Class. The functions OnInspectorView and OnSceneGUI will only be called when CustomScript is attached to GameObject and selected during Edit-mode.

EditorWindow

Sometimes a GUI is needed to provide functionalities that are not dependent on currently active and selected GameObject, nor should it be attached to GameObject. This GUI could be created through classes derive from *EditorWindow*. *EditorWindow* will be used for creating a custom editor GUI window in Unity while in Edit-mode. Editor windows are normally opened through menu. Classes derived from EditorWindow usually have two parts: static function for initialization and a drawing function for its contents.

As an example in Figure 4.9 we create a CustomWindow class, derived from the EditorWindow class. It has the static function *Init* for initializing a window in Edit-mode. To call it from Menu, it needs to have the attribute *MenuItem,* and this function will be called once the menu item is selected. Once the window is initialized, the *OnGUI* function will be

called. In Figure 4.9, the *OnGUI* function is used to show two labels (string). As in the *OnInspectorGUI* of *Editor*-classes, the Unity developer can add further view components such as buttons or text boxes.

Figure 4.9 Example of EditorWindow class: CustomWindow script (left), creation of window through the static function Init (right above), GUI of window based on the OnGUI function (right below).

Creating Mesh in Unity

A mesh will be used for defining the shape of a character, and Unity supports defining and creating a polygonal mesh (s. 3.2.3). A mesh in Unity usually comes from external applications such as *Maya* or 3DsMax. Unfortunately, Unity does not provide direct GUI for creating a 3D model and the developer can only create mesh primitives, such as a simple cube or sphere. The only way to create a custom polygonal mesh from scratch in Unity is through scripting.

Polygonal mesh in Unity is represented as *mesh* assets. Mesh assets could be created through scripting by using the mesh class from Unity. In order to create mesh, an instance of Mesh class need at least two information: vertices and triangles [Unit00e]. The vertices of a mesh

is an array from the class *Vector3*[12] that defines the vertices' position in a 3D world. Vertices will be stored in *Mesh.vertices* variable. Each triangle of a mesh is represented in an array of integers (*Mesh.triangles*), while every member of this array represents an index of the vertices array. So if a mesh has three vertices, then triangles should have a minimum of three integers.

Having only a mesh without texture in Unity will result in blank shapes on the screen. Calculating texture to be added in polygonal mesh require extended calculations and s special asset called *material*. After attaching texture to a *material* asset, it could later be used within component as a property for defining texture. Mesh objects need an access to *material* asset and to calculate its texture's UV map (s. 3.2.3 Texturing). In calculating the UV map, every vertex in mesh will have information in which position in a texture does a vertex has [Unit00e]. In a mesh instance, UV map information is contained within the array of *Vector2* (*Mesh.uv*), which represents the UV map for every vertex in array *Mesh.vertices* with the same index.

After creating a mesh, we add it to the GameObject through the *SkinnedMeshRenderer* component (Figure 4.10). This component is a core component for attaching the skeleton to its mesh (*Skinning* in s. 3.2.3). Other than having information about mesh and material (texture), this component also needs an array of bones in skeleton (in this case an array of the *transform* class. This array of bones will be stored in variable *Mesh.bones.*

The mesh attached to this component also needs additional information such as vertex's *boneWeights* (which represented as an array of the *BoneWeight* class) and *bindposes* (which is represented as an array of *Matrix4x4* class) [Unit00f].

As mentioned in S.3.2.3 *Skinning*, for binding a vertex to follow the transformation of bone, they need two transformation which is transformation from the vertex local coordinate into the world space coordinate, and then, from the world space coordinate into the bone local coordinate. In Unity, those transformations is represented in Mesh.*bindposes* array, which is represented as the *Matrix4x4* class. The *bindposes* of each index in this array refers to the bone of the same index in *SkinnedMeshRenderer.bones*.

---

[12] Vector3 is a Unity class that represents the position of x, y and z. In case of the vertices of a mesh, this represents the position in object space.

A vertex weight in Unity is represented as *BoneWeight* Class. A vertex can attached up to 4 skeleton point (GameObjects) and all weights should sum up to 1.0 [Unit00g]. For example if a vertex attached into 2 Game Objects (A and B) and GameObject A and have weight vertex.weight0 = 0.6 then B should have weight vertex.weight1 = 0.4. A Mesh in *SkinnedMeshRenderer* should have *BoneWeight* information for every vertex in it.

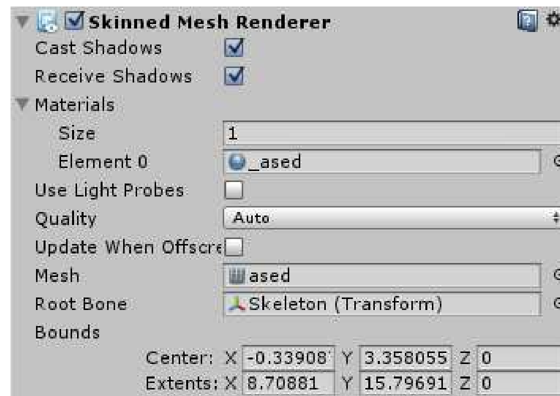

Figure 4.10: Skinned mesh renderer component, it takes material (texture), mesh, and root bones (Game Object's transformation).

Animation System

Unity provides an integrated animation system called *Mecanim*. Mecanim consists of many animation tools, such as key frame animation editor and state machine for transition between animation clips and other functionalities. The typical Mecanim workflow in Unity consists of setting and creating animations clips, setup of transitions between clips, and lastly controlling the animation from code [Unit00h].

An Animation in Unity is defined in *Animation Clips*. They represent a piece of looping motion such as running, walking, or simple movement of an object such as a point moving from left to right [Unit00i]. An Animation Clip contains the information about key frames and also interpolation curve that can be edited and created through animation window (Figure 4.11 above and middle). Animation clip is being saved as an asset in Unity.

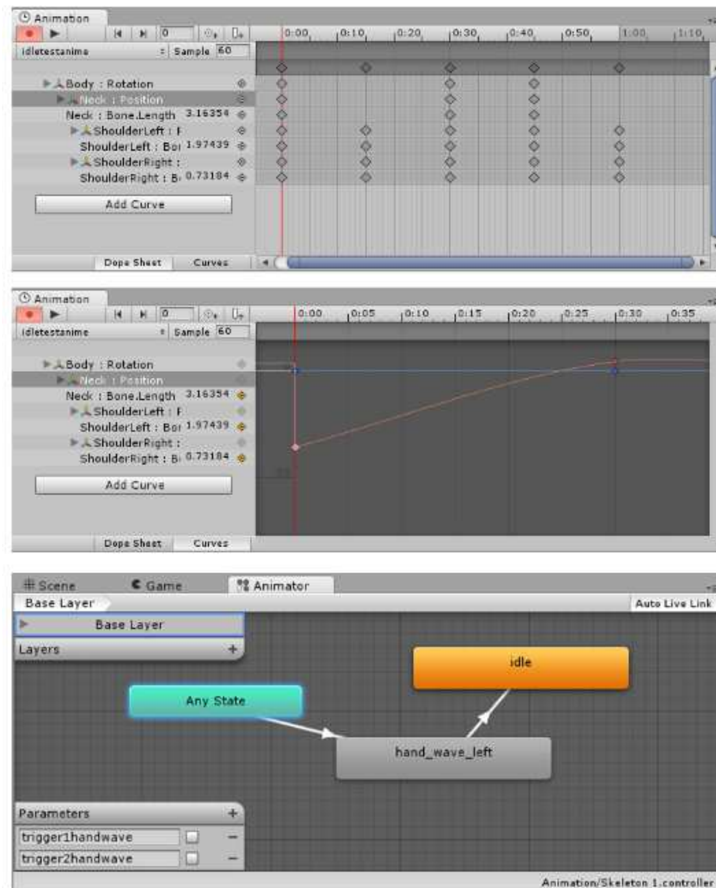

Figure 4.11: Animation window in Unity: Dope Sheet window (above) for managing key-frame, Curves window (middle) for managing interpolation function and animator window (below) for managing transitions.

It is not enough to just having animation and attaching it to GameObject. There is also a need for Attaching multiple animation altogether for creating complex behavior (s. 4.1). If a GameObject is supposed to have variation of animation sequences, or it needs to “behave’” based on conditions, then GameObject can use *Animator* and *Animator controller* [Unit00j]. *Animator* is a component and *Animator Controller* is an asset. With an *Animator Controller,* developer could keep many animation clips and switch between them with a predefined condition from developer. Another use of *Animator* and *Animator Controller* is blending between animation clips. The developer could define how the transition between two animations occurs.

Transition between animation clips could be edited within *Animator window* (Figure 4.11 below). This window shows Animation clips as a state machine and their transition. Through this window we can add new transition between states and determine the parameter as conditions for switching animations clips. In this thesis we use only parameter type 'Trigger'[13].

After finishing the setting up the transition between animations, the animator will be attached into the root skeleton (GameObject) and could be accessed within script with *GetComponent<Animator>* function. From this, developer could access the *Animator Controller* parameters and trigger the animation.

Conclusion

Functionalities covered within Unity are the following: hierarchical modelling system, animation system and mesh representation.

1. GameObjects in Unity provide a data model that has been described in section 4.3. A GameObject has its own local coordinate system and could be built into a hierarchical structure through a parenting system. Later in the Implementation, GameObjects will act as skeleton joints and those skeleton joints will be arranged in a hierarchical structure.

2. Unity provides animation system for animating GameObjects called *Mecanim*. Through this animation system, developer could create animation complex sequences and controls it through scripting

3. Mesh are basic representation for 3D model in Unity and Unity supports rendering mesh representation. However, mesh creation is not built in Unity and this functionality can only be attained through scripting.

In this thesis, scripting is necessary for creating and extending functionalities. Other than creating mesh, scripting will be used for integrating external libraries in order to cover the functionality of character modelling from images.

---

[13] There are also other parameter type such as boolean and float

Unity also doesn't covers gesture recognition and motion capture functionalities. In order for having those functionalities, external libraries and plugins will be used. These libraries will be described in the next section.

4.4.2 Libraries

Libraries in this thesis are mainly used for these three functionalities: helping for creating polygon mesh, gesture recognition functionalities and also motion capture component functionality.

Creating mesh from graphic will use special libraries from *Physics2DDotNet*[Port00], *Poly2Tri*[Cont00] and *Farseer Physics*[Qvis00]. In order to trace the shape of a character from the graphic format, *Physics2DDotNet* is used because it contains functionality to create vertex from texture by using the transparent property of a texture (alpha channel). Library *Physics2DDotNet* provides the function *CreateFromBitmap,* which gives a list of outer shape vertices that surround a bitmap[14]. *Poly2Tri* library take care of the Triangulation after vertices being made. It uses *Delaunay* algorithm[ILSS06] and produce triangles that useable in Unity. If a vertices generated is too complex or too detail it could be simplified using the *Douglas-Peucker* algorithm [HeSn94] from *Farseer Physics* library. Once mesh is created with its vertices and triangles, it could be used as skeleton shapes through *SkinnedMeshRenderer* of unity

The library from Project [Rymi00] for gesture functionality. This project uses Dynamic-Time-Warping Algorithm[Gent00] for recognizing the input gesture from participant. The algorithm takes the two sequences for calculating similarity. Every element in sequences will be compared to each other and overall distance will be calculated. If the calculated overall distance is qualified (less than threshold), then the algorithm will tell that those sequences are match.

The library within the project [Rymi00] have functionality for saving gesture as template during runtime and recognizing input gesture based on saved gesture. So template gestures should be added first before it could recognize any gesture. This project is written in c# and

[14] Bitmap in this case is a two-dimensional array of Boolean that represent transparency of an image (0 = transparent, 1 = not transparent).

could be integrated within Unity. They covers the functionalities to recognize and saving gesture.

Another important component is the Kinect plugin for Unity. In order to have skeleton position of participants, a Kinect library for Unity will be used [Univ00]. This plugin provides a configured GameObject that developer could add into world scene and provide skeleton information during runtime. If a script have access to the *SkeletonWrapper* instance from this GameObject then it can access *bonePos[index]* of this instance for getting bone position of the desired bone-index. Bones from Kinect plugin covers 20 bones from a participant which is: hips, spine, shoulder-center, shoulder-left, elbow-left, wrist-left, hand-left, shoulder-right, elbow-right, wrist-right, hand-right, hips-left, knee-left, ankle-left, foot-left, hips- right, knee-right, ankle-right and foot-right

4.4.3 Conclusion

Section 4.4.1 shows the functionalities that Unity provides: hierarchical structure through parenting system, basic object GameObjects for creating character bones, animation system Mecanim. Unity also provide *scripting* for extending its functionalities.

Section 4.4.2 describes libraries that cover functionalities that Unity doesn't have: libraries for creation of polygonal mesh from images, Kinect plugin library for motion capture functionalities and gesture-recognizer library for interaction functionality.

The implementation of the environment for this thesis will integrate libraries with Unity and develop additional functionalities such as providing GUIs through scripting. Integration and extension is achieved through scripting.

Given the functionalities from libraries and Unity, functionalities that still need to be implemented for character modelling requirement are: mesh creation based on images, skeleton mapping and Skinning (adding skin to skeleton).

Functionalities for imitation and interaction that still needs to be implemented are: creation of character hierarchical model through Unity parenting system, mapping the skeleton system onto Kinect skeleton points. Lastly, additional functionalities for interaction are: integration of DTW libraries and functionalities to unity, mapping between gesture and animation.

4.5 Design

This subchapter discusses about the design of the environment based on the requirements in s 4.1 and 4.2. Section 4.5.1 explains each component of the developed system and how it will use external libraries. This section will also explains the general workflow from each between each component for creating functionalities from sections 4.1 and 4.2. Section 4.5.2 will give overview on the system implementation component-wise. Lastly, section 4.5.3 describes the implementation environment for developing this system.

4.5.1 Component Overview

The system design of this thesis is explained in Figure 4.12. Components in blue are external libraries, while green components are the developed system. Most of the components in this design are built as a bridge between external libraries and Unity, while some other component are also provide GUIs for developer's input and configurations. These component also covers functionalities that still need to be implemented from Section 4.4.3.

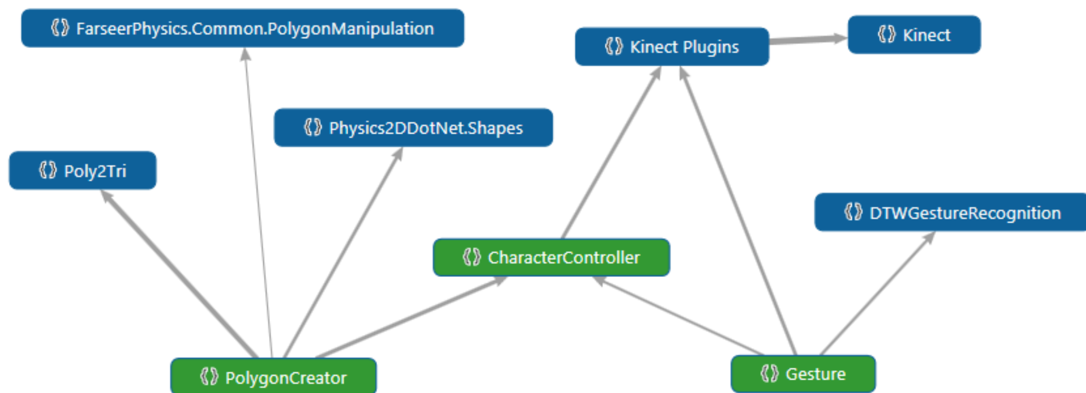

Figure 4.12: System architecture of the developed environment. Components in blue are external libraries, while the green components are internal components.

