MASTERARBEIT

Titel der Masterarbeit

„Capturing Performance Based Character Animation in Real Time for Unity“

verfasst von

Hannes Wagner, BSc.

angestrebter akademischer Grad

Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2015

Studienkennzahl lt. Studienblatt:	A 066 935
Studienrichtung lt. Studienblatt:	Masterstudium Medieninformatik
Betreuerin / Betreuer:	Univ.-Prof. Dipl.-Ing- Dr. Helmut Hlavacs

Contents

List of Figures

List of Tables

1 Introduction

Creating animations is a time consuming process. Making them look natural is even harder. Animations are a vital part of digital media like movies or computer games. With the drastic rise in model quality, there has been a growing demand for complex animations with more and more details to make them seem natural.

In the early years of game development creating animations was sometimes simply done by showing different 2D graphics of an object successively, like a flipbook. Today, 3D models are mostly used, which can have simulated bones and muscles to create an even more realistic look. It is only logical that creating animations for such models is much more complex. This raises the question of how to deal with these new requirements.

One possible option is to manually produce the animations for these models step by step, using special software tools for modelling like Blender[1] or Maya[2]. But this costs a lot of time and requires very skilled and experienced graphic designers. This can be very expensive and thus might not be possible for small companies or indie developers. Another way of creating animations is using a motion capture system, which collects data about movement with some sort of tracking mechanism and applies it to a virtual model.

Motion capturing is a technology which can be applied to many areas, like [22] and [28], so it is not surprising that there are numerous publications every year pertaining to computer vision and motion capture. This thesis puts its focus on combining different input sources to offer a simple way of generating complex character animations for 3D humanoid character models.

The prototype for the tracking system was developed with Unity, a free game engine

[1]`https://www.blender.org/`, Accessed: 2015-06-07
[2]`http://www.autodesk.de/products/maya/overview`, Accessed: 2015-06-07

which offers some very useful features, like easy import and export of 3D models and incorporating various plugins to facilitate the integration of the input technologies. The current prototype combines motion tracking input from three sources: the body is tracked via Kinect v2.0 sensor for Windows, face feature points are calculated from webcam input and hand movement is captured with a 5DT Data Glove.

The goal of this prototype is to offer a new way of tracking multiple features at the same time since it uses both optical and non optical capturing methods. The gathered data is combined to a single animation, which can be saved and reapplied to any desired model. It is possible that future implementations might even be able to improve upon the quality of the tracking data by comparing and merging the data from different input technologies.

This thesis is organized as follows: Chapter 2 gives an overview of relevant projects in the field of motion tracking. Unlike this prototype, most of these projects focus on a single feature like face tracking or body movement. Since this project makes use of multiple technologies and methods to capture different parts of the human body, Chapter 3 gives an overview of the individual tracking methods and details how their input is combined. Chapter 4 documents the implementation process and main design decisions for the prototype, before giving reasons for selecting the chosen technologies and naming possible alternatives. It also contains a section about the technical difficulties which came up during the implementation process. After that, in Chapter 5 the results of the evaluation process will be shown. They will be discussed and compared with other technologies in Chapter 6. Finally in Chapter 7 there will be a summary of the improvements and benefits of this project, as well as a review of the limitations and its most common issues.

2 Related Work

Since the goal of this thesis is it to develop a performance capturing system with different input sources, this section is distributed into three main parts: hand, body and face tracking. Each section will cover related work for the individual research field. The combination of these different tracking approaches is then undertaken in the following chapters.

2.1 Hand Tracking

In [50] a completely markerless approach of hand tracking is introduced. It uses the Kinect's depth sensor for obtaining 3D image data. The images are analysed by using OpenCV functions to find the contours of the hand. Then the hand position is tracked with a special hand recognition algorithm that detects the center of the palm and constructs a convex hull around the hand as was described in the work referenced above.

A similar approach, but with a different algorithm is introduced by [36]. Both methods manage to correctly track hands in real time. However the obtained information's level of detail about the finger movement might not be good enough for applying the data to a model to generate natural looking finger animations.

Another interesting approach can be found here: [18]. This work is about real-time hand tracking, which specially sets its focus on tracking robustness. It uses a combination of the Camshift algorithm and motion velocity, but it does not retrieve enough information for detailed hand animations.

The approach introduced in [39] uses inverse kinematics to match a model to the 3D point cloud obtained from the Kinect sensor. The hand movement tracked with this approach shows promising results: The animations are detailed and look natural, but the system is prone to tracking errors of the Kinect. But while capturing, it can happen that the hands are moved faster than they can be tracked by the Kinect sensor. So in general this approach could be used, but a more robust tracking method would be appropriate for this prototype.

One of the most promising projects in this area is being developed by Microsoft Research [40], which has been working on a vision based approach using the Kinect sensor, and provides a very accurate tracking mechanism. Figure 2.1 shows the project in use from three different views.

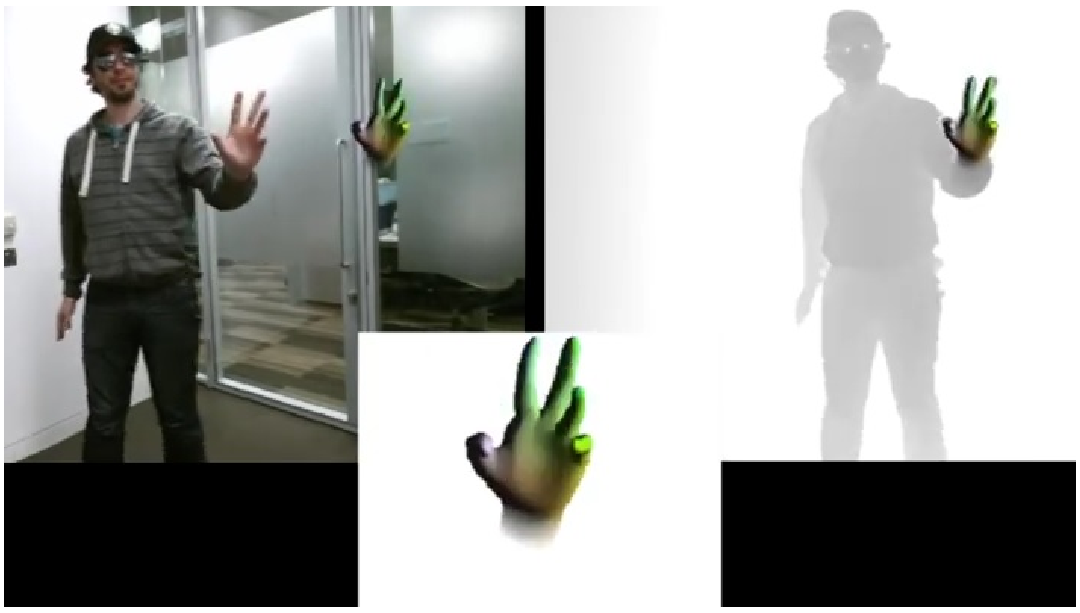


Figure 2.1: Screenshot of Kinect Hand Tracking Demo Video[1]

This tracking method only makes use of the Kinect's depth sensor. It has a high tolerance for changes in lighting and can quickly recover from tracking loss. It can track a user's hand, even if he is several meters away from the camera. Also the system does not need a fixed camera position in order to perform hand tracking. Which means that the Kinect can be moved freely during the tracking process.

The tracking of the features is very detailed and looks more natural than the hand animation of the prototype of this thesis. However though it is a vision based only

[1]`https://www.youtube.com/watch?v=A-xXrMpOHyc`, Accessed: 2015-07-24

approach there is always the issue that the view of the camera can be blocked by objects in the scene or the actors body. Therefore a vision based approach might not be accurate enough for recording an actor's overall performance.