CharacterController represents the skeleton component from Section 4.3. Its main functionality is to take data provided by the Kinect and to transforms it into movement of skeleton GameObjects. So the main input for this component are from Kinect library (Figure 4.12) and the output is the calculated rotations for every GameObjects in hierarchical

structure. *CharacterController* also extends the Unity editor functionalities, which helps the developer when editing and generating the skeleton.

PolygonCreator represents the shape component from Section 4.3. This component has two main functionalities: first, it creates mesh and material assets from an image with help from external library. They take an image as an input and calculate new mesh that have vertices, triangles, and UV map based on the image. They also take texture asset and creates a material asset from it. Second functionality is preparing and attaching mesh into skeleton. This functionality include creating *SkinnedMeshRenderer* instance and attaching it to the skeleton GameObject, calculating *boneWeights* and *bindposes* of a mesh, attaching the mesh and material asset into *SkinnedMeshRenderer* of the skeleton and lastly calculating bone array for *SkinnedMeshRenderer* (ref. Creating Mesh in Unity Section 4.4.1). *SkinnedMeshRenderer* will be attached to skeleton so that's why it also takes skeleton instance from *CharacterController*

The last component to be implemented is the *Gesture* component. It has the following functionalities: it provides functionalities and a GUI for recording template gestures using library from *DTWGestureRecognition*. This component also provides functionality and GUI for connecting gesture to an animation. Developers could start connecting gestures with animations when template gestures are loaded and skeleton already has an *Animator* component configured[15]. The result of this action is a *GestureController* Script that maps between animation and gesture. *GestureController* script will make sure that the configured connection is happening during runtime.

Developing an environment for creating an interactive application in Unity requires solution in Play-mode and in Edit-mode. After discussing the functionalities component-wise, this section overviews how the components work together based on functional requirements from section 4.1.

Character modeling functionalities are used only in Edit-mode. Figure 4.13 shows how the *PolygonController* operates. In this example, skeleton GameObject is already created complete with *SkeletonController* attached and its child bone GameObjects with *BoneController* attached. *PolygonController* will create Mesh, Material using external libraries. *PolygonController* will then attach SkinnedMeshRenderer to the skeleton

[15] Mean animation, transition, and parameter condition is configured within animator

GameObject. Result of this process is skin (mesh) that follows movement of bones in skeleton GameObject.

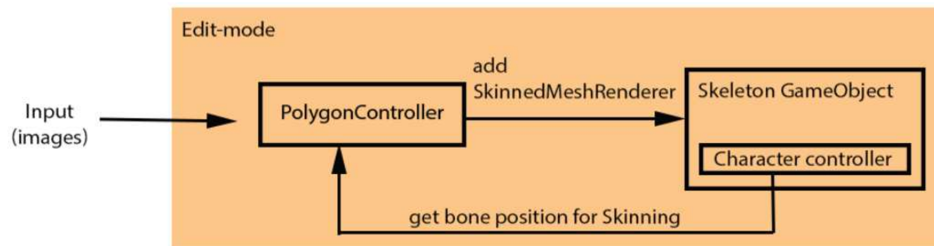

Figure 4.13 Character modelling process

Once the process of attaching skin is finished, the character can be tested for imitation functionality. This functionality occurs only in Play-mode. Through Kinect plugin, the *CharacterController* (*SkeletonController* instance) will access Kinect's 3D skeletal position and calculate the bone rotation (through *BoneController*). The calculated bone rotation will be applied to the character's bone GameObjects. (Figure 4.14)

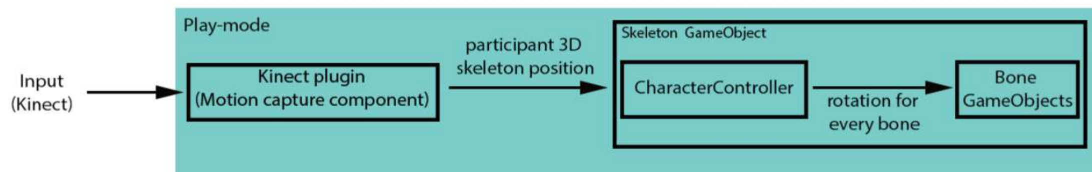

Figure 4.14 Imitation functionality during Play-mode

Another functionality from this environment is interactive functionality. This functionality use both Edit-mode and Play-mode. Gesture component provide recording functionality and requires Kinect-plugin functionality during Play-mode for getting skeletal position. Once the gesture template is created, developer can connects gesture with animator parameter in Edit-mode (Figure 4.15). The result of this process is GestureController instance attached to the Skeleton GameObject as Component.

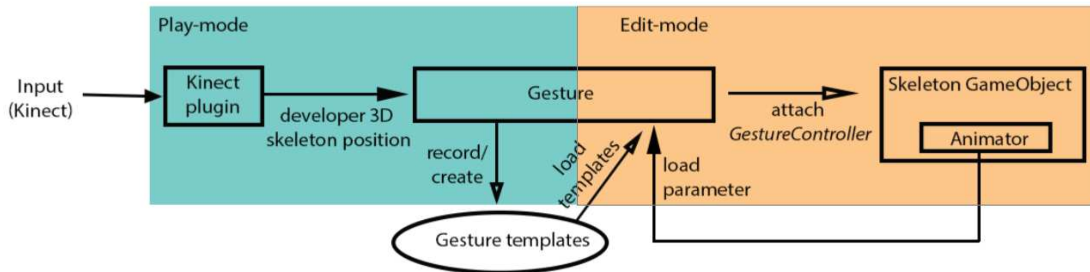

Figure 4.15 Recording a gesture template and connecting gesture with animator parameter

Once developer finished with process of attaching GestureController instance, the character now has interaction functionality. When the application is built later, the *GestureController* of Character's skeleton GameObject will accessing the Kinect for 3D position and try to recognized gesture during Play-mode. Once a gesture is recognized, it will access *Animator* component and trigger the animation clips.

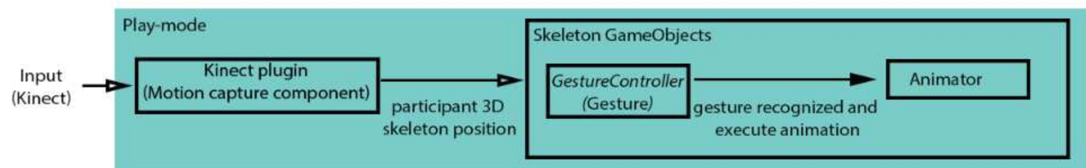

Figure 4.16 accessing Kinect, recognizing gesture and triggering animation

4.5.2 System Implementation

In section 4.5.1, functionalities of every component and their dependency has been discussed. This section will is about how a single component will be implemented using Unity programming model (s. 4.4.1 scripting). Afterwards, the main functionalities of each class for every component will be explained.

CharacterController

As mentioned in Section 4.4.3, the requirement for having a hierarchical structure for the skeleton point is almost covered within Unity. For imitation purpose, the CharacterController

component needs access to the Kinect-plugin. The CharacterController is responsible for mapping the skeleton joint yielded by Kinect to the GameObject skeleton.

For this purpose, the MonoBehaviour classes *SkeletonController* and *BoneController* will be created to manage the character skeleton and access the Kinect plugin (Figure 4.18). Later when developing the skin of character, editing the position of the bone-skeleton in scene-view is also needed (This will be explained later in Section 4.5.3, *PolygonController*). For this reason, the *Editor*-classes *BoneControllerEditor* and *SkeletonControllerEditor* will be implemented.

Kinect skeleton joints are connected to the bone GameObjects of the character inside two classes: *SkeletonController* and *BoneController*. Basically, SkeletonController is a container for *BoneController* GameObjects (reflected in the association between both classes in figure 4.18). *SkeletonController* has access to the Kinect-plugin, and it calculates the vector direction of every bone during runtime and gives the result to the bone's *BoneController*. *BoneController* will be attached to every bone GameObjects for modifying the rotation of a specific bone. Using the vector direction from *SkeletonController*, *BoneController* will update its GameObject rotation.

The class *SkeletonController* has a public parameter called *skeletonWrapper* that refers to an instance of *SkeletonWrapper* of the Kinect-plugin. During runtime, when the *Update* function of SkeletonController is called, the *skeletonWrapper* checks for a position-update through the plugin function *pollskeleton*. If the position is updated, then the *RotateJoint* function will be called for every bone. *RotateJoint* calculates the direction vector from the parent bone to the child bone of Kinect. For example, if the actual bone is shoulder, then *RotateJoint* will calculate the direction vector from shoulder to elbow. After getting a new direction vector, the rotation will be calculated for a specific bone GameObject through the function *UpdateRotationBoneOnDir* of *BoneController*. *UpdateRotationBoneOnDir* will calculate the new rotation, based on the new vector direction, and applies it to the bone where the scripts (*BoneController*) have been attached. This function calculates only two-dimensional rotation because the representation of virtual character is also two-dimensional.

Both *SkeletonControllerEditor* and *BoneControllerEditor* provide functionalities for directly manipulating GameObjects in the scene view during Edit-mode. The OnSceneGUI function in both *Editor*-classes will create a new handle at every position of the skeleton joint (handles for skeletons are depicted as white boxes in Figure 4.17). The position of bones can be changed by moving these handles. In OnSceneGUI function, both *Editor*-classes will check whether the handle has changed and will calculate the new position for bone by calling the function *UpdateStandPointOnBones* within the *BoneController*. In this function, the position

of the actual bone will be changed, and the rotation of its parent bone will also be modified through *UpdateRotationBoneOnDir*.

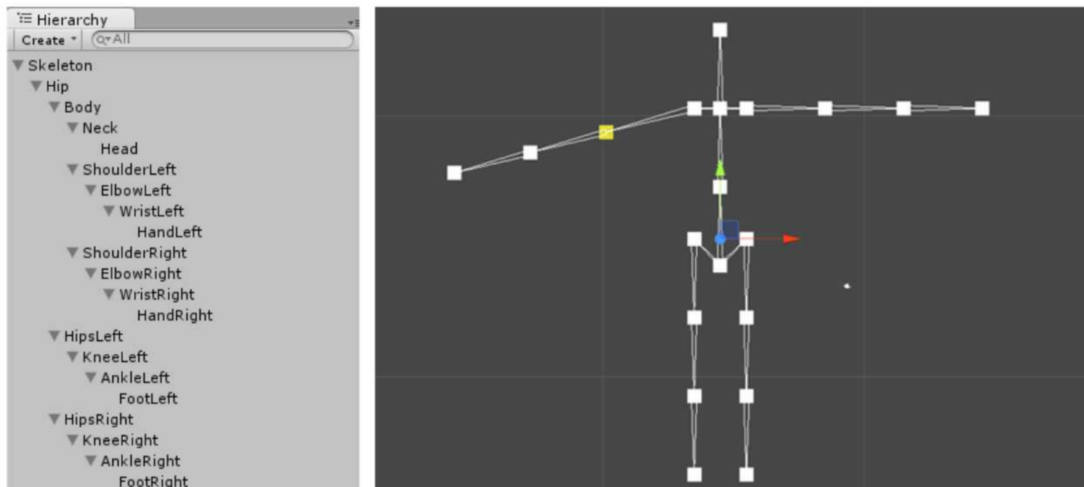

Figure 4.17 The skeleton 'template' created from SkeletonControllerEditor. The number of skeleton points refers to the number of bones from the Kinect-plugin. Handles will be built at every skeleton point for direct rotation editing.

The OnSceneGUI function from the BoneControllerEditor and the SkeletonControllerEditor differs in the number of bones: OnSceneGUI process all bones in skeleton, while the BoneControllerEditor processes only the bones in its children. The SkeletonControllerEditor class has a static function CreateSkeleton that automatically creates a “template” skeleton like in figure 4.17 right. This “template” consists of a skeleton GameObject with a *SkeletonController* attached and bone GameObjects with a *BoneController* attached[16] (figure 4.17 left). This function will also automatically arrange GameObjects in a tree structure. This

[16] So that a developer doesn’t have to create GameObjects, attaching scripts to them and arrange the tree structure manually

function can be called through the Unity menu when initial setup of the interactive application.

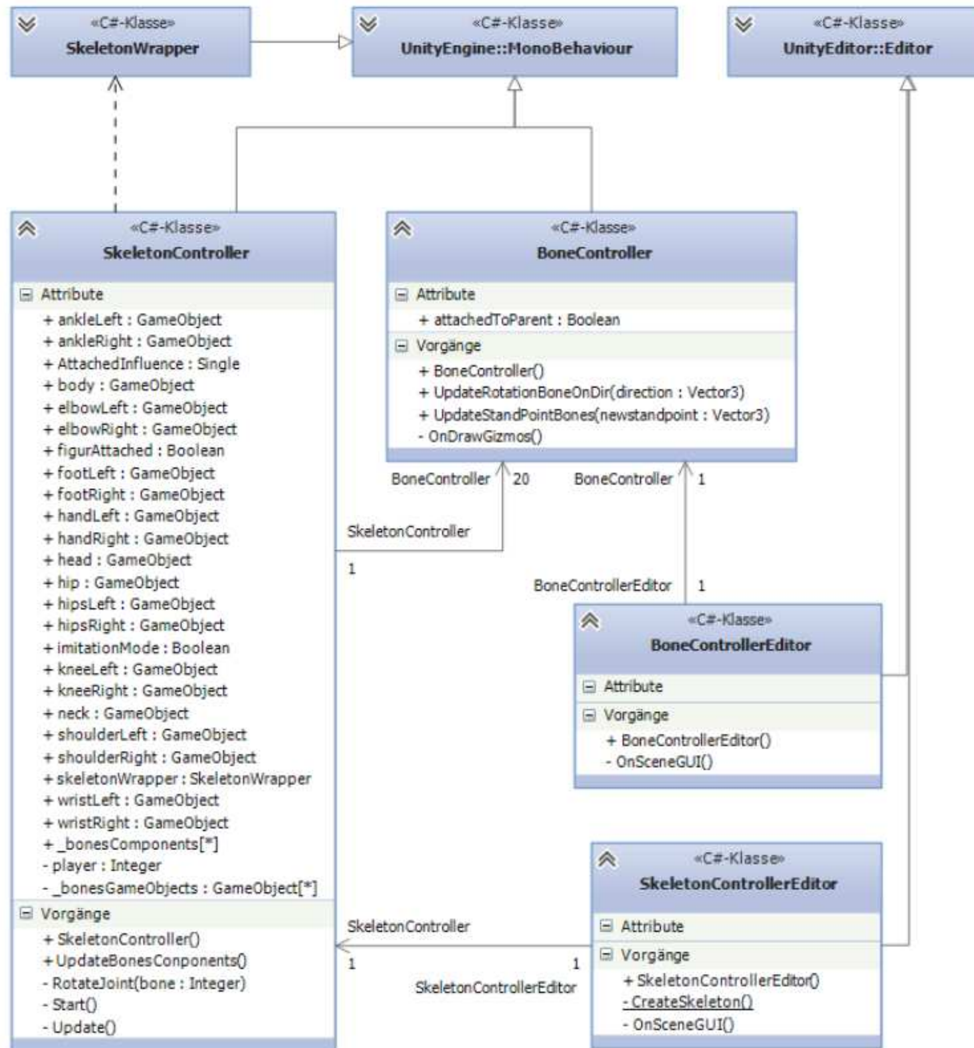

Figure 4.18 Class-diagram of the CharacterController component.

PolygonController

The main functionality of *PolygonController* is to create a mesh that will be attached to the bone GameObject from *SkeletonController*. Attaching a mesh to the skeleton will require the *SkinnedMeshRenderer* component (refer to Section 4.4.1, Creating Mesh in Unity). At the end of this process the character skeleton (GameObject with SkeletonController attached to it) will have the *SkinnedMeshRenderer* component attached to it, which acts as a connector between bones and mesh. The *PolygonController* does not have any class that is derived from *MonoBehaviour*. This means that these classes of this component will only be used during Edit-mode.

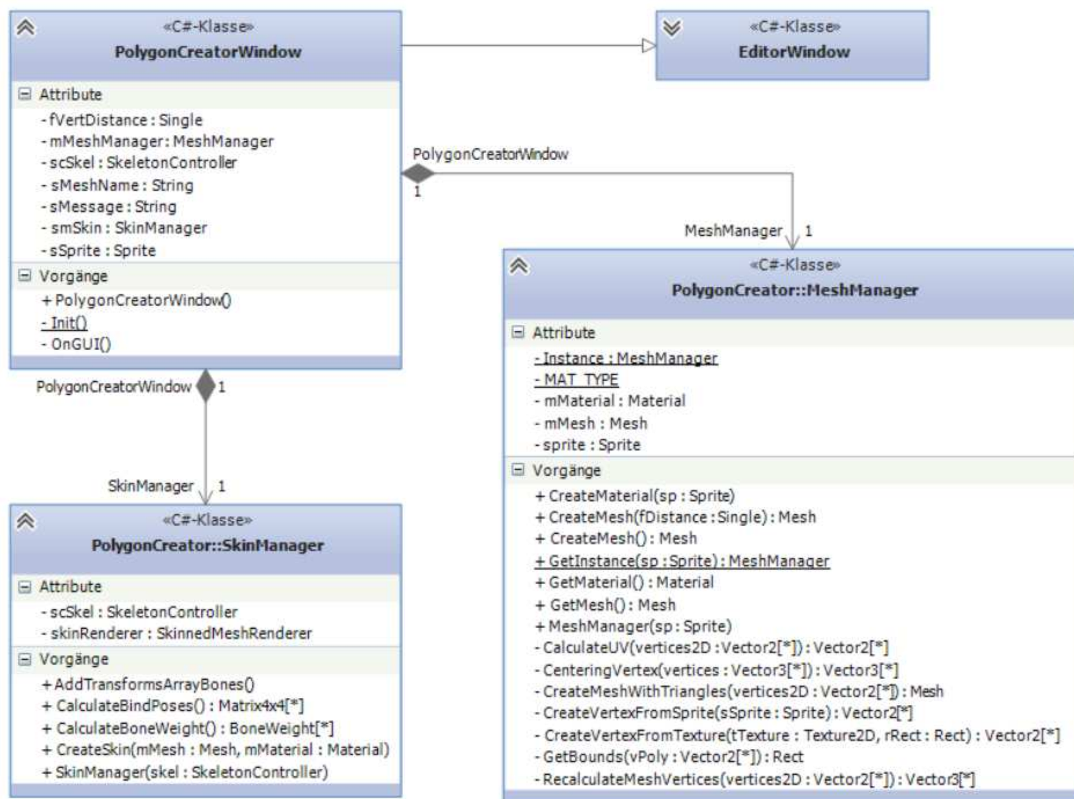

Figure 4.19 Class-Diagram of Component PolygonController

The *PolygonController* functionality is divided into two main classes: *MeshManager* and *SkinManager* (Figure 4.19). *MeshManager's* main functionality is to create mesh and material based on a graphic input. Allowed graphic input is graphic format with an alpha channel (for example *.PNG) and configured as type 'Sprite' in Unity (ref Appendix 6.2.2). *SkinManager* is there to prepare a mesh for *SkinnedMeshRenderr* and attaches it to the skeleton. An editor class connecting these functionalities is the *PolygonCreator* class, which is derived from the *EditorWindow* class. This window will provide a GUI for creating mesh, creating material and attaching a SkinnedMeshRenderer to a skeleton GameObject.

Figure 4.20 shows the GUI of *PolygonCreatorWindow*. This window will provides 4 input boxes: skeleton reference ("Skeleton"), texture from asset ("Texture (sprite)"), vertex distance ("Vertex distance") and mesh name ("Mesh Name"). Texture, mesh name and vertex distance information will be used to create mesh in MeshManager. Input Skeleton information will be used later together with the calculated mesh in SkinManager. This custom window also provide 4 buttons that implement functionalities from SkinManager and MeshManager.

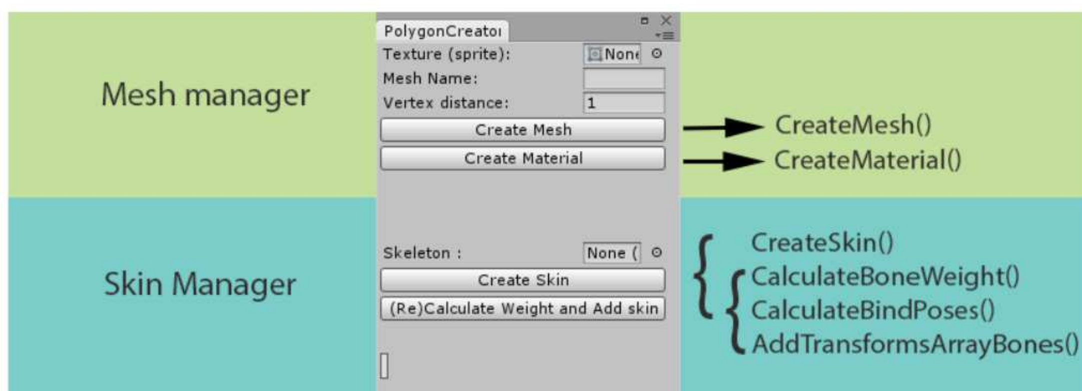

Figure 4.20 PolygonCreatorWindow takes 4 Input: Texture (Sprite), Mesh Name, Vertex Distance and SkeletonController. "Create Mesh" and "Create Material" button use functionality from MeshManager class. "Create Skin" and "(Re)Calculate Weight and Add Skin" uses functionality from SkinManager.

MeshManager

Mesh creation consist of 4 stages:

1. Calculate vertices

Calculation of vertices will use functionality from the *Physics2DDotNet* library [Port00].In order to create vertex from a texture, this library will check texture's alpha channel (transparency) and create a character's outline-vertices based on the transparency. Then the generated vertices will be simplified the using *Douglas-Peucker* algorithm [HeSn94] from the *Farseer Physics* library. This functionality is included within the *CreateVertexFromSprite* and the *CreateVertexFromTexture* function.

2. Calculate triangles

Creating triangles from vertices will use the functionality from the library *Poly2Tri*[Cont00]. The output from this function is a *Mesh*-class instance. *CreateMeshWithTriangles* of class *MeshManager* will be called for creating triangles using *Delaunay* algorithm on the vertices from previous operations.

3. Calculate UV Map

The UV map will be calculated on the basis of texture. First, the mesh width and height will be calculated by obtaining the leftmost, rightmost, topmost, and bottommost vertices. Second, the texture width and height will be accessed. Finally, with information about the size of the texture (in texture-define space) and the mesh (in vertex-define space), we can calculate the UV map for each vertex by comparing every vertex position in vertex-defined space to the position in the texture-defined space.

The result of this process is an array of *Vector2*. The index of this array refers to the index of each vertex. This tells each vertex which point in the texture they belong. Calculating the UV map occurs in the function *CalculateUV* from *MeshManager*

4. Optimizing mesh

Lastly, the position of the mesh will be optimized by re-scaling the calculated mesh. Because of the calculation of vertices using texture pixels, the generated vertices represent the pixel coordinate of texture. For example, if the graphic input format is 1080x780 wide, then the generated vertices will have the same width.

If those vertices are applied to the world space coordinate system in Unity, the mesh will become too big for the world space (scene). Therefore, re-scaling the texture size in world space is needed. After rescaling, the mesh will be repositioned in the center

of its local coordinate system so that it is easier to align skeleton bones to the mesh later. These functionalities are included in the *RecalculateMeshVertices* function.

Another function of *MeshManager* is the creation of a *Material* asset that contained within function *CreateMaterial*. This function instantiate a new *Material* instance and attaches texture to it.

PolygonCreatorWindow provides two buttons for MeshManager functionalities: “Create Mesh,” which uses the *CreateMesh* function; and “Create Material,” which calls the *CreateMaterial* function.

SkinManager

After saving mesh and material, the developer should be able to add mesh to the skeleton. This functionality is included in the class *SkinManager*. First, the component *SkinnedMeshRenderer* will be added to the skeleton GameObject. Afterwards, *bind poses* and *vertex weight* of the mesh will be calculated. Lastly, the mesh along with the material from *MeshManager* will be added to *SkinnedMeshRenderer*. These functionalities will be included in the *CreateSkin* function.

As mentioned in Section 4.4.1 (Creating Mesh in Unity), *SkinnedMeshRenderer* also requires more calculation: calculating vertex weight, binding pose, and array of bones:

1. Calculation of vertex weight needs information from mesh vertices and bones from SkeletonController. For every vertex in the mesh the nearest bone will be searched and assigned with full influence value (1.0). Calculating vertex weight is included in the function *CalculateBoneWeight* of the class *Skinmanager*.

2. Calculation of the binding pose in Unity is basically creating the Matrix4x4 transformation array for every bone (Section 4.4.1). This functionality is contained within the function *CalculateBindPoses.*

3. Another calculation for SkinnedMeshRenderer is the calculation of an array of bones (section 4.4.1). This will be included in the *AddTransformsArrayBones* function. In this function, an array of the *Transform* component from the bone GameObject will be created and added into the *bones* variable of *SkinnedMeshRenderer*

CreateSkin, CalculateBoneWeight, *CalculateBindPoses,* and *AddTransformsArrayBones* functions will be called within the OnGUI function of *PolygonCreatorWindow.* Calling those functions in one process will result in a mesh following the skeleton movement.

Skelton to mesh mapping cannot be completely automated: the “template” skeleton will not always have the same pose as the generated mesh. Figure 4.21, on the left side, is an example where the generated mesh does not match the “template” skeleton. This is where the handles from CharacterController are used. The developer should manually arrange the bones of the character shape by dragging the handles. This will be done within the scene view.

After finished manually aligning the skeleton bone to its position in the shape, the full process of adding skin can be called and the shape will follow the skeleton movement.

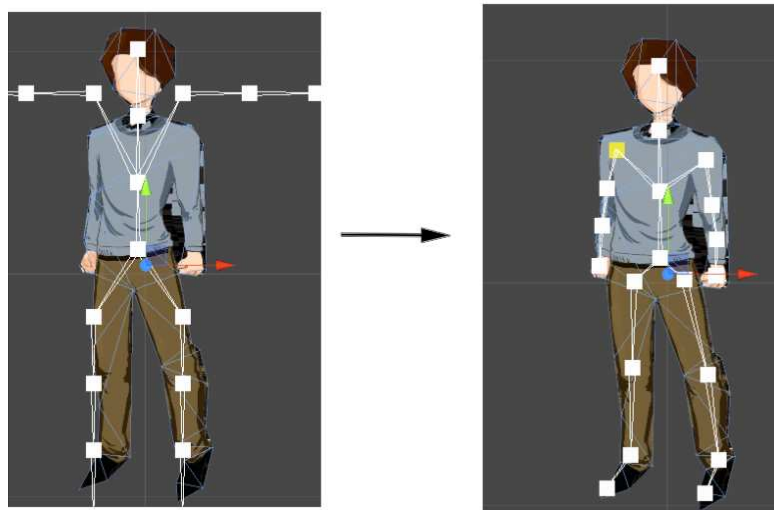

Figure 4.21 The developer needs to align the skeleton positions manually to the right position of the character shape (“Add Skin”- button)

Gesture

The Gesture component, together with the Kinect-plugin and *DtwGestureRecognizer* library, provides the interaction functionality. Interaction functionalities include recording a gesture, creating animation clips, connecting between animation and gesture, and recognizing gestures for triggering animation. As already mentioned in section 4.4.3, recording and recognizing gesture functionalities have already been implemented by help of the library *DtwGestureRecognizer*. Creating animation is done within the Unity animation system *Mecanim*. The system still needs to create a connector between the libraries (*DtwGestureRecognizer and Kinect*) and Unity. Moreover, the functionality for connecting animation to the gesture needs to be implemented.

The Gesture component functionalities are divided into three main classes: *GestureRecorder* for recording functionalities, *GestureControllerEditorWindow* for connecting gestures to animations, and lastly *GestureController* that makes sure that animation is triggered when a gesture is recognized. *GestureRecorder* creates a gesture template, and this template will be connected to the animation clip parameter through the editor window from *GestureControllerEditorWindow*. After mapping each gesture to the animation, the window will add the *GestureController* instance, along with the mapping information to the skeleton GameObject. The *GestureController* instance will make sure that an animation is triggered when the gesture template is recognized while in Play-mode.

GestureController and *GestureRecorder* classes derive from the *MonoBehaviour* class (Figure 4.22 and 4.23). Both classes use functionality from the Kinect-plugin in order to get the Kinect skeleton position while in Play-mode. The *GestureControllerEditorWindow* class is derived from *EditorWindow* (Figure 4.23).