2.2 Body Tracking

There have been many studies in the field of motion capturing, especially about capturing the movement of an actor's body. Therefore, compared to the vast number of methods available, only a rather small selection of tracking methods will be discussed in this section.

An interesting and low budget approach to body tracking has been the real-time Interactive Shadow Avatar [31] published in 2007. It only uses visual tracking via webcam: A virtual shadow is constructed from the webcam picture. By tracking hand positions through the virtual shadow, an estimate can be made as to the most likely position of the head. However, in this approach there is no information retrieved about the rotation and position of the body joints, because the tracking is done by using the body outline of the recorded person. Therefore the data would not satisfy the claims for creating animations at real time.

Another project from 2012 [45] explores how joint orientations, that are captured by a Kinect, can be interpreted and applied to a model. This project creates Biovision Hierarchy (BVH) -files at runtime, which can be used to store animation data and information about how the bones are connected with each other.

This approach [47] tries to create character animations with a Kinect. This is achieved by forming semantic skeletons from a motion capture data base. During the process feature points are calculated, and from this data an appropriate skeleton from the database is selected for the animation.

A rehabilitation exercise monitoring and guidance system is described here [51]. It uses the Kinect together with Unity to track patient's bodies during exercise. The difference to this thesis is that heuristics are used to check if the exercises are done correctly.

In this paper [52] an approach of mapping optical motion capture data to skeletal motion is realized. It is described as follows:

> To accomplish the mapping, we attach virtual springs to marker positions located on the appropriate landmarks of a physical simulation and apply resistive torques to the skeleton's joints using a simple controller

The results show solid mapping of the tracking data to the model. This approach could be considered as an alternative to the Kinect tracking of this thesis's prototype.

These three works [44], [17] and [26] show an approach with surface motion graphs. Here the surface motion and the pose of the model are used together for a new way of displaying movement and retrieving additional knowledge. The intensity of the motion is visualized by different colors. For example jump height, movement speed or the distance between steps are factors in computing the motion intensity. With this kind of additional data it is possible to further improve animations, for example by adding rules on how the actor should be tracked if he moves at a certain speed or when he jumps.

Another interesting technology in this field is Control VR [3], a device to track body motion. The device uses special sensors to track interesting feature points of the body. It uses mechanical feature tracking and therefore it even deals well with loss of sight of the actor.

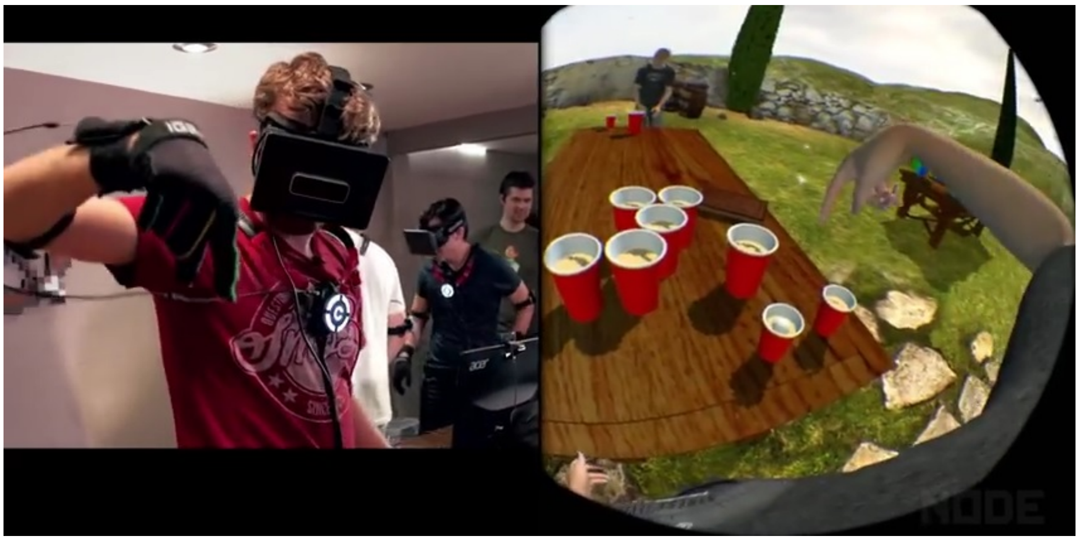


Figure 2.2: Screenshot of ControlVR Demo Video[2]

[2]https://www.youtube.com/watch?v=_jgAJcmmlVs, Accessed: 2015-07-24

The screenshot of the Demo Video in Figure 2.2 shows the ControlVR wearable controller used together with the Oculus Rift[3]. It simulates a scene where the user can interact with his environment and other players. By adding collides to the model it seems to the user like he can really affect his surroundings by touching, pushing or even grabbing objects in the scene. This approach combines motion tracking and augmented reality, to give the user the feeling that he can interact with the virtual scenery like in real life. Although the main goal of the prototype presented in this thesis is the recording and playback of animations, it would be possible to achieve similar results by using and adjusting some components of the ControlVR hardware and adding Oculus Rift support to the project.

An alternative, advanced way of motion capturing is performed by Xsens [12]. It uses mechanical motion capturing, where the actors wear special suits equipped with sensors that record motion and send the data to the devices used for capturing. With this approach it is possible to create detailed animations without having any issues of tracking loss. Also, the actors can move very fast and the tracking is still stable.

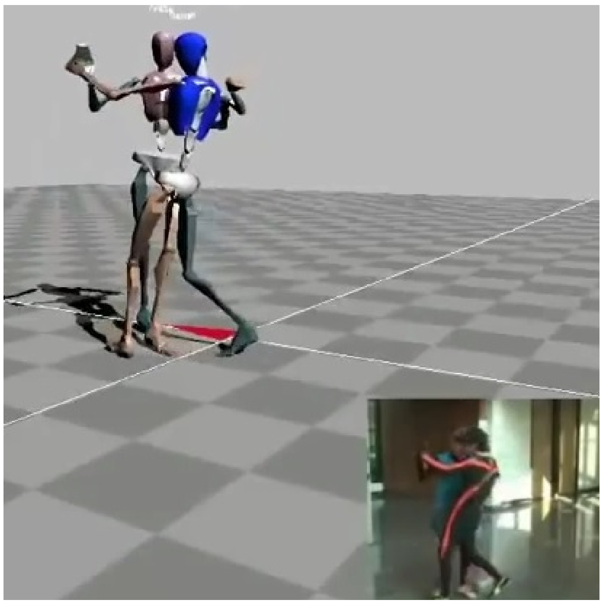


Figure 2.3: Screenshot of Xsens Demo Video[4]

Figure 2.3 shows the tracking of two people performing dance moves in close body contact with each other. The presented tracking system is very well suited for this

[3]`https://www.oculus.com/en-us/`, Accessed: 2015-07-24
[4]`https://www.youtube.com/watch?v=5ZWfIXmI5gY`, Accessed: 2015-07-24

kind of tracking, since it does not rely on visual information. If the same scene was supposed to be tracked with optical motion tracking methods, multiple cameras would be needed, because the two actors would frequently cover each other's body parts by being in such close proximity to each other. So regardless of whether marker-based or markerless tracking methods are used, there is always the possibility of tracking loss. This approach would be a good alternative to visual motion capturing like it is performed in this thesis' prototype.

Some further approaches for motion capturing are [42], [48], [24] and [29]. The first one compares a 3D measuring device with the Kinect sensor. The second one introduces an approach which combines 3D estimated motion data with 2D data obtained from posture tracking. The third also uses 2D motion data to reconstruct 3D human motion from a sequence of images. The last project puts its focus on markerless tracking of animals.

However, even though motion capturing is a very interesting and diverse field of research,it is not the only focus of this thesis. The capturing of facial expressions is performed by a separate component of the system.

2.3 Face Tracking

There are a lot of approaches for real time face feature point extraction. This section will only list references to published scientific work, but later in Chapter 6 the prototype also will be compared against some commercially used products.

This project from 2013 [30] proposed a way to produce real-time performance-driven facial animation without having to use markers. It uses Kinect Face input together with 3ds Max[5] and maps 2D features to a face model which contains a skeleton that is modelled in a special way to simulate face muscles. This project also provides options to track blinking and pupil motion.

Another approach for vision based face animation can be found here [20]. It first extracts feature points and additionally to displaying them it also tries to calculate the emotion of the user.

[5]`http://www.autodesk.de/products/3ds-max/overview`, Accessed: 2015-07-03

This paper [16] from 2006 shows a way how face feature points can be tracked with a webcam. First the face is detected via an OpenCV, using Haar-like features, which was also initially attempted by the prototype. Then it records the skin color distribution and detects face features with the help of edge orientations.

An interesting approach is shown in [21]. This work concentrates on dealing with the fact that input images might be partially blurred. It introduces new algorithms to reconstruct the images to high quality using depth information recorded by a binocular camera setting. An approach like this could greatly help stabilize face feature detection even when actors move around quickly. On the other hand additional calculations would be needed for computing the image recovery. This could negatively influence the run time performance of a tracking system. Alternatively the smoothing and feature capturing could be done on a video stream. If the tracking is not real time there is no need for fast calculations.

In [32] a face tracking method is proposed which uses face detection to identify regions of interest and then extract feature points. It works in real time and performs the tracking on a stream of input images. The whole approach works with a simple webcam, which is exactly what this thesis is trying to achieve as well, except that after retrieving the face feature points this project applies them to a model to generate 3D face animations. Therefore an approach like this would be suitable to compute the desired face feature points for the facial animations.

A great state of the art approach for capturing facial animations is Face Plus [6], provided by Mixamo[6]. It is a tool which can create 3D character facial animations by using pre recorded videos or live capturing with a simple webcam. It works with bone driven characters as well as blend shapes. Recording can be performed at real time and additional animations can be made even after the data is recorded. Figure 2.4 shows a snapshot of a live capturing demo.

Face Plus is a highly advanced method of what this project is trying to achieve in terms of capturing facial animations for 3D characters. Compared to the developed prototype it supports much more functionality. In its current state the prototype of this thesis only supports bone driven animation capturing, which means that simulated bones have to be added to the model which are then moved according to the captured face features.

[6]`https://www.mixamo.com/`, Accessed: 2015-07-30
[7]`https://www.youtube.com/watch?v=yNaympgBVpQ`, Accessed: 2015-07-24

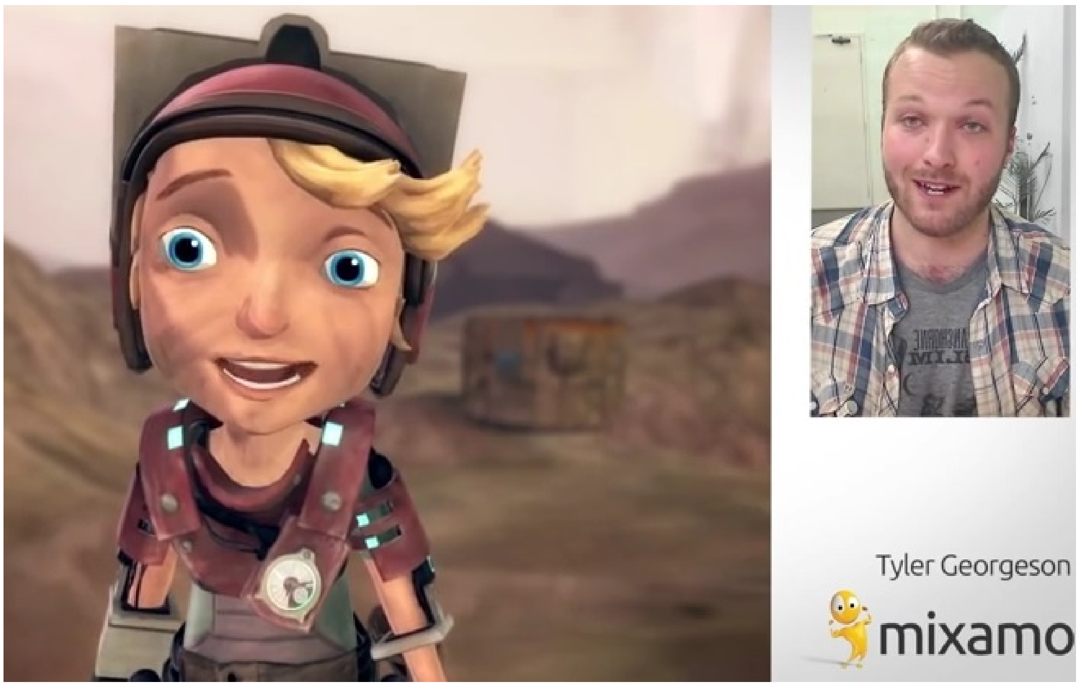


Figure 2.4: Screenshot of Face Plus Demo Video[7]

Furthermore Face Plus supports export formats which can be used with other animation tools. It is also available as a free trial version for Unity. Even though it is a commercial product, it is a cheap way of generating high quality facial animations. Therefore it can be considered as a possible input source for the face tracking component of this thesis's prototype.

2.4 Summary

Some of these projects show promising results in their particular area. None of these projects provide a complete tracking system for all of a person's features, but if some of them were to be used in a combined approach, they could show great potential.

3 Background and Basics

Since the prototype uses many different technologies, this chapter will give a general outline of the thesis and will define some terminologies. It also contains an overview of the development environment and the parts of it which were important for the prototype.

At the end of the chapter there will be a list of the used technologies, an short description what they are used for and the sections in which they are covered.

3.1 Motion Capture

In [22] motion capture and motion capture systems are referred to as

> (...) the process of recording motion data in real time from live actors and mapping it into computer characters. Motion capture systems include (...) devices that directly or indirectly track the actor's movements.

There are many different types of motion capture systems with a variety of tracking techniques. This section focuses mainly on optical and mechanical tracking methods, since the prototype is built from a combination of both. But it should be mentioned at this point, that there are also other techniques like acoustic or magnetic signals.

3.1.1 Optical Motion Capture

With optical motion capturing the features are retrieved through the analysis of a video stream. Such systems are often prone to tracking loss due to change of lighting or to objects blocking the view between the camera and the actor. Another issue might be the distance between the actor and the optical device. There are two common approaches in optical motion capture, marker based and markerless motion capture.

Marker-based Capturing

The marker-based approach uses optical beacons to capture important feature points of the tracked body. The beacons are often made out of materials which are reflecting or have a certain color. The so tracked data can be mapped to the model's joints corresponding to the position of the respective beacon. Often reflecting spherical markers are used for the tracking. These markers are attached to the body of the actor in all the places that are interesting to track. But there are also other marker technologies, as illustrated in [15], which presents an alternative contour-based marker system. This system is well-suited for tracking non-rigid motion by attaching the markers to key areas where the skin is close to the bones of the body.

Motion tracking technologies can also be used in other fields than generating animation data. For example in [19] a marker-based approach is used to create a system for applying a dance scoring system to the performers.

In [35] marker based tracking technologies are used to control a humanoid robot.

Even though no marker-based technologies are used for the tracking of the current prototype of this thesis, in future releases marker-based capturing methods could be added to improve and stabilize the tracking.

Markerless Capturing

Markerless capturing methods make no use of optical beacons. They often use algorithms which calculate features just from the bare image. To make calculations more efficient the image used in most cases is a resized and gray scaled version of the original.