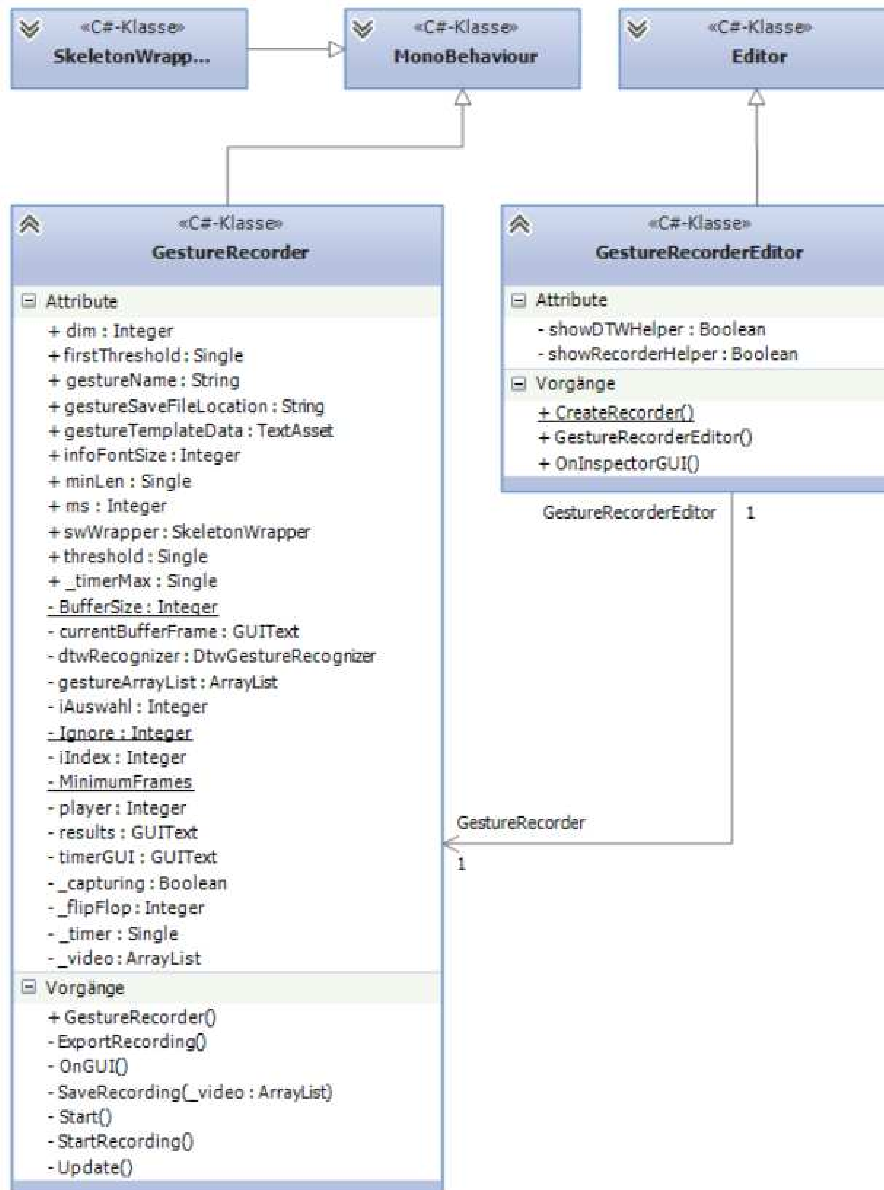

Figure 4.22 *Gesture* Component class diagram 1

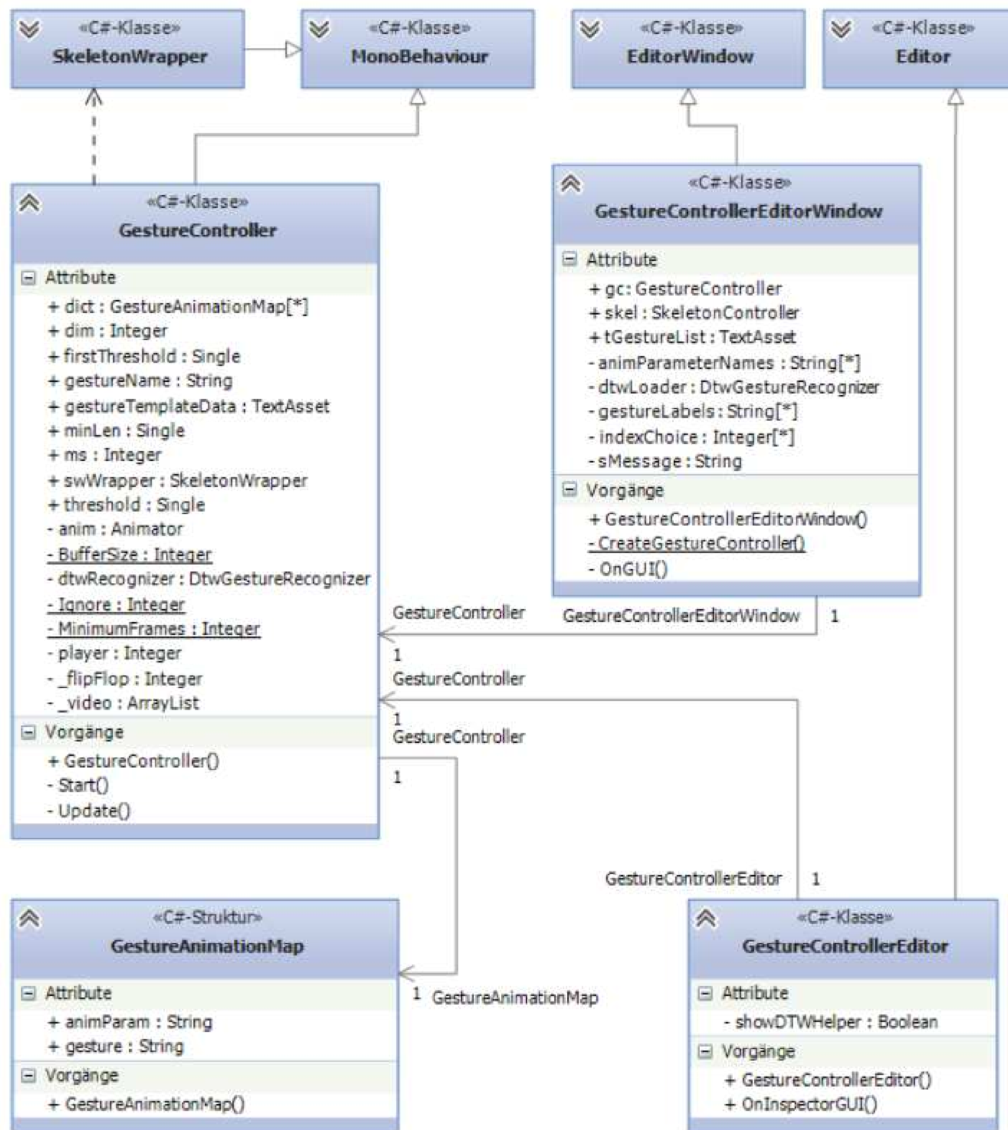

Figure 4.23 gesture component class diagram 2

GestureRecorder for recording functionality

GestureRecorder's main functionality is located in the Start and Update methods: in the Start function, the *DtwGestureRecognizer* instance will be initialized together with other helper variables. In the Update function, an *ArrayList _video* is always updated for every frame. This list holds a sequence of skeleton point positions from Kinect (Figure 4.22). For every new Kinect's upper-body skeleton position, they will be added into variable *_video* for every frame. If the *_video* contains frames more than *BufferSize,* then the first frame in *_video* will be removed so that in every frame *_video* always has the last skeleton position for the last *BufferSize* frame.

Recording Gesture

The first time a *DtwGestureRecognizer* instance initializes, it does not hold any template gesture. In order to create a template gesture, the function *AddOrUpdate* from *DtwGestureRecognizer* will be used. This function uses inputs from variable *_video.* The position sequence from *_video* will be transformed into the template gesture of the *DtwGestureRecognizer* instance in the *AddOrUpdate* function. The use of *AddOrUpdate* also requires an input gesture label (name). The template gesture, along with its unique name, will be saved in the *DtwGestureRecognizer* instance during runtime. *AddOrUpdate* will be called through the *SaveRecording* function of *GestureRecorder.*

Recognizing Gesture

Recognizing a gesture happens through *DtwGestureRecognizer's* function *Recognize*. Recognize will **always** be called in the Update function for every frame so that it always compares the last Kinect's position sequences with the saved template gestures. The output from this function is *string* label of the detected gesture based on the algorithm.

This function takes the current *_video ArrayList* and compares it with saved gestures with the help of the dynamic-time-warping algorithm. If Kinect's position sequences matches to any of the template gesture it will give the gesture label (string) as output.

Exporting Gesture-list

Saving gesture through *AddOrUpdate* function of *DtwGestureRecognizer* is only during runtime. That means if Unity exits Play-mode, the saved template gesture will be gone. There is a need for having functionality to exporting saved gestures from the *DtwGestureRecognizer* instance. This functionality is obtained through *ExportRecording* function. This function will

call the *DtwGestureRecognizer's* function *RetrieveText RetrieveText* that collects the saved gesture of the instance and exports it as *.txt format.

Loading a Gesture-list

Having a way to export gesture data, *GestureRecorder* also needs a functionality to load the exported data. This functionality is implemented through the static function *LoadGestureFromAsset* of the class *GestureUtillities*. In order to load a template gesture data, this function needs a *DtwGestureRecognizer* instance. The function *LoadGestureFromAsset* will be called after initializing the *DtwGestureRecognizer* instance in the Start function of *GestureRecorder*. Inside this function, the template gesture data will be transformed into gesture templates of the *DtwGestureRecognizer* instance by using the *AddOrUpdate* function.

Recording GUI

Recording functionalities, such as *SaveRecording* and *ExportRecording,* will be available in the Update function, but it will not be called actively on every frame (like *Recognize* function). To call those functions, the *GestureRecorder* instance provides a GUI Menu. This GUI menu will be created through the *OnGUI* function of *GestureRecorder,* and will provide menus and other interface for recording a gesture.

Figure 4.24 shows the *GestureRecorder* instance in Play-mode. The *OnGUI* function provides buttons for recording functionalities. The "(Update) Record" button will be used for calling the *SaveRecording* function

"Export Gesture" button will call the *ExportRecording* function. Once a gesture is recorded, it will be shown as a list at the Interface (Figure 4.24 left below).

In addition, the *OnGUI* of the class will be used to create GUI Text during runtime. The GUI Text gives real-time information about the current output from *DtwGestureRecognizer* instance's *Recognize* function. Figure 4.24 shows Unity in Play-mode and provides information that it is currently recognizing the "@handwave"-gesture. The GUI Text also shows how much frames the *_video* variable currently possesses. For more insight into how the GUI works, please refer to the Appendix section.

Figure 4.24 GUI from GestureController for recording gesture

Custom Inspector view

GestureRecorderEditor is a class, which is derived from the *Editor* class. This class is used for creating the custom inspector for the *GestureRecorder* Instance (through *OnInspectorGUI*). The custom inspector view will show recording properties (Figure 4.25). Additionally, this *Editor* class also has the static function *CreateRecorder*. This function will automatically create a new empty GameObject and attach *GestureRecorder* to it.

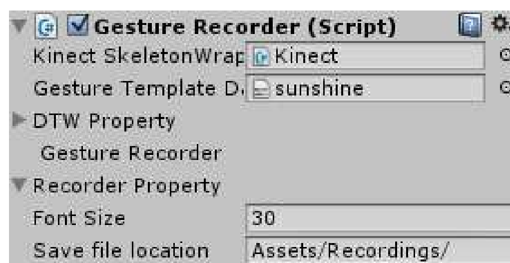

Figure 4.25 Inspector view from *GestureRecorder*

GestureControllerEditorWindow for connecting Animation to Gesture

GestureControllerEditorWindow creates a window for connecting gestures with animation parameters. This window needs template gesture data and the skeleton GameObject from *CharacterController* as its input (Figure 4.26 left). Template gesture data is the result of *GestureRecorder*. Gesture labels will be extracted from this data through the static function *LoadGestureLabelFromAsset* of *GestureUtillities*

The second input is the skeleton from *CharacterController*. The skeleton is used within the window to access its *Animator* component. The animator parameters will be loaded within the *OnGUI* function of the window. For this thesis we will only use a certain animator parameter type "**Trigger**."

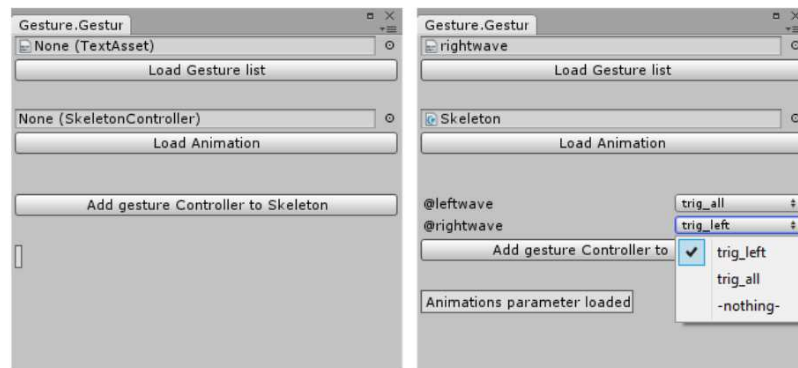

Figure 4.26 The *GestureControllerEditorWindow* window (left) before the gesture list and animator are loaded; (right) after loading the gesture list and animation parameter, the developer can attach them together

After loading the gesture list and animation parameters, gesture labels will be shown as a list in the window and animator parameters as a drop down menu for every gesture label (Figure 4.26 right side). The connection between the animation parameter and the gesture label will be stored in an array of Struct-type *GestureAnimationMap*. Every *GestureAnimationMap* object in this array has always one gesture name and one animation parameter name. At the end of the *OnGUI* function from the window, there will be the button "Add Gesture Controller to Skeleton". This button is used to create a *GestureController* instance, add the *GestureAnimationMap*-array and template gesture data to its member variables and to attach it to the skeleton GameObject.

GestureController for triggering animation

GestureController is a MonoBehaviour class that is attached to the skeleton GameObject and operates within Play-mode. Its main functionality is to trigger character animation during the Play-mode runtime. It has similar functionalities like *GestureRecorder*: in the Start function, the *DtwGestureRecognizer* instance will be initialized and the template gesture data will be loaded. In the Update function, an *ArrayList _video* is updated for every frame and the *Recognize* function of *DtwGestureRecognizer* instance will be called. The difference from *GestureRecorder* is that every time the instance recognizes a gesture, it will look up the information in the *GestureAnimationMap*-array and trigger the matching Animator parameter.

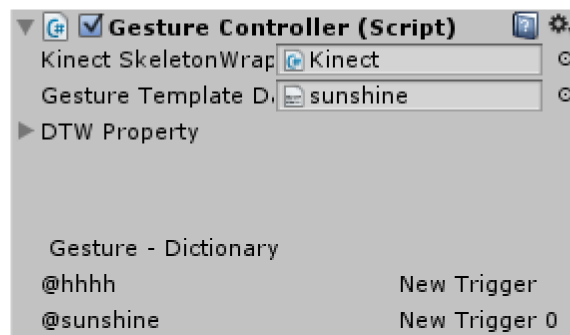

Figure 4.27 *GestureController's* inspector views

Custom Inspector view

The last class from this component is called *GestureControllerEditor*. This class is responsible for creating the custom inspector view for *GestureController* (figure 4.23). The inspector view from *GestureController* shows gesture-animation parameter connections (Figure 4.27).