In the last years countless approaches have been made to achieve motion capture with markerless capturing methods as in [46]. Many of these approaches use multiple cameras to stabilize the tracking, as for example in the following systems:

In [27] unsynchronized input from multiple cameras is used for generating motion capture data. The captured data is synchronized after recording by a specially developed mechanism.

The more cameras are used the better the motion of the actors can be reconstructed. For example [33] specialized in tracking two performers in close contact. This is especially challenging, since body parts of the actors might block the view of the camera. However using multiple cameras also means having a bigger amount of data to process, which can affect the run time performance.

The prototype exclusively uses markerless capturing methods for face and body tracking. While the Kinect makes use of the depth camera to calculate the position and rotation of the body joints the face tracking features are obtained by analysing the bare input image from a webcam.

3.1.2 Mechanical Motion Capture

Mechanical motion capture uses sensors which move along with the body of the actor to simulate an exoskeleton, which can represent the movement of the actors joints. These sensors deliver the desired data which is then used to animate the model. The prototype uses this kind of technology to track finger movement more accurately.

Using this kind of tracking has the big advantage that the actor does not always have to be in an area where he can be filmed. Also the mechanical tracking is not limited to hand tracking only. For example in [43] a full body tracking system is introduced, which could be used with this thesis's prototype as an alternative to the visual body tracking with the Kinect sensor.

Again, a tracking approach like this can have other application areas as well. For example [34] uses their tracking system to control a robotic puppet in real-time, by tracking a person's movement. The data is then interpreted and mapped to a link model of the human body, which is similar to the strategy used in the prototype of this thesis.

All of these sensor tracking methods produce data that needs to be transmitted to a computer where it can be processed in some way. For the prototype this is done by connecting the data gloves to a computer via USB cable, but there are other ways, too. In reference [25] the data of the motion capture system is transmitted by using a wireless

MARG (Magnetic, Angular Rate, and Gravity) sensor network, which has a high-speed update rate and has the advantage of supporting multi user tracking.

But however the data is transmitted, it is clear that mechanical motion capture technologies are well suited for detailed tracking. Especially in critical tracking areas like hands, which may be covered by objects held by the actor or by his body.

3.2 Performance Capture

Of course the prototype developed for this study can not be compared to professional performance capturing studios like for example House of Moves[1]. Figure 3.1 shows a professional performance capture in the studios. On the other hand the benefit of a simpler system is that it is much easier to use and of course it is way less expensive.

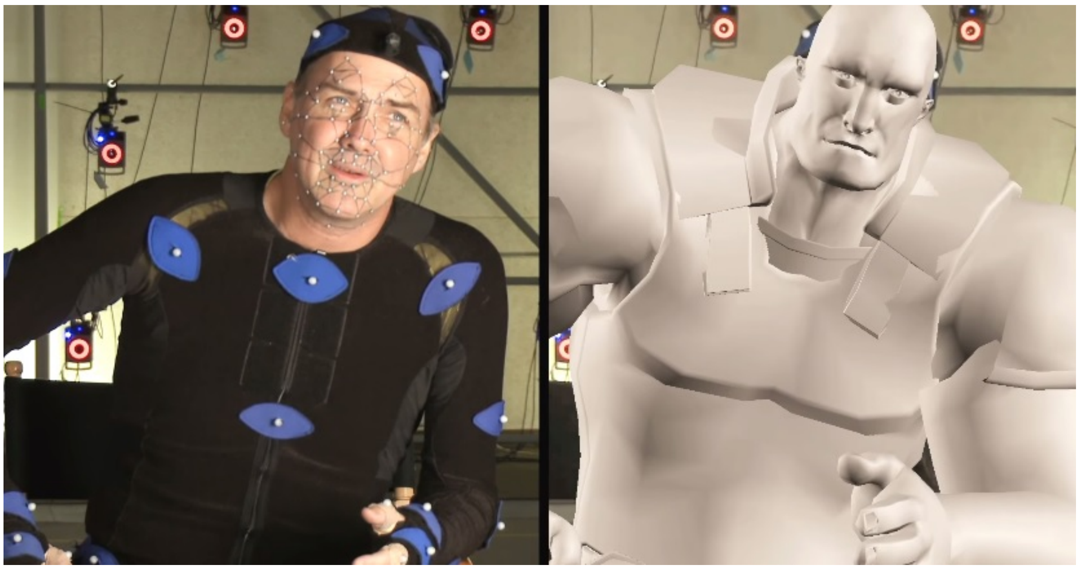


Figure 3.1: House of Moves Performance Capturing[2]

Professional performance capturing systems which are used for creating high quality animations for movies or video games need a lot of preparation. Every day the actors have to first be prepared for recording, which means attaching markers all over their body and face which must exactly match the model's proportions.

[1]`http://www.moves.com/`, Accessed: 2015-07-25
[2]`https://www.youtube.com/watch?v=WmtzTu2rygk`, Accessed: 2015-07-30

Also, for the actors to be able to interact with the environment, it is necessary to build a scenery which must match the proportions of the virtual models.

Another big performance capturing studio is for example Digital Domain[3]. All these studios offer a lot of different high quality services: Not only performance capturing but also specific motion or face capturing animations, scenes with green screens, editing, rendering and much more.

The prototype targets independent developers or small companies which do not have the money to hire a professional motion capture company, but still want realistic and detailed animations for their models. The tradeoff clearly lies in the animation quality, but the prototype is free to use and can be modified to match individual requirements.

3.3 Unity

Unity[4] is a development platform for creating games. It is free of charge and since version 5 it also supports dynamic linked library (DLL) files, which earlier was only possible with the professional versions.

Another powerful feature is that projects created in unity can be exported to many different target platforms. Since the prototype was developed solely with the Unity environment, this section will cover the aspects of Unity which were relevant during the implementation process and how they were used in the project.

3.3.1 Projects and Scenes

Every Unity Project contains an Assets folder. This folder contains the materials, models, prefabs, scripts, plugins and all other assets used by the project. A project can contain multiple scenes. A Scene represents an arrangement of assets in a 3D or 2D space.

[3]`http://www.digitaldomain.com/`, Accessed: 2015-07-30
[4]`http://unity3d.com/`, Accessed: 2015-06-07

3.3.2 GameObjects

GameObjects are containers for all the objects inside a scene and are used to composite and arrange them. For example: in a hierarchy of GameObjects, if the root is moved, scaled or rotated the same actions apply to all underlying GameObjects as well.

Every GameObject also has a transformation object attached to it. Those objects contain information about position, rotation and scale of a GameObject. Especially position and rotation play an important role for the prototype. They are essential for mapping all of the input sources to the GameObjects, so that their features can be stored as animations.

3.3.3 Joints

In the prototype empty GameObjects are used as joints between the bones of a model. They can be manually added, positioned and rotated. Then the bones are added as child nodes of the joints. If done correctly this combination of joints and bones can act as a skeleton within a model. This process is also referred to as rigging.

3.3.4 Rigging

For the rigging process it is very important to get the bone hierarchy in the right order, so that all joints that depend on each other are properly connected. The recommended joint hierarchy from the Unity documentation [9] is composed as follows:

- HIPS - spine - chest - shoulders - arm - forearm - hand
- HIPS - spine - chest - neck - head
- HIPS - UpLeg - Leg - foot - toe - toe_end

However, for the mapping of the Kinect input the bone hierarchy is slightly different and will be described in detail in Section 4.6.

3.3.5 Prefabs

If a scene contains a complex GameObject which should be accessible for reuse, Unity provides the option to save it as a prefab. The prefab object contains all the elements and specifications made in the editor and it can be used in every scene, as often as desired.

3.3.6 Scripts

Scripts are executed in Unity by attaching them to GameObjects and calling predefined functions like:

```
void Start () {} // when the script is enabled , before Update
   is called
void Update () {} // called every frame
```

In the first implementation all scripts are written in C# language. The prototype heavily depends on the scripts since they are used for retrieving input data, and interpreting and applying them to the right joints.

3.4 Integration Overview

This section provides an overview of all technologies, code projects and programs which were used for this thesis, and lists the specific purpose they were used for and the section in which they are cited and described in detail.

The entries of Table 3.1 are sorted by name, but will be described grouped by logical context within the prototype.

The 5DT Data Gloves are special gloves that have sensors to track the finger movement of the wearer's hand. For the prototype they are used to record more detailed information about finger spread and flexure, since the tracking data which is retrieved from the Kinect sensor is not detailed enough.

The Kinect V2.0 SDK for Windows needs to be installed in order to use the Kinect V2.

Name	Usage	Sections
5DT Data Gloves	Hand tracking	4.4.2, 4.5
Beyond Reality Face	Face tracking(removed from prototype)	4.4.4
CSIRO Face Analysis SDK	Face tracking	4.4.4, 4.7.2
EmguCV	OpenCV wrapper (removed from prototype)	4.4.4
Kinect V2.0 SDK for Windows	Body tracking	4.4.3, 4.6
Kinect v2 with ms-sdk	Kinect concepts	4.6
Luxand FaceSDK	Face tracking (removed from prototype)	4.4.4
OpenCVSharp	OpenCV wrapper (removed from prototype)	4.4.4
Unity	Development environment	3.3, 4.4.1, 4.9.2

Table 3.1: Integration Overview

It also contains basic examples on how to use the SDK with Unity. The project Kinect v2 with ms-sdk on the other hand is a code project which is available on Unity's asset store. It contains detailed examples and tutorials about how to use the Kinect V2.0 SDK.

In the beginning the face tracking of the prototype was done by using OpenCV. To be able to use the necessary libraries two different wrappers, OpenCVSharp and EmguCV, were tested. Even though the OpenCVSharp wrapper was easier to use, the EmguCV wrappers seemed to be maintained better and proved to have more detailed documentation. In the end the approach of using OpenCV did not satisfy the needs of the prototype because the tracking was not stable enough, nor was it possible to retrieve the desired number of face feature points. Therefore an external SDK had to be integrated for the tracking. Beyond Reality Face and Luxand FaceSDK provide solutions for face tracking which can retrieve face feature points by using a simple webcam. However, there were compatibility issues with the prototype, which are described in Section 4.9. For this reason the CSIRO Face Analysis SDK was chosen instead.

All these technologies were integrated into Unity and the input from the different sources was combined to achieve performance tracking of an actor. The following sections describe the detailed process of how this was attained.

4 Implementation

This section provides deeper insight into the structure and the implementation of the prototype. It starts by giving an overview of how the tracking is done in general, and then illustrates the components, the system architecture and general system requirements to run the prototype.

This is followed by sections about each tracking method and its respective hardware and software components, as well as one section about how the animation data can be stored and imported at runtime.

The last section covers issues that might affect the prototype during runtime or cause problems with the system integration.

4.1 Tracking Process

Figure 4.1 illustrates in detail, how the individual hardware and software components work together to generate real time character animations, by demonstrating a complete setup of all supported animation tracking sources and their dependencies.

The prototype consists of three main input devices: Body/Skeleton tracking with Kinect v2.0, face tracking via a webcam and measuring finger abduction and flexure with 5DT Data Gloves. The first step of this project was to integrate each of these components into separate Unity projects. In the second implementation cycle all three of these projects were combined into one prototype.

In the current state of the project, at runtime, the input from each device is handled individually before being combined and mapped to the avatar. The resulting animation data can also be exported by writing it to a file.

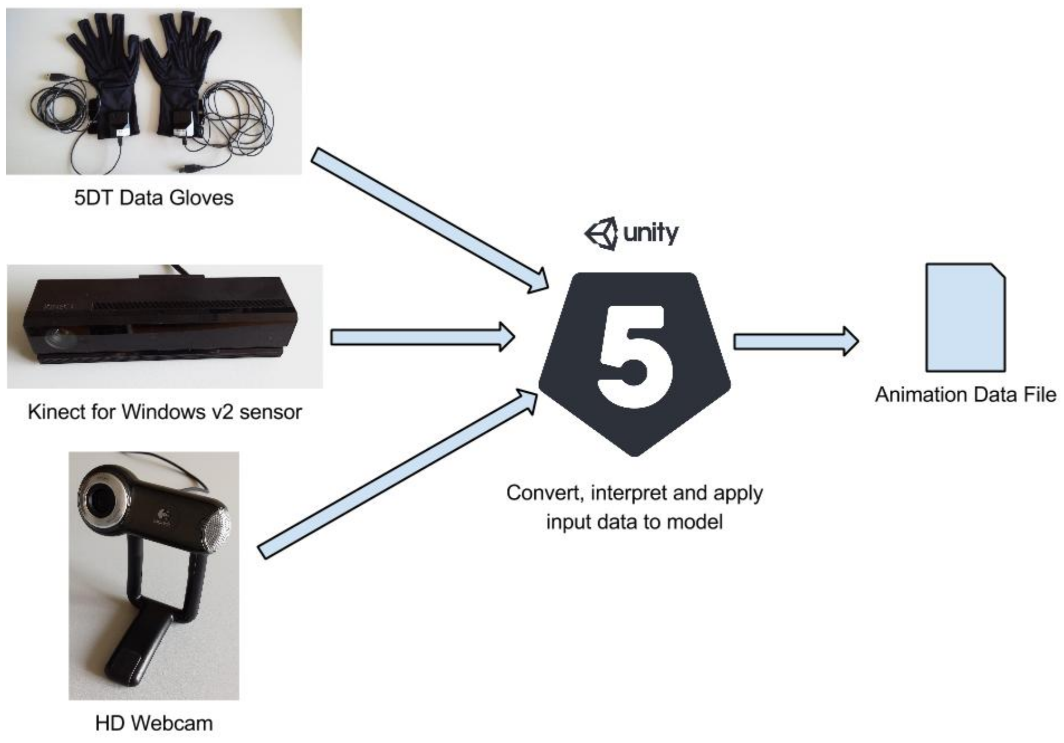

5DT Data Gloves
Kinect for Windows v2 sensor
HD Webcam
unity
5
Convert, interpret and apply input data to model
Animation Data File

Figure 4.1: Animation Process Pipeline. (Source for the Unity Logo [10])

4.2 Architectural Overview

This section is about the architecture of the prototype and how the hardware is connected to the software parts in the tracking process. Figure 4.2 shows the essential core components of the prototype, and how they work together while creating animations or applying them to a model.

Every input device is accessed by a separate SourceManager component. From there the input data is passed on to the model controller scripts. These scripts can manipulate the model by calling referenced model manager scripts which are attached to the model and hold references to its joints.

While a model is manipulated the AnimationManager can record these animations and store them in an animation file at runtime.

Such an animation file can be loaded and reapplied to any model with model manager scripts attached to it via the AnimationReader. However it is not possible to track and

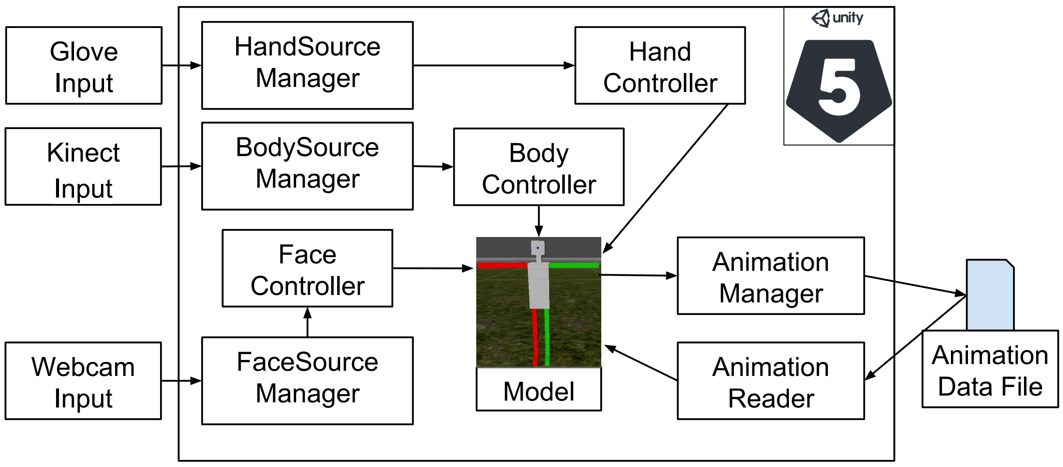


Figure 4.2: Architectural Overview of the Core Components

apply animation data at the same time to the same model. Although if there are multiple models in one scene each model can have either a controller or AnimationReader script manipulating it.