4.5.3 Implementation Environment

For implementation in this thesis we will use C# as a programming language. The development environment of the system uses window 8.1 with Unity version 4.5.4.f1. As IDE is MonoDevelop 2.0 from Unity used with the target framework .Net Framework 3.5. Kinect hardware used in this thesis is Kinect for Windows with its driver Kinect for Windows SDK v1.8. Table 4.1 shows the specification of the computer used for developing the system

CPU	AMD A10-5700 APU
Processor Core	4 * 3,4 GHz
OS	Windows 8.1 Pro N 64Bit
HDD	464 GB
RAM	6,00 GB
Graphics Card	AMD Radeon HD 7400

Table 4.1 : Specification detail of the computer used to develop and test the system.

4.6 Conclusion

In this chapter, the solution for this thesis has been shown. In chapter 4.1 and 4.2 the general requirement have been described. Section 4.3 outlines the main components required for creating system solutions. Section 4.4 of this chapter introduced the libraries and the tool that support the solution. This section identifies the functionalities from Unity and at the same time also overviews necessary development and external libraries to fulfill the solution requirement.

Section 4.5 showed the implementation of requirements in section 4.1 and 4.2. Based on the components from section 4.3, this section describes complete implementation of the system environment in details.

5 Conclusion

This Thesis aims to develop an environment for creating an interactive application that combines that combines gesture recognition and human character animation. The functionality of this thesis includes: **character modeling**, **imitation, and Interaction with a virtual character**. The target user of the proposed system is not limited to developers, but also designer and other users who do not have any programming knowledge. In order to achieve this, the solution in this thesis provides tools so it lightens the burden of developing interactive application. Instead of creating the environment from a scratch, this thesis has decide to utilize game-engine Unity. Unity already provides many functionalities for creating the interactive application and covers most functional requirements. Other than using the tool Unity, this thesis also integrates the developed system with other libraries and extends Unity functionalities in such a manner that they fulfill the functional requirements described in the chapter Introduction.

Chapter 2 examines related work and their relation to their developed system. This chapter shows which aspect of these project can be adapted in the developed system.

Chapter 3 the theoretical background used in this thesis has been discussed. This chapter is divided into three main parts: first, introduction to the key-frame animation. Second, the basic of character modelling and animation. This part also addresses the issues of creating character shapes that use of *Skinning*. Third part discussed the motion capture system. Since the developed environment requires imitation and gesture recognition, a motion capture component is also required in the system. This chapter is concluded with decision on the character modelling/animation methods and motion capture technique.

Chapter 4 concentrates on the design and implementation of the environment. First, the functional requirements and non-functional requirements are described in detail. Afterwards this chapter identified the key components to the section and discussed the functionalities provided by Unity and also additional functionalities which have to be develop by mean of scripting and integrating external libraries. This section also introduced the programming model of Unity. This chapter concludes with detailed description of the design and overall solution.

To summarized, the main aim of this thesis is to develop an environment for creating an interactive application has been reached. The main functionalities for character modeling, imitation, and interaction have been covered within the system. All those functionalities—from creating character to creating imitating movement and interactive character—can be achieved without having to write a code.

Modification and extending functionalities of the system are still possible. First, the creation of a shape can be done automatically: in the current solution, the developer has to manually align skeleton points to the right position of the shape. This process could be automated in which the system analyzes the body parts of the calculated shapes and positions of the skeleton points accordingly. Second, the movement of the character while interacting or imitating could be more realistic. The current solution uses maximal influence for every vertex on bones when calculating vertex weight (1.0). This could be improve by applying skinning algorithm compatible with environment's data model. Physic simulation, such as gravity and force, can also be applied to the skin for creating realistic character animation. Third, modeling and animating characters in this thesis are limited to a 2D mesh model because the visual of a character is obtained from a texture. If there is a need to create a 3D character model, which also interacts / imitates in the 3D world (like in [ShGh14]), then implementation of bone rotation should be extended into the 3D rotation. Finally, since the developed system uses Kinect and also has imitation requirements, it is also possible to create animation clips in Unity within the *CharacterController* component. Recording movements during the imitation mode will require extended implementation on the *SkeletonController* class. If this functionality is implemented, then creating animation clips will be easier, and Mecanim in Unity will be used only for creating transition between animation clips.

6 Appendix

This section describes the process of creating an interactive application using the developed environment. This section will also be divided into four sub-sections: configuring Unity, character modelling, imitation and interaction

6.1 Configuring Unity

In order for a developer to use the developed system, some configurations of Unity is needed. First, developer should make sure that Unity is in 2D-Mode. This can be achieved from Unity-editor (figure 5.1 left). Afterwards, the developer can start importing extended Unity package and start developing an application (figure 5.1. right).

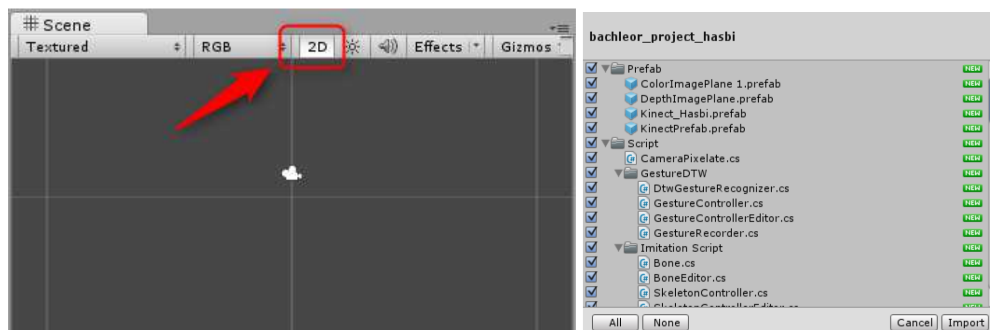

Figure 6.1 configuration before using the Unity

6.2 Character modeling

6.2.1 Creating Skeleton

Modelling a character starts with creating a skeleton “template” from the menu. If the package is imported correctly, developer should see the menu “Skeleton” in Unity menu items (figure 6.2 a). Other than create skeleton GameObject (and its bone GameObject), selecting this menu item will also add Kinect GameObject to the scene and assign it to the *skeletonWrapper* variable of the *SkeletonController* component (figure 6.2 b and 6.2 d).

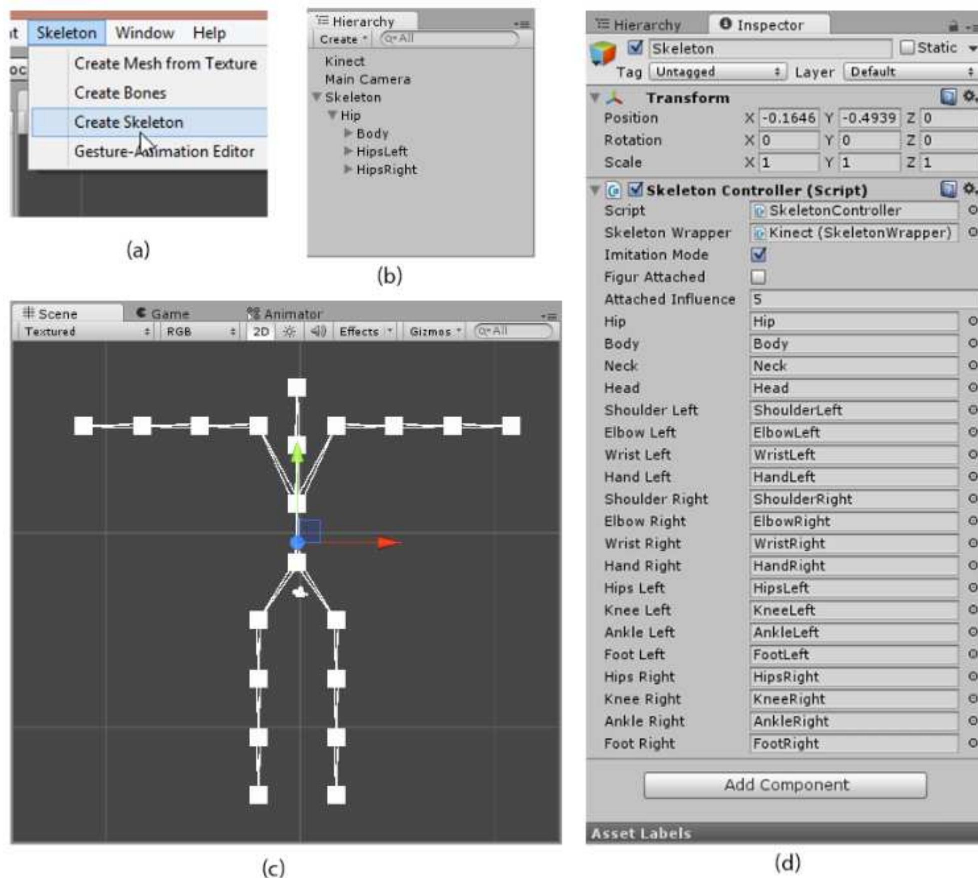

Figure 6.2 (a) Creating skeleton menu. (b) Hierachy view after adding skeleton. (c) Skeleton “template” in scene view. (d) Skeleton inspector view

In the scene view, the skeleton GameObject will draw the handles at each skeleton. Developer can drag the handles and modify bone rotation (Figure 6.3).

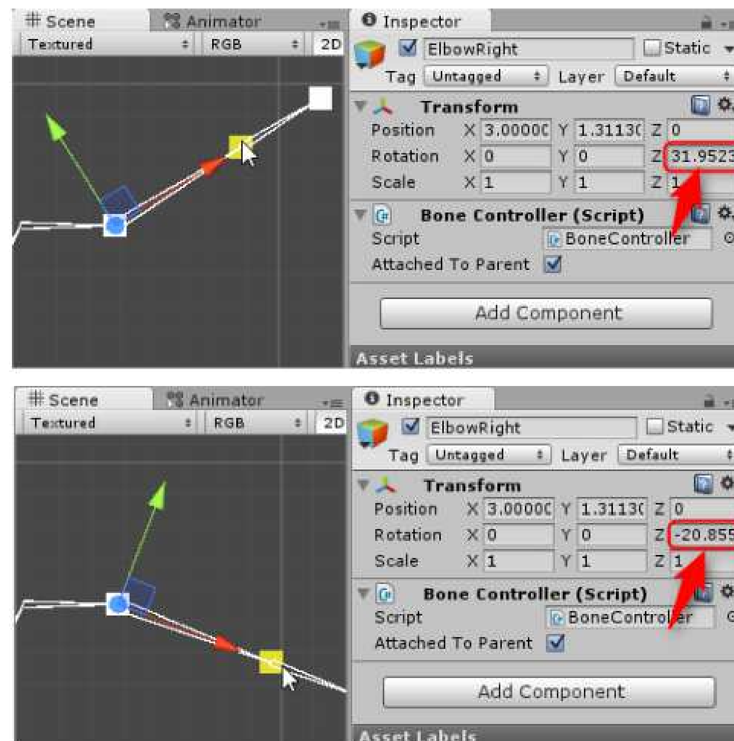

Figure 6.3 changing skeleton rotation with handles.

6.2.2 Creating Shape

When finish creating skeleton, the developer can starts creating a shape for the skeleton. A mesh will be created through an input texture. That's why developer need to import the texture first. The imported texture should configured as type 'Sprite' (figure 6.4). This could be change from the texture's inspector view (figure 6.4).

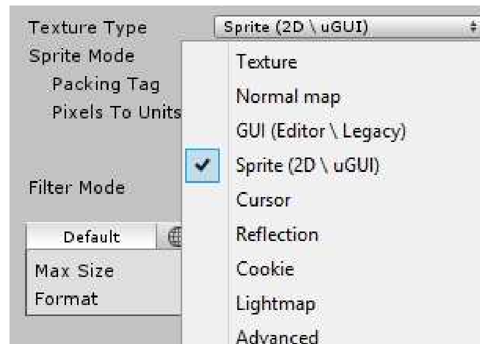

Figure 6.4 Texture type should be "Sprite"

After finish importing the texture, the developer can start calculating mesh and adding the calculated mesh to the skeleton using customized window. This windows will be created through menu “Create Mesh from Texture”. Figure 6.5 shows menu item and window GUI for creating shape.

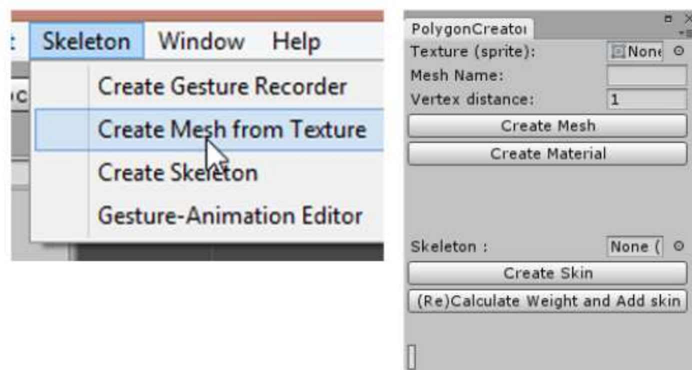

Figure 6.5 Menu for creating shape and its window GUI

Creating mesh and material in this window will result in mesh and material being saved in Unity asset folder (figure 6.6)

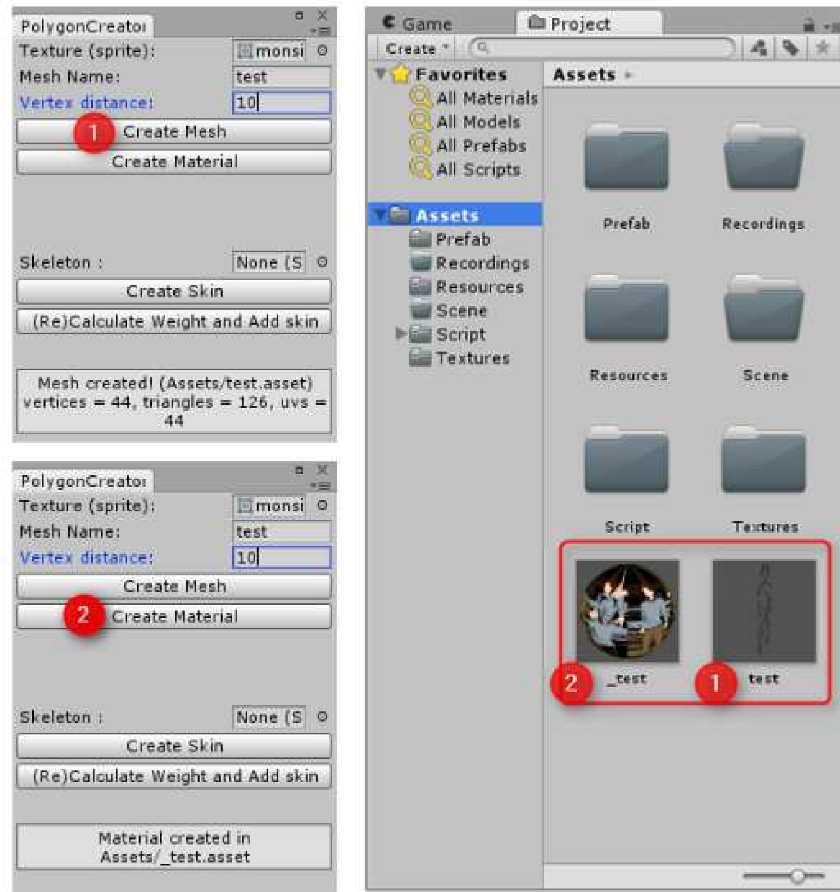

Figure 6.6 Creating Mesh and Creating Material

The process of adding mesh to the skeleton will need skeleton to be assigned to the input box first. After assigning the skeleton, the developer can create skin (figure 6.7-1). Figure 6.7-2 to figure 6.7-3 shows the process where developer manually align the skeleton joint position into the right mesh position in the scene view. When finished, the developer can adds skin to skeleton by pressing the second button "(Re) calculate Weight and Add Skin" (Figure 6.7-4).