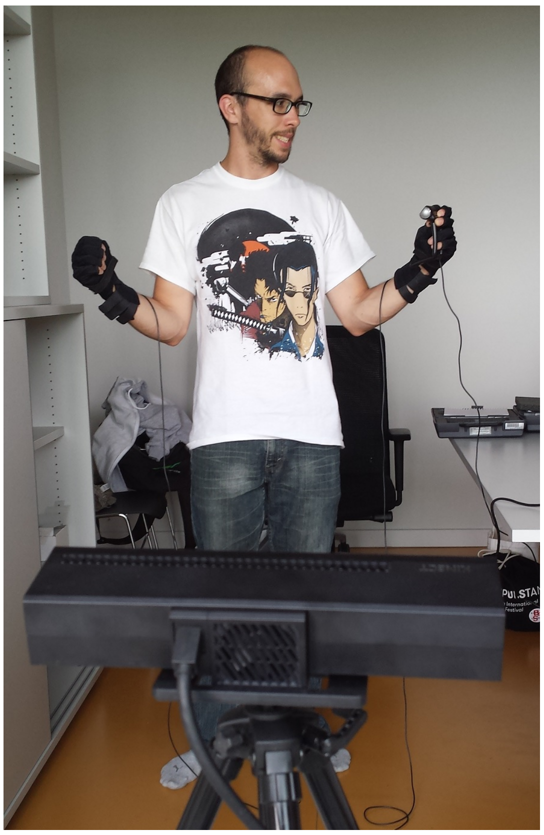
Figure 4.3 illustrates how all the modules interact, when all devices are used together in one tracking session.

The 5DT Data Gloves are also available in a wireless version. This would make it easier for the actor to move and perform freely. The Kinect should be placed in a way so that there is enough space for the actor to move around. Also there should be no objects between the Kinect sensor and the actor. The webcam should be attached in a way that it can film the actors face from a fixed position to prevent tracking loss of the face. Also no objects should move between the webcam and the face of the actor.

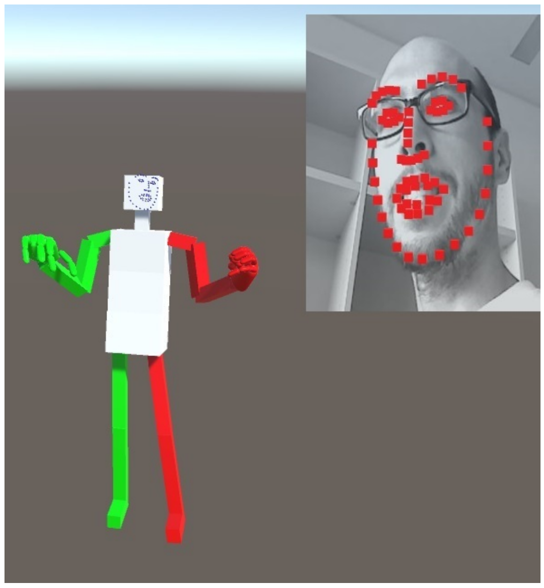
4.3 Implementation Overview

To visualize how the project is organized, Figure 4.4 gives an overview of the most important components which were used in building the structure of the code.

For each of the three input sources, there is one individual DeviceManager, ModelController and DataManagement component.



(a) Performing Actor



(b) Results in the Unity Editor Window

Figure 4.3: Prototype in Use

The DeviceManagers are responsible for retrieving the input data from the corresponding device. They usually use included .dll files and wrapper classes and serve as access points for the ModelController. Every DeviceManager holds a reference to exactly one input source and should be placed in a separate GameObject to create an organized project structure.

The ModelController is responsible for interpreting the device input and applying it to the model via the DataManagement component. Therefore, each ModelController holds a reference to a DeviceManager and its corresponding Manager component (HandInput-Source -> HandController <- HandManager).

The DataManagement component contains a Manager script for each part of the model that can be animated (Body, Hand, Face). The Manager scripts have references to the model's joints and also contain individual, additional specifications which influence how the model components are interpreted by the ModelController.

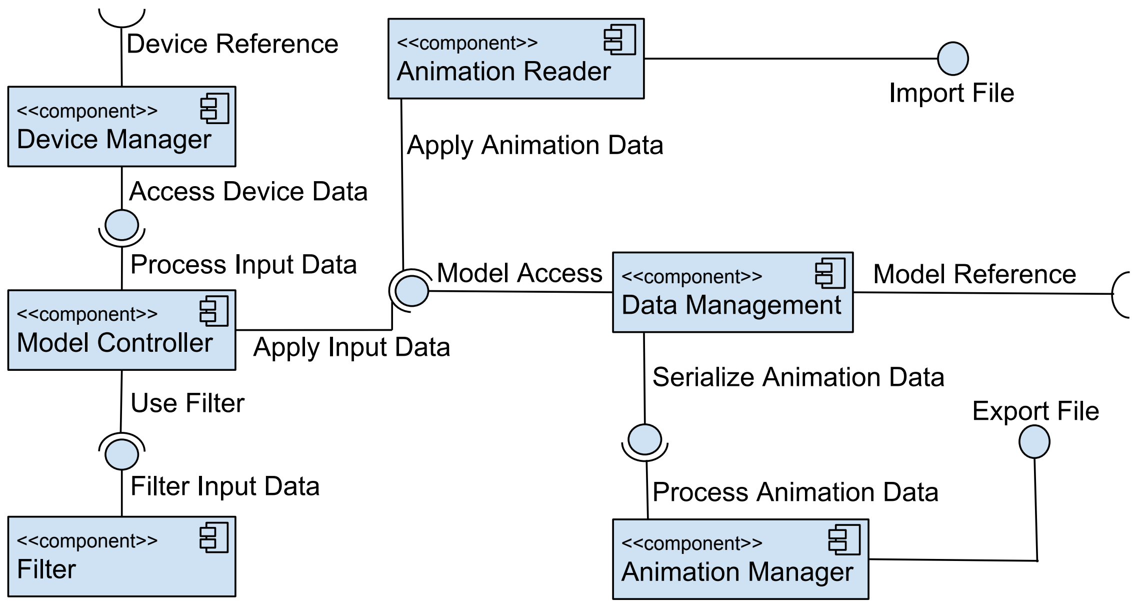


Figure 4.4: Component Diagram of the Prototype

The AnimationManager is used for real time animation capturing. Therefore it can hold a reference to each DataManagement component and retrieve animation data which then is converted to feature vectors. These vectors are then stored in a file which can be loaded and interpreted by the AnimationReader component.

If the scene contains an animation, the AnimationReader loads the animation file when the prototype is started. It can hold a reference to each DataManagement component in the scene and access their joints to apply the previously loaded animation data to them.

Both, the ModelController and the AnimationReader components use the same interface to access the 3D model. Therefore the user has the choice to use the ModelController to generate animation data, which can be exported by the AnimationManager, or to apply an animation file to the model via the AnimationReader. As a design decision, it is not possible to do both at the same time. Consequently, a GameObject which holds an AnimationReader can not have a ModelController on it.

4.4 General Requirements and Setup

Multiple technologies were involved to make it possible to use so many different input sources within a single project. But this also requires certain prerequisites from a system to be able to run the prototype properly.

The following section covers all the preparatory adjustments that need to be made to be able to execute all the scripts inside the Unity environment.

4.4.1 Unity

The minimum version to run the prototype is Unity 5. Some of the components also run with Unity 4.6.5 Pro edition, but the face tracking was exclusively developed with Unity 5, so there is no guaranty that it will work with earlier versions.

For the prototype to be able to fully utilize all the possible tracking data input sources, it is necessary to use the 32-bit version of Unity, since some of the .dll files were only available as x32 builds. Further details about this issue are covered in Chapter 4.9.

4.4.2 Finger Tracking

Before the 5DT Data Gloves can be used with a Unity project, the corresponding drivers need to be installed. They can be found online on the 5DT homepage [1] in the download section, where the official SDK is available. The SDK mainly consists of two files, the FDTGloveUltraCSharpWrapper.dll and the fglove.dll.

In October 2012 a Unity example was added to the website, which demonstrates the correct way to access the Data Glove and its parameters and also shows how to recognize finger gestures. The fglove.dll file has to be placed in the Assets/Plugins folder and the FDTGloveUltraCSharpWrapper.dll needs to be in the same folder as the C# scripts. Once the setup is complete the input of the gloves can be accessed through the wrapper class.

4.4.3 Kinect

The first step is to ensure that the system requirements are met, as recommended on the official SDK website [4] under Getting Started/System Requirements. The next step is to download and install the Kinect v2.0 for Windows SDK and the Kinect v2 Unity packages from the technical documentation and tools section [11]. Notice that it is only possible to use these packages with the free version of the Unity editor if Unity 5 or later is installed. Otherwise the .dll files can not be included into the project.

The Unity package is a custom package and can be imported via Assets -> Import Package -> Custom Package... Once this is done and the Kinect sensor is plugged in at an USB3.0 port, the SDK can be used to build Kinect-based Unity applications. However, the current release (Oct 2014) still has some issues that, in some cases, can even make Unity crash. The most common problems are listed in the known issues section, which can be found at the SDK website [4].

The .zip file contains the following Unity packages:

- Kinect.2.0.1410.19000.unitypackage
- Kinect.Face.2.0.1410.19000.unitypackage and
- Kinect.VisualGestureBuilder.2.0.1410.19000.unitypackage

The first package is necessary for all the general Kinect functions and also includes some tutorials about how to use them. The second package contains the Kinect Face scripts and plugins and the third one contains the visual gesture builder, which was not used for the implementation of the prototype.

When including these packages to Unity, it is important that the Kinect.2.0.1410.19000.unitypackage is imported before the Face and the VisualGesture-Builder packages can be used. If it is not imported first, the project will not work correctly. Another important thing is that some packages partly import identical scripts in the Standard Assets folder. This can cause problems with the project structure because of multiple namespace definitions in the Helper and the Windows.Kinect namespaces.

The affected scripts are the following:

- Standard Assets/CameraIntrinsics
- Standard Assets/CollectionMap
- Standard Assets/EventPump
- Standard Assets/ExceptionHelper
- Standard Assets/INativeWrapper
- Standard Assets/KinectBuffer
- Standard Assets/NativeObjectCache
- Standard Assets/NativeWrapper
- Standard Assets/SmartGCHandle
- Standard Assets/ThreadSafeDictionary
- Standard Assets/Editor/KinectCopyPluginDataHelper

Early implementations showed, that unfortunately there seems to be a compatibility issue with the Kinect v2.0 Face scripts and Unity 5 (tested version 5.0.2f1). Due to memory leak problems the Editor crashed every time the project attempted to retrieve the face data. For this reason the Face packet was omitted in following implementations. The current solution to this problem is to set the orientation of the head at a fixed angle, since the Kinect bones do not reflect the head orientation well enough. Instead, the head orientation is obtained by using its face tracking feature to make sure the head faces the correct direction.

The basic Kinect package should work fine right after being imported. The .zip file also contains some code examples that illustrate how to use the basic features of the Kinect v2.0 with Unity. For this prototype some of them were extended to build the basic structure for retrieving the Kinect input data. Once the package is imported the Kinect features can easily be included with the following statements:

```
using Windows.Kinect;
using Microsoft.Kinect.Face;
```

4.4.4 Face Tracking

Originally the following methods were considered for face tracking and feature extraction:

- OpenCV together with C# Wrapper
  - OpenCVSharp[1] –> works with Unity, but no sufficient face feature support, and tracking is often unstable
  - Emgu CV[2] –> works with Unity, but no sufficient face feature support, and tracking is often unstable
- Luxand FaceSDK[3]–> only native plugins, but Unity 5 needs at least one managed plugin which contains only .NET code
- Beyond Reality Face[4] –> Not supported for C#
- CSIRO Face Analysis SDK[5] –> Not supported for Microsoft platforms, but for Linux and Mac.

OpenCVSharp

The OpenCVSharp project provides the necessary .dll files to integrate OpenCV into Unity via C#. However, to be able to use it properly the OpenCVSharp master project must be recompiled. The desired .dll files are then placed into different directories.

The OpenCvSharp.dll can be found under opencvsharp-master/src/OpenCvSharp/bin and the other .dll files can be obtained from various locations e.g. opencvsharp-master/src/OpenCvSharp.Sandbox/bin. Depending on the build options they are then available as x32 or x64 builds.

Once recompiled the .dll files can be copied into Unity's Assets/Plugins folder. After that it can be simply imported with the "using OpenCvSharp;" statement.

[1]`https://github.com/shimat/opencvsharp`, Accessed: 2015-07-04
[2]`http://www.emgu.com/wiki/index.php/Main_Page`, Accessed: 2015-07-04
[3]`https://www.luxand.com/facesdk/`, Accessed: 2015-07-04
[4]`http://www.tastenkunst.com/#/home`, Accessed: 2015-07-04
[5]`http://face.ci2cv.net//`, Accessed: 2015-07-04

EmguCV Wrapper

Including the EmguCV wrapper is done similarly to including the OpenCVSharp wrapper. The only additional step that needs to be performed is switching the API compatibility level from .NET 2.0 Subs to .NET 2.0. The settings for this option can be found under File->Build Settings... There is a Settings button which can be used to bring up the Player Settings window in the inspector. In the Other Settings category there is an under category Optimization where the API Compatibility Level can be selected.

CSIRO Face Analysis SDK

Finally, the CSIRO Face Analysis SDK was chosen for the prototype, even though it had to be included into Unity by writing a wrapper for it.

The documentation for this SDK can be found here [5] and the source code is available on Github[6]. Furthermore, the face tracking component of the SDK is based on [37]. The expression transfer, which is not supported by the prototype, is based on [38].