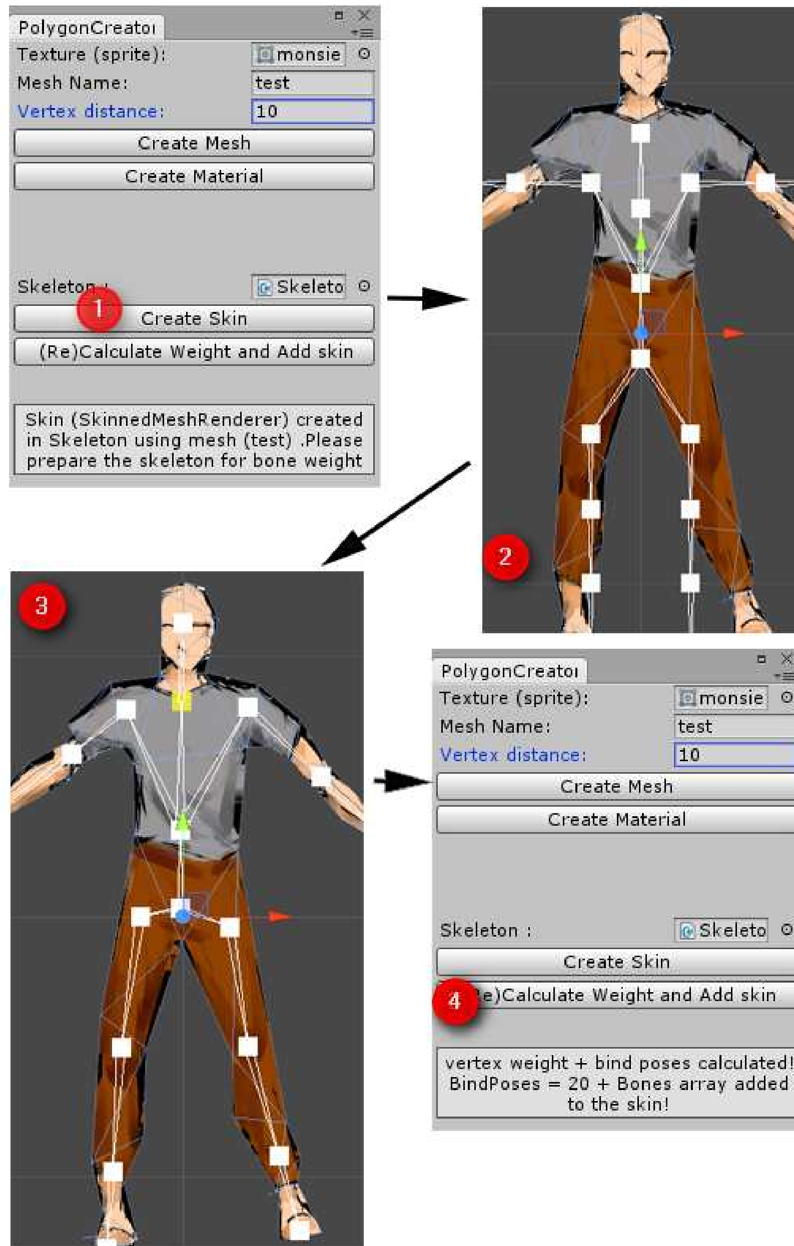

Figure 6.7 Adding skin to skeleton

After the process of adding skin into skeleton, the mesh will follow the skeleton movement. This could be tested by dragging the skeleton handles in Edit-mode (figure 5.8)

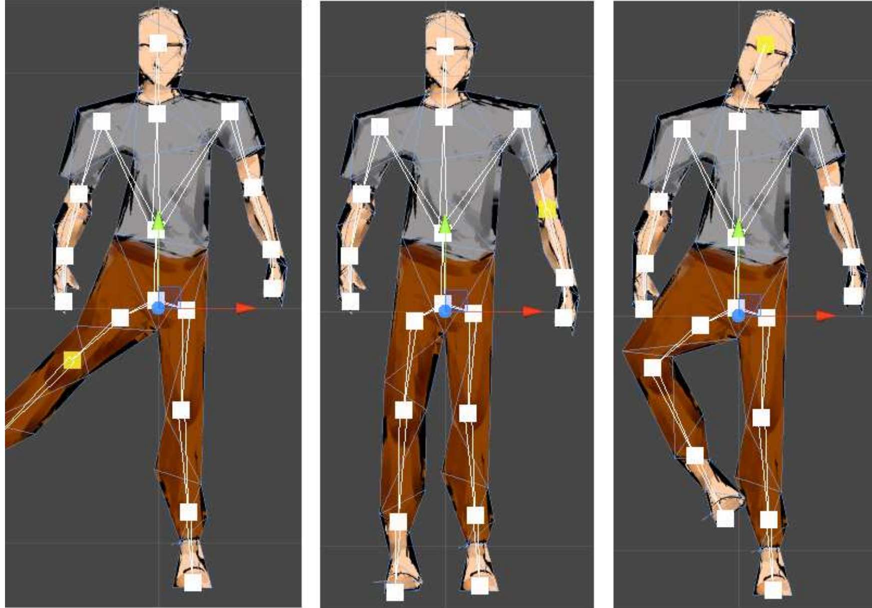

Figure 6.8 Testing the mesh deformation

6.3 Imitation

The skeleton GameObject created in section 6.2.1 already have functionality for imitation: the *SkeletonController* script will take input from Kinect plugin and transforms the movement from the Kinect skeleton position into bone rotation of the character. Once the skin is added, the application is ready for imitating participant movement in Play-mode

Before testing the application for imitation functionality, the developer needs to make sure that the variable *skeletonWrapper* is assigned to the Kinect GameObject and *imitation Mode* variable is checked (figure 6.9).

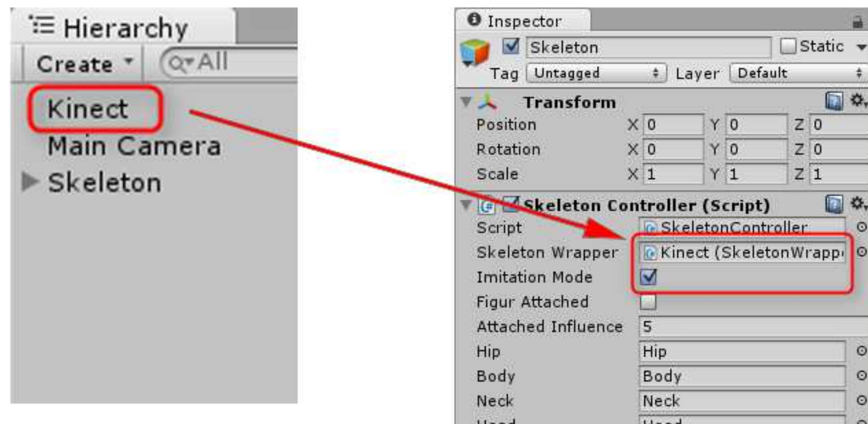

Figure 6.9 variable *SkeletonWrapper* needs to be assinged to the Kinect GameObject and Imitation Mode should be chekced

6.4 Interactive

Unlike imitation, this functionality needs extended configuration in order for application to have an interactive character: recording/creating template gesture, creating animation, connecting animation with gesture, testing interactivity.

6.4.1 Creating template gesture

Creating a template gesture is achieved by creating a recorder GameObject, which will be created automatically from the menu “Create Gesture Recorder”. This menu will add Kinect GameObject automatically to the Component, so developer doesn’t need to assign it manually. (figure 6.10)

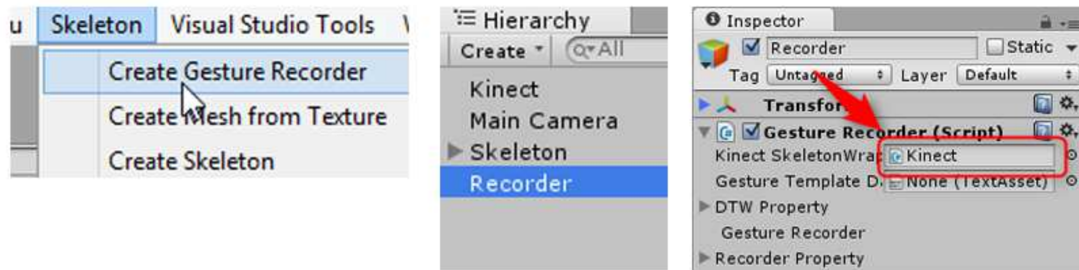

Figure 6.10 from left to right: menu for creating gesture recorder; Gesture recorder GameObject in *hierachy view*; *Inspector view* of Gesture-recorder GameObject (Kinect GameObject is assigned automatically)

After creating *recorder GameObject*, developer can start recording template gesture by bring Unity in Play-mode. The *GestureRecorder* instance in recorder GameObject will create GUI for recording functionality.

Figure 6.11 shows a gesture recorder GUI in Play-mode. At left upper side is main button for recording: "(Update) Record!"-button for start recording a new gesture. Saving gesture require inputs of the gesture name (Input box for gesture name). If a same name is already saved, then recorder will updated it instead of adding it. "Delete Recording"-button deletes any gesture that have same name like in gesture name input box. "Export recording" will export the saved template gestures. Like gesture recording, using this button also require input on a text-box beside the button for the name of the exported gesture data.

The details of the recording GUI is as follows: At the left below side is a list of a saved gesture, that shown as a button list. Developer can choose any gesture in this list and the name will be appear on the gesture input name. At the upper right and below right side will be space for GUI text for information. Information that being shown are: name of the recognized gesture, Buffer in _*video* variable and other information such as error message or timer.

Figure 6.11 GUI of gesture recorder

Recording a gesture occurs after developer inputs the gesture name and clicks “(Update) Record!”-button. Developer will be given 3 second for preparing before the actual recording occurs. Afterwards the buffer will be reset to zero and start saving gesture for a fixed number of buffer (default is *32* frames). At this times, the developer should create move in a looping movement (figure 6.12).

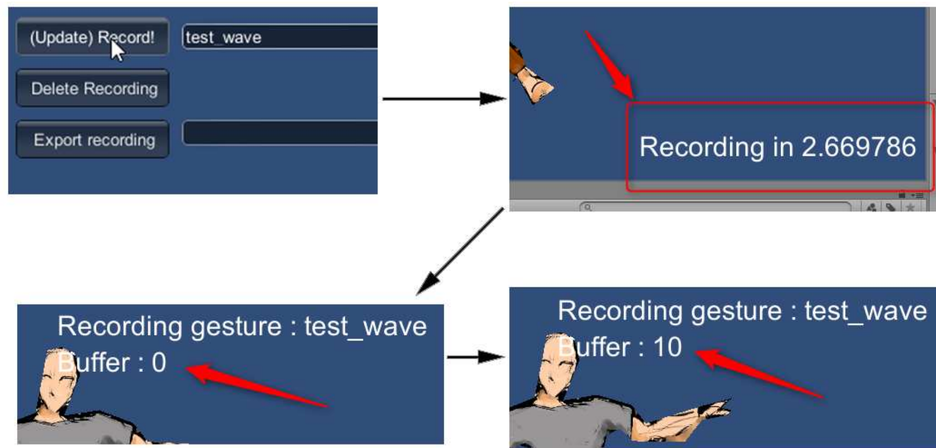

Figure 6.12 record a gesture

When finished, the gesture will be saved automatically in the list of saved gesture (figure 6.13-2). Recognizing gesture is always active so the developer can test the newly created gesture by repeating the movement.

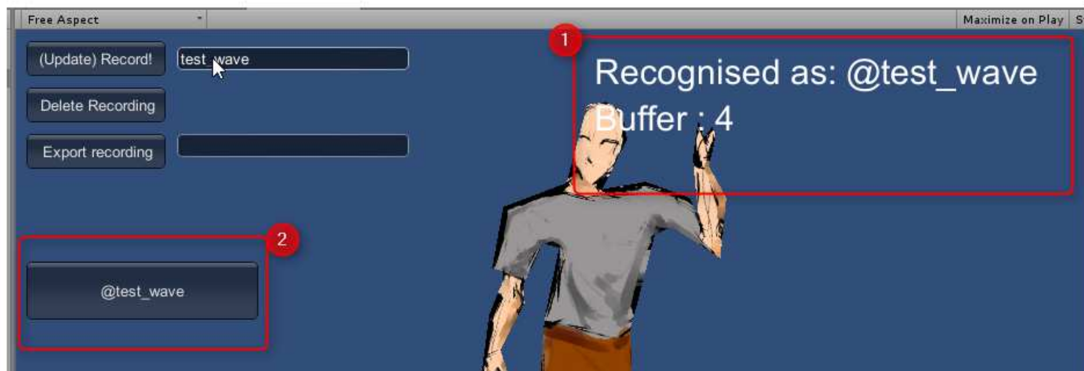

Figure 6.13 after finish recording a gesture

When finish creating / editing gestures, the gesture list must be exported (figure 6.14). Exported gesture data will be saved in defaults folder: Assets/Recordings/<name of data>.txt.

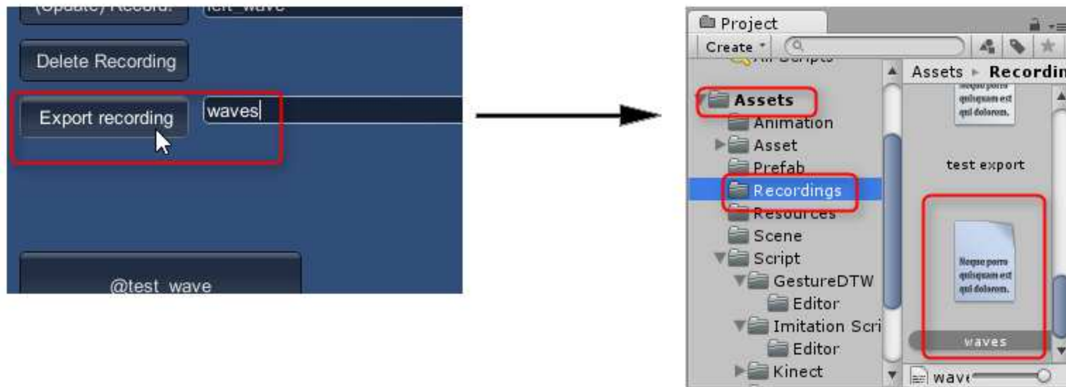

Figure 6.14 exporting a gesture data

6.4.2 Creating animation

After having gesture data exported, developer could start creating animation sequences as response. Using Mecanim (Animation system from Unity) and the skeleton GameObject of the character, the developer could start create animation right away.