To be able to use the functionality of the face tracking component a wrapper class was added to the project, containing the following functions:

```
FACETRACKER::FaceTracker* WrapperLoadFaceTracker(char*
    configPath, char* paramsPath, char* trackerPath) {
    cv::Mat con = FACETRACKER::IO::LoadCon(configPath);
  globalParam = FACETRACKER::LoadFaceTrackerParams(paramsPath);

  FACETRACKER::FaceTracker* tracker =
     FACETRACKER::LoadFaceTracker(trackerPath);
  return tracker;
}

FACETRACKER::FaceTracker* WrapperLoadFaceTrackerDefault() {
  FACETRACKER::FaceTracker* tracker =
     FACETRACKER::LoadFaceTracker();
```

[6]https://github.com/ci2cv/face-analysis-sdk, Accessed: 2015-06-27, M. Cox, J. Nuevo, J.Saragih and S. Lucey, “CSIRO Face Analysis SDK“, AFGR 2013

```
    return tracker;
}
```

The WrapperLoadFaceTracker and WrapperLoadFaceTrackerDefault methods are responsible for initializing the face tracker. They are called once, when the prototype is started. The tracker parameters should be specified through the WrapperLoadFaceTracker function, but if there are no specifications available the default settings can be loaded via the WrapperLoadFaceTrackerDefault function.

```
FACETRACKER::FaceTrackerParams* WrapperLoadFaceTrackerParams()
   {
  return FACETRACKER::LoadFaceTrackerParams();
}
```

The WrapperLoadFaceTrackerParams method is used together with the WrapperLoadFaceTracker function and is responsible for passing the function calls to the actual script of the SDK which loads the additional tracking parameters.

```
void WrapperDeleteFaceTracker(FACETRACKER::FaceTracker*
   tracker) {
  delete tracker;
}
void
   WrapperDeleteFaceTrackerParams(FACETRACKER::FaceTrackerParams*
   param) {
  delete param;
}
```

When the prototype is stopped the memory that was reserved for the face tracker and the face tracker parameters has to be freed again. The WrapperDeleteFaceTracker and WrapperDeleteFaceTrackerParams methods are responsible for passing these calls on, once the script's OnDestroy() method is called.

```
int WrapperTrack(FACETRACKER::FaceTracker* tracker, unsigned
   char* image, int imageSize, int rows) {
  cv::Mat mat = cv::Mat(imageSize, 1, CV_8U, image);
  mat = mat.reshape(0, rows);
  return tracker->Track(mat, globalParam);
}
```

Once the tracker is initialized the WrapperTrack can be called every frame. The input grayscale image has to be passed on to the method, together with the FaceTracker, size and rows of the image.

It returns an integer value between 0 and 10 which represents the accuracy of the tracking. If 0 is returned, that means that the tracking possibly got lost and that the tracker should be reset.

```
void WrapperReset(FACETRACKER::FaceTracker* tracker) {
   tracker->Reset();
}
```

In this event the WrapperReset method can be used. In the following implementations a button was added which calls this function, should the optical feedback of the prototype show that the tracking is not accurately mapped to the face.

```
int WrapperGetShape(FACETRACKER::FaceTracker* tracker, float*
    points) {
   std::vector<cv::Point_<double> > shape = tracker->getShape();

   int pointCount = shape.size();
   for (int i = 0; i < pointCount; ++i)
   {
      points[i * 2] = (float)(shape[i].x);
      points[i * 2 + 1] = (float)(shape[i].y);
   }

   return pointCount;
}
```

Finally, the last method of the wrapper, which is called WrapperGetShape, retrieves the x and y values and inserts them into the passed on array.

It also returns an integer value which states how many feature points were tracked in the last WrapperTrack call. Usually the value is either 0, which means that something has gone wrong during the tracking, or 66 which is the exact number of face features.

A returned number between 0 and 66 very uncommon.

The Wrapper is contained in the libclmTracker.dll file. The following code snippet shows how to include all the required .dll files which are dependant on each other.

```
[DllImport("kernel32.dll")]
    private static extern System.IntPtr LoadLibrary(string
       dllToLoad);
    [DllImport("kernel32.dll")]
    private static extern bool FreeLibrary (System.IntPtr
       hModule);
    [DllImport("libclmTracker")]
    private static extern System.IntPtr
       WrapperLoadFaceTracker(string configPath, string
       paramsPath, string trackerPath) ;
    [DllImport("libclmTracker")]
    private static extern System.IntPtr
       WrapperLoadFaceTrackerDefault() ;
    [DllImport("libclmTracker")]
    private static extern void WrapperDeleteFaceTracker
       (System.IntPtr tracker);
    [DllImport("libclmTracker")]
    private static extern int WrapperTrack (System.IntPtr
       tracker, byte[] image, int imageSize, int rows);
    [DllImport("libclmTracker")]
    private static extern int WrapperGetShape(System.IntPtr
       tracker, float[] points);
    [DllImport("libclmTracker")]
    private static extern void WrapperReset (System.IntPtr
       tracker);
```

These DLL files have to be added in the same order as shown in the code snippet below, otherwise the code will probably fail and cause runtime errors. The path parameters need to be customized according on the structure of the Assets folder.

```
 private System.Collections.Generic.List<System.IntPtr> _DLLS =
    new System.Collections.Generic.List<System.IntPtr>();
 void Awake() {
 //Load
 _DLLS.Add (LoadLib(Application.dataPath +
    "\\Plugins\\FaceTracking\\x86\\libgcc_s_dw2-1.dll"));
 _DLLS.Add (LoadLib(Application.dataPath +
    "\\Plugins\\FaceTracking\\x86\\libstdc++-6.dll"));
 _DLLS.Add (LoadLib(Application.dataPath +
    "\\Plugins\\FaceTracking\\x86\\libopencv_core2410.dll"));
 _DLLS.Add (LoadLib(Application.dataPath +
    "\\Plugins\\FaceTracking\\x86\\libopencv_highgui2410.dll"));
 _DLLS.Add (LoadLib(Application.dataPath +
    "\\Plugins\\FaceTracking\\x86\\libopencv_imgproc2410.dll"));
 _DLLS.Add (LoadLib(Application.dataPath +
    "\\Plugins\\FaceTracking\\x86\\libopencv_objdetect2410.dll"));
 //_DLLS.Add (LoadLib(Application.dataPath +
    "\\Plugins\\x86_64\\libopencv_imgproc2411.dll"));
}
```

Once these steps have been completed correctly the rest of the scene needs to be set up like described in Section 4.7.2.

4.5 Simulating Finger Movement With the 5DT Data Glove Input

The 5DT Data Gloves are used to track finger flexure as well as finger abduction. Shown in Figure 4.5 are the 5DT Data Glove 5 Ultra on the left, and the 5DT Data Glove 14 Ultra on the right. The difference is that the 5 Ultra glove does not have the sensors required to track abduction between fingers, as well as the second flexure sensor responsible for the flexure tracking of the proximal interphalangeal joint. To cater to both versions of the glove the prototype provides the option to independently turn off abduction tracking as well as flexure tracking.

Figure 4.6 shows the location, name and abbreviation for each joint.

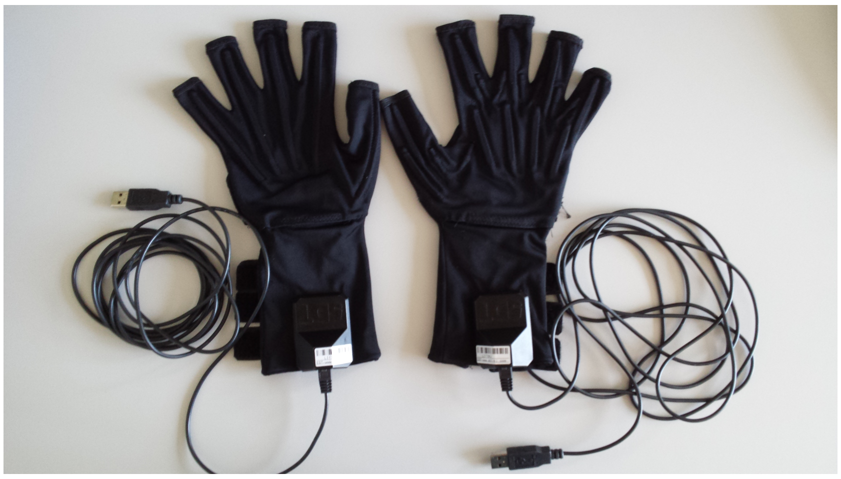