Creating animation occurs in animation window. Using Mecanim, the developer can easily create an animation clip based on key frame system (figure 6.15) and create complex animation sequences within animator window (figure 6.16). For more information how Mecanim work please refer to [Unit00h]

Figure 6.15 shows an example of creating an animation clip. The developer should make sure that animation clip is created when skeleton GameObject is selected and open the animation window of Unity. (a) After creating an animation clip, developer can start recording an animation by bring animation window in record-state (b). (c) Afterwards, the property being animated can be chosen through “Add curve”. In this case rotation. (d) Developer creates key frame at t=0.00s. e) Developer creates other key frame at t=0.30s and iterates the process for creating other animation sequences.

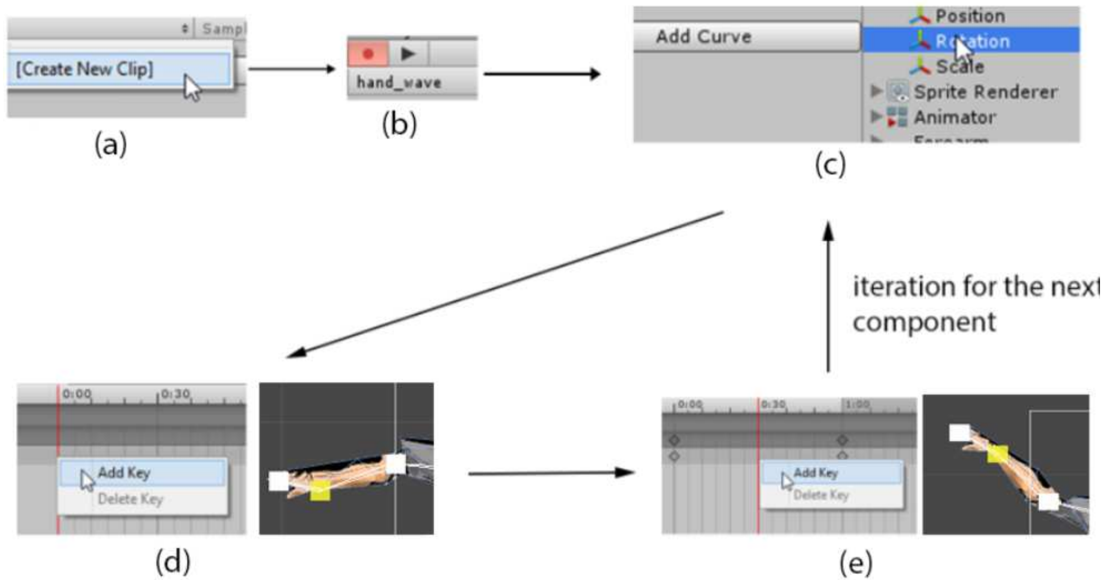

Figure 6.15 creating animation clips in Unity

After creating animation clips, developer could start creating transition between animation clips through animator window. This part is necessary because animator window is also needed for creating parameter for triggering an animation when gesture recognized later.

Figure 6.16 shows an example of configuring transition between animation clips: (a) animator window views animation-clips as state machine. The default-animation state is shown in color orange (that means this animation will be played first when in Play-mode). The other animation states will have grey color. A green state “Any State” is a built-in animator states that is useful for escaping any condition in state-machine. (b) shows creation of transition between animation clips. (c) Animator window after connecting between animation clips. (d) Creating parameter for transition conditions. This environment will only use **Trigger**-type parameters**.** (e) The inspector view when transition between states in selected. Here the developer can set the condition with parameter created from (d)

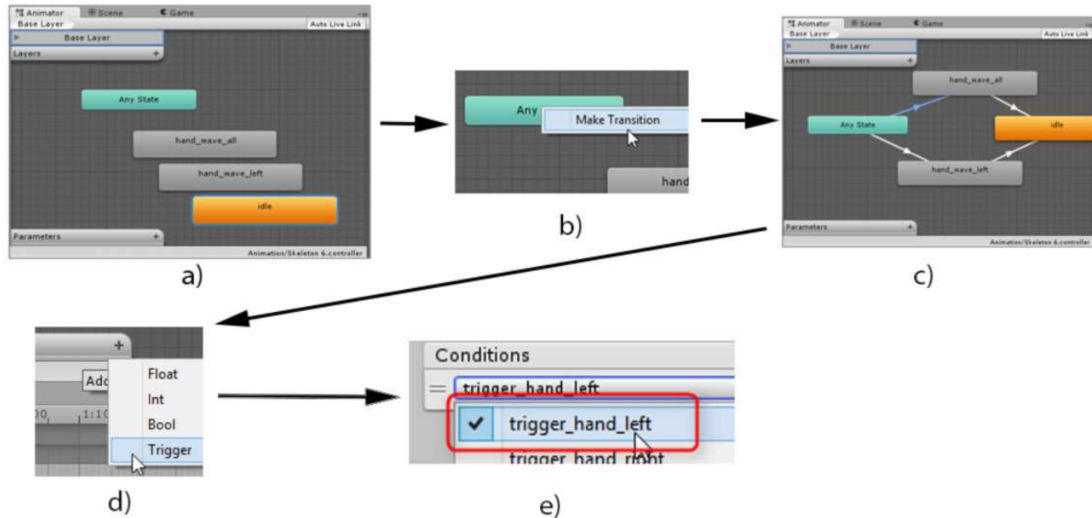

Figure 6.16 creating animation transitions in the *Animator window*

6.4.3 Attach gesture to animation

After creating gesture templates and configuring animator, the developer can start connecting gestures with animator parameters through gesture-animator window (figure 6.17). Mapping a gesture to an animator parameter occurs once gesture export data and animator (of skeleton GameObject) is loaded. Once they're loaded, developer can chose the animator parameter for every gesture template.

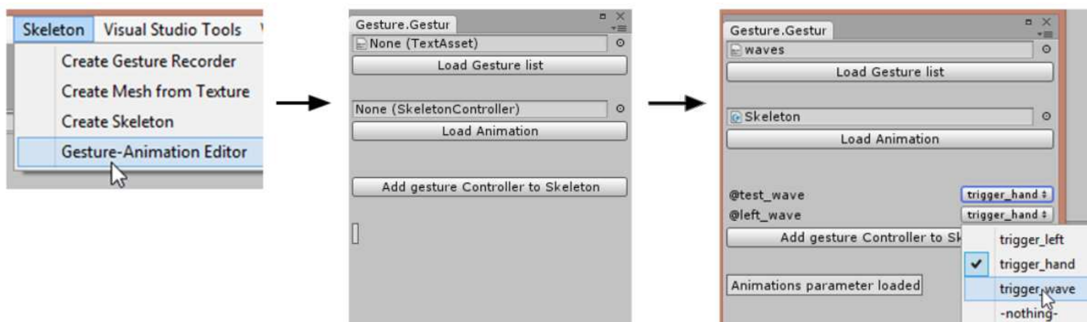

Figure 6.17 creating gesture-animation editor

Afterwards developer can click the last button on this window "Add gesture Controller to Skeleton" for adding *GestureController* instance to the skeleton GameObject.

6.4.4 Gesture recognition

When *GestureController* is added, the skeleton GameObject now has interactive functionality. Developer should uncheck the *imitation mode* variable from SkeletonController component before testing it. This way, *SkeletonController* won't changing skeleton position by imitating participant's movement.

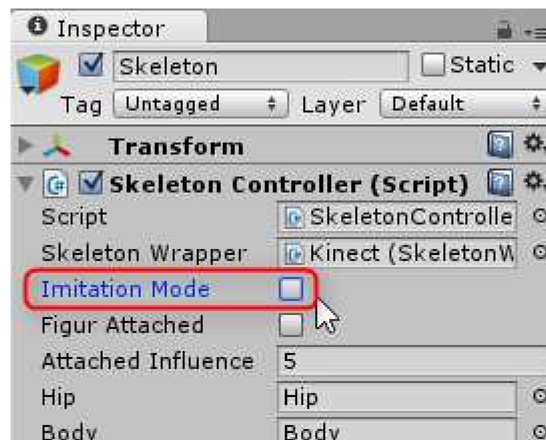

Figure 6.18 Imitation mode should be unchecked when using interactive functionality

Bibliography

[AkHH08] AKENINE-MÖLLER, TOMAS ; HAINES, ERIC ; HOFFMAN, NATY: *Real-Time Rendering*. vol. 85, 2008 — ISBN 1568814240

[BaSm79] BADLER, NORMAN I. ; SMOLIAR, STEPHEN W.: Digital Representations of Human Movement. In: *ACM Computing Surveys* vol. 11 (1979)

[Cont00] CONTRIBUTOR, POLY2TRI: *Poly2Tri*. URL https://github.com/JakeCataford/unity-tinkerbox/tree/master/Lib/Poly2Tri

[Crai89] CRAIG, JOHN: *Introduction to Robotics: Mechanics and Control (3rd Edition)*, 1989

[Desi11] DESIGN I/O: *PUPPET PARADE*. URL http://design-io.com/projects/PuppetParadeCinekid/. - abgerufen am 2014-12-19

[DoMB11] DODDS, TREVOR J. ; MOHLER, BETTY J. ; BÜLTHOFF, HEINRICH H.: Talk to the virtual hands: Self-animated avatars improve communication in head-mounted display virtual environments. In: *PLoS ONE* vol. 6 (2011) — ISBN 1932-6203 (Electronic)r1932-6203 (Linking)

[Gent00] GENTXWARPER: *GenTXWarper - Mining gene expression time series*. URL http://www.psb.ugent.be/cbd/papers/gentxwarper/DTWalgorithm.htm. - abgerufen am 2014-11-14

[HaZi05] HARTLEY, RICHARD ; ZISSERMAN, ANDREW: *Multiple View Geometry in Computer vision*. vol. 23. UK : CUP, Cambridge, UK, 2005 — ISBN 0521540518

[HeSn94] HERSHBERGER, JOHN ; SNOEYINK, JACK: An O(nlogn) implementation of the Douglas-Peucker algorithm for line simplification. In: *Proceedings of the tenth annual symposium on Computational geometry*, 1994 — ISBN 0-89791-648-4, pp. 383–384

[ILSS06] ISENBURG, MARTIN ; LIU, YUANXIN ; SHEWCHUK, JONATHAN ; SNOEYINK, JACK: Streaming computation of Delaunay triangulations. In: *ACM Transactions on Graphics* vol. 25 (2006) — ISBN 1595933646

[JaTw05] JAMES, DOUG L. ; TWIGG, CHRISTOPHER D.: Skinning mesh animations. In: *ACM Transactions on Graphics* vol. 24 (2005)

[Krue77] KRUEGER, MYRON W.: Responsive environments. In: *Proceedings of the June 13-16, 1977, national computer conference on - AFIPS '77* (1977), p. 423 — ISBN 0750605669

[KuZa06] KUCUK, SERDAR ; ZAFER, BINGUL: Robot Kinematics: Forward and Inverse Kinematics. In: *Industrial Robotics: Theory, Modelling and Control*, 2006 — ISBN 3-86611-285-8

[Lass87] LASSETER, JOHN: Principles of traditional animation applied to 3D computer animation. In: *ACM SIGGRAPH Computer Graphics* vol. 21 (1987) — ISBN 0897912276

[Macc00] MACCORMICK, JOHN: *How does the Kinect work?*. URL http://users.dickinson.edu/~jmac/selected-talks/kinect.pdf. - abgerufen am 2014-10-30

[MoGr01] MOESLUND, THOMAS B. ; GRANUM, ERIK: A Survey of Computer Vision-Based Human Motion Capture. In: *Computer Vision and Image Understanding* vol. 81 (2001), pp. 231–268 — ISBN 10773142 (ISSN)

[Nobl09] NOBLE, JOSHUA: *Programming Interactivity*. vol. 54, 2009 — ISBN 144931144X

[Owen99] OWEN, SCOTT: *Motion Capture: Acclaim's Optical system*. URL http://www.siggraph.org/education/materials/HyperGraph/animation/character_animation/motion_capture/motion_optical.htm. - abgerufen am 2014-10-20

[Pare08] PARENT, RICK: *Computer Animation: Algorithm and Techniques*, 2008 — ISBN 978-0-12-532000-9

[Port00] PORTER, JONATHAN MARK: *physics2d*. URL https://sites.google.com/site/physics2d/. - abgerufen am 2014-11-14

[Qvis00] QVIST, IAN: *Farseer Physics Engine - Home*. URL http://farseerphysics.codeplex.com/. - abgerufen am 2014-11-14

[RoBB13] ROSENBERG, ROBIN S. ; BAUGHMAN, SHAWNEE L. ; BAILENSON, JEREMY N.: Virtual Superheroes: Using Superpowers in Virtual Reality to Encourage Prosocial Behavior. In: *PLoS ONE* vol. 8 (2013)

[Rymi00] RYMIX, RHEMYST AND: *Kinect SDK Dynamic Time Warping (DTW) Gesture Recognition - Home*. URL http://kinectdtw.codeplex.com/. - abgerufen am 2014-11-14

[SaKo11] SADEGHIPOUR, AMIR ; KOPP, STEFAN: *Imitation Mechanisms of Social Resonance for Embodied Agents (IMoSA)*. URL http://www.techfak.uni-bielefeld.de/ags/soa/research/gesture-imitation/. - abgerufen am 2014-12-21

[Salo06] SALOMON, DAVID: *Curves and surfaces for computer graphics*, 2006 — ISBN 0387241965

[SFCS13] SHOTTON, JAMIE ; FITZGIBBON, ANDREW ; COOK, MAT ; SHARP, TOBY ; FINOCCHIO, MARK ; MOORE, RICHARD ; KIPMAN, ALEX ; BLAKE, ANDREW: Real-time human pose recognition in parts from single depth images. In: *Studies in Computational Intelligence* vol. 411 (2013), pp. 119–135 — ISBN 9783642286605

[ShGh14] SHINGADE, ASHISH ; GHOTKAR, ARCHANA: Animation of 3D Human Model Using Markerless Motion Capture Applied To Sports. In: *International Journal of Computer Graphics & Animation* vol. 4 (2014), Nr. 1, pp. 27–39

[Sori00] SORIANO, MARC: *Skeletal Animation*. URL http://alumni.cs.ucr.edu/~sorianom/cs134_09win/lab5.htm. - abgerufen am 2014-10-20

[ToMa06] TOBLER, ROBERT ; MAIERHOFER, STEFAN: *A Mesh Data Structure for Rendering and Subdivision*, 2006

[Unit00a] UNITY: *Unity - Game engine, tools and multiplatform*. URL http://unity3d.com/unity. - abgerufen am 2014-10-21

[Unit00b] UNITY: *Unity - Mecanim - Simple and powerful animation technology*. URL http://unity3d.com/unity/animation. - abgerufen am 2014-08-20

[Unit00c] UNITY: *Unity - Manual: GameObject*. URL http://docs.unity3d.com/Manual/class-GameObject.html. - abgerufen am 2014-08-20

[Unit00d] UNITY: *Unity - Manual: Hierarchy*. URL http://docs.unity3d.com/Manual/Hierarchy.html. - abgerufen am 2014-08-20

[Unit00e] UNITY: *Unity - Manual: Anatomy of a Mesh*. URL http://docs.unity3d.com/Manual/AnatomyofaMesh.html. - abgerufen am 2014-11-26

[Unit00f] UNITY: *Unity - Manual: Skinned Mesh Renderer*. URL http://docs.unity3d.com/Manual/class-SkinnedMeshRenderer.html. - abgerufen am 2014-11-26

[Unit00g] UNITY: *Unity - Scripting API: Skinned Mesh Renderer Example*. URL http://docs.unity3d.com/ScriptReference/Mesh-bindposes.html. - abgerufen am 2014-11-26

[Unit00h] UNITY: *Unity - Manual: Mecanim Animation System*. URL http://docs.unity3d.com/Manual/MecanimAnimationSystem.html. - abgerufen am 2014-10-30

[Unit00i] UNITY: *Unity - Manual: Animation Clip*. URL http://docs.unity3d.com/Manual/class-AnimationClip.html. - abgerufen am 2014-10-01

[Unit00j] UNITY: *Unity - Manual: Animator and Animator Controller*. URL http://docs.unity3d.com/Manual/Animator.html. - abgerufen am 2014-11-29

[Univ00] UNIVERSITY, CARNEGIE MELLON: *Microsoft Kinect - Microsoft SDK - Unity3D*. URL http://wiki.etc.cmu.edu/unity3d/index.php/Microsoft_Kinect_-_Microsoft_SDK. - abgerufen am 2014-11-14

[Wiki00a] WIKIPEDIA: *Polygon mesh - Wikipedia, the free encyclopedia*. URL http://en.wikipedia.org/wiki/Polygon_mesh. - abgerufen am 2014-08-17

[Wiki00b] WIKIPEDIA: *UV mapping - Wikipedia, the free encyclopedia*. URL http://en.wikipedia.org/wiki/UV_mapping#UV_mapping. - abgerufen am 2015-01-08

[Wiki00c] WIKIPEDIA: *Motion capture*. URL http://en.wikipedia.org/wiki/Motion_capture. - abgerufen am 2015-01-04

[YiJS06] YILMAZ, ALPER ; JAVED, OMAR ; SHAH, MUBARAK: Object tracking: A survey. In: *ACM Computing Surveys* vol. 38 (2006), p. 13 — ISBN 0360-0300

[Zhao14] ZHAO, CHENGLONG: *Working prinicple of Kinect: Structured light for applications in motion control - Welcome to Chenglong Zhao's Research Website*. URL http://chenglongresearch.weebly.com/information-center/working-prinicple-of-xbox-360-structured-light-for-applicationsin-motion-control. - abgerufen am 2015-01-08

[ZhCS02] ZHANG, LI ZHANG LI ; CURLESS, B. ; SEITZ, S.M.: Rapid shape acquisition using color structured light and multi-pass dynamic programming. In: *Proceedings. First International Symposium on 3D Data Processing Visualization and Transmission* (2002) — ISBN 0-7695-1521-4

Table of Figures

FIGURE 2.1 VIRTUAL WORLD AVATAR FOLLOWS GESTURE FROM PARTICIPANTS (LEFT) IS HOW THE PARTICIPANT SAW EACH OTHER IN VR WORLD 10

FIGURE 2.2 (LEFT) PUPPET PARADE IN RUNTIME. (RIGHT) SKELETON POINT FROM THUMB AND THE FOREFINGER WILL BE CREATED FOR CONTROLLING BIRD-AVATAR'S BEAK. 11

FIGURE 2.3 (LEFT) VIRTUAL WORLD WHICH PARTICIPANT SEE THROUGH HEAD MOUNTED DISPLAY. (RIGHT) PARTICIPANT RAISES THEIR ARM FOR FLYING SIMULATION 12

FIGURE 2.4 (LEFT) PARTICIPANT TRIES TO DRAW A CIRCLE HAND-MOVEMENT IN IMOSA (RIGHT) THE PROCESS OF RECOGNIZING THE HAND MOVEMENT 13

FIGURE 3.1: (A) IN THE POSE-TO-POSE METHOD THE IMPORTANT DRAWINGS ARE FIRST TO BE DRAWN (IN RED LINE) AND INTERMEDIATE DRAWINGS IN INBETWEENING (IN GRAY LINES); (B) IN THE STRAIGHT-AHEAD METHOD THE FRAMES ARE DRAWN ONE BY ONE FROM THE START TO END OF A SEQUENCE. 16

FIGURE 3.2: TRANSFORMING OBJECTS BETWEEN SPACES [PARE08] P. 45. 17

FIGURE 3.3: EXAMPLE OF TWO KEY FRAMES WITH ITS VALUE (POSITION). 19

FIGURE 3.4: KEY FRAMES WITH THEIR INTERPOLATED VALUES. 19

FIGURE 3.5: ARC AND NODE, $Nodei$ IS LEADING NODE OF $Arci$ [PARE08]. 21

FIGURE 3.6: NODE TRANSFORMATION WITHIN OBJECT SPACE, MAKING THE ROTATION POINT IN THE MIDDLE OF THE RECTANGLE. 21

FIGURE 3.7: EXAMPLE OF THE OBJECTS' TREE STRUCTURE IN ITS ORIGINAL DEFINITION AND THEIR ARRANGEMENT IN TREE STRUCTURE. 22

FIGURE 3.8: TRANSFORMATION FROM THE OBJECT SPACE OF THE HIERACHICAL MODEL INTO WORLD SPACE. (A) AND (B) SHOW NORMAL TRANSFORMATION OF ROOT OBJECT $Link0$ FROM OBJECT SPACE TO WORLD SPACE. (C)–(F) SHOW THE TRANSFORMATION FROM OBJECT $Link1$TO WORLD SPACE. 24

FIGURE 3.9: HIERARCHICAL OBJECTS IN WORLD SPACE AFTER TRANSFORMATION. 25

FIGURE 3.10: TREE STRUCTURE REPRESENTING THE HIERARCHICAL STRUCTURE OF AN ARTICULATED FIGURE[PARE08] P. 192. 27

FIGURE 3.11: EXAMPLE OF FORWARD KINEMATICS SPECIFICATION OF JOINT ROTATION. 28

FIGURE 3.12: EXAMPLE OF LINKAGES SYSTEM WITH END-EFFECTOR. 29

FIGURE 3.13: SAMPLE SEQUENCE OF POSITIONING THE END-EFFECTOR OF THE LINKAGE SYSTEM BY USING INVERSE KINEMATICS. NOTICE THAT THERE ARE TWO OPTIONS, (B) AND (C), AS POSSIBLE ANSWERS...............................30

FIGURE 3.14: REPRESENTATION OF THE INVERSE KINEMATICS AND FORWARD KINEMATICS OF ARTICULATED BODIES BASED ON [KUZA06]. 31

FIGURE 3.15: EXAMPLE OF USING A TEXTURE FOR CREATING THE SHAPE OF AN ARM. THE TWO METHODS DIFFER ON HOW THE TEXTURE IS MAPPED ONTO THE OBJECT. THE FIRST ROW DIVIDES THE TEXTURE IN THEIR CORRESPONDING OBJECTS AND USES IT IN THE HIERARCHICAL STRUCTURE AS PART OF THE OBJECT, WHILE THE SECOND ROW MAPS TEXTURE INTO ONE DEFORMABLE OBJECT AND DEFORMS IT ON THE BASIS OF THE MOVEMENT OF LINKS AND JOINTS. 32

FIGURE 3.16: VERTEX, EDGE, FACE, POLYGON, AND SURFACE IN POLYGON MESH [WIKI00A]. 33

FIGURE 3.17: POLYGONAL MESH WITH NORMALS (SHOWN IN RED ARROW) IN (A) A FACE DRAWS THE TEXTURE BASED ON ITS NORMAL (B) SHOWS THE TRIANGLE FROM BELOW (NOTICE THERE IS NO TEXTURE IN IT). (C) AND (D) SHOW THE VERTEX NORMAL AND HOW IT INFLUENCES THE LIGHT IT RECEIVES (THE TRIANGLE IS DIRECTLY UNDER THE LIGHT). 34

FIGURE 3.18: UV MAPPING FROM TEXTURE INTO THE POLYGONAL SPHERE OBJECT.[WIKI00B] 35

FIGURE 3.19: DEFORMABLE OBJECT BEFORE AND AFTER TRANSFORMATION: A IS THE ROTATION OF A SKELETON POINT, WHILE B IS THE ROTATION OF VERTEX V1 AND C IS THE ROTATION OF VERTEX V2. VERTICES V1 AND V2 ARE ATTACHED TO THE ELBOW ROTATION AND THUS FOLLOWS ITS ORIENTATION..36

FIGURE 3.20 MOTION CAPTURE BY USING LED INFRARED MARKER FOR ANIMATING A VIRTUAL CHARACTER [WIKI00C]. 38

FIGURE 3.21: (A) KINECT CAMERA AND ITS SENSOR/PROJECTOR; (B) THE IR DEPTH PROJECTORS EMIT LIGHT PATTERNS DETETCTED BY THE INFRARED DEPTH SENSOR; (C) THE LIGHT DOT PATTERN FROM INFRARED PROJECTORS.[ZHAO14] 39

FIGURE 3.22: HOW KINECTS GET SKELETAL POSITIONS [SFCS13]. 40

FIGURE 4.1 COMPONENT OVERVIEW 46

FIGURE 4.2: WORKFLOW IN UNITY 48

Figure 4.3: A GameObject with light component. The right window is the inspector view that shows the components of selecting a GameObject. The left window is the scene view, the world space of Unity. 49

Figure 4.4: Hierarchy view creating hierarchical structure in arms 50

Figure 4.5 Inheritance class from Unity Engine and Unity Editor 51

Figure 4.6 CustomScript C# script (left) and default inspector view of this script (right) after it is attached into a GameObject 52

Figure 4.7 Example of *EditorForCustomScript* C# script (left) and its custom inspector view (right) for *CustomScript*. Notice there is different with default view of script in figure 4.6 53

Figure 4.8 : Example of the OnSceneGUI function that creates the handle. First, the current CustomScript instance will be aquired through Editor.target. After assinging GUIstyle font color, a label will be written to the scene through the Handle.Label function. 54

Figure 4.9 Example of EditorWindow class: CustomWindow script (left), creation of window through the static function Init (right above), GUI of window based on the OnGUI function (right below). 55

Figure 4.10: Skinned mesh renderer component, it takes material (texture), mesh, and root bones (Game Object's transformation). 57

Figure 4.11: Animation window in Unity: Dope Sheet window (above) for managing key-frame, Curves window (middle) for managing interpolation function and animator window (below) for managing transitions. 58

Figure 4.12: System architecture of the developed environment. Components in blue are external libraries, while the green components are internal components. 62

Figure 4.13 Character modelling process 64

Figure 4.14 Imitation functionality during Play-mode...64

Figure 4.15 Recording a gesture template and connecting gesture with animator parameter.................65

Figure 4.16 accessing Kinect, recognizing gesture and triggering animation 65

Figure 4.17 The skeleton 'template' created from SkeletonControllerEditor. The number of skeleton points refers to the number of bones from the Kinect-plugin. Handles will be built at every skeleton point for direct rotation editing. 67

FIGURE 4.18 CLASS-DIAGRAM OF THE CHARACTERCONTROLLER COMPONENT. 68

FIGURE 4.19 CLASS-DIAGRAM OF COMPONENT POLYGONCONTROLLER...... 69

FIGURE 4.20 POLYGONCREATORWINDOW TAKES 4 INPUT: TEXTURE (SPRITE), MESH NAME, VERTEX DISTANCE AND SKELETONCONTROLLER. "CREATE MESH" AND "CREATE MATERIAL" BUTTON USE FUNCTIONALITY FROM MESHMANAGER CLASS. "CREATE SKIN" AND "(RE)CALCULATE WEIGHT AND ADD SKIN" USES FUNCTIONALITY FROM SKINMANAGER. 70

FIGURE 4.21 THE DEVELOPER NEEDS TO ALIGN THE SKELETON POSITIONS MANUALLY TO THE RIGHT POSITION OF THE CHARACTER SHAPE ("ADD SKIN"- BUTTON) 73

FIGURE 4.22 *GESTURE* COMPONENT CLASS DIAGRAM 1 75

FIGURE 4.23 GESTURE COMPONENT CLASS DIAGRAM 2 76

FIGURE 4.24 GUI FROM GESTURECONTROLLER FOR RECORDING GESTURE 79

FIGURE 4.25 INSPECTOR VIEW FROM *GESTURERECORDER* 79

FIGURE 4.26 THE *GESTURECONTROLLEREDITORWINDOW* WINDOW (LEFT) BEFORE THE GESTURE LIST AND ANIMATOR ARE LOADED; (RIGHT) AFTER LOADING THE GESTURE LIST AND ANIMATION PARAMETER, THE DEVELOPER CAN ATTACH THEM TOGETHER 80

FIGURE 4.27 *GESTURECONTROLLER'S* INSPECTOR VIEWS 81

FIGURE 6.1 CONFIGURATION BEFORE USING THE UNITY...... 85

FIGURE 6.2 (A) CREATING SKELETON MENU. (B) HIERACHY VIEW AFTER ADDING SKELETON. (C) SKELETON "TEMPLATE" IN SCENE VIEW. (D) SKELETON INSPECTOR VIEW 86

FIGURE 6.3 CHANGING SKELETON ROTATION WITH HANDLES. 87

FIGURE 6.4 TEXTURE TYPE SHOULD BE "SPRITE" 88

FIGURE 6.5 MENU FOR CREATING SHAPE AND ITS WINDOW GUI 88

FIGURE 6.6 CREATING MESH AND CREATING MATERIAL 89

FIGURE 6.7 ADDING SKIN TO SKELETON...... 90

FIGURE 6.8 TESTING THE MESH DEFORMATION...... 91

FIGURE 6.9 VARIABLE *SKELETONWRAPPER* NEEDS TO BE ASSINGED TO THE KINECT GAMEOBJECT AND IMITATION MODE SHOULD BE CHEKCED 92

FIGURE 6.10 FROM LEFT TO RIGHT: MENU FOR CREATING GESTURE RECORDER; GESTURE RECORDER GAMEOBJECT IN *HIERACHY VIEW*; *INSPECTOR VIEW* OF GESTURE-RECORDER GAMEOBJECT (KINECT GAMEOBJECT IS ASSIGNED AUTOMATICALLY) 93

FIGURE 6.11 GUI OF GESTURE RECORDER 94

FIGURE 6.12 RECORD A GESTURE 95

FIGURE 6.13 AFTER FINISH RECORDING A GESTURE 95

FIGURE 6.14 EXPORTING A GESTURE DATA 96

FIGURE 6.15 CREATING ANIMATION CLIPS IN UNITY 97

FIGURE 6.16 CREATING ANIMATION TRANSITION IN *ANIMATOR WINDOW* 98

FIGURE 6.17 CREATING GESTURE-ANIMATION EDITOR 98

FIGURE 6.18 IMITATION MODE SHOULD BE UNCHECKED WHEN USING INTERACTIVE FUNCTIONALITY 99

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den ________________ ____________________________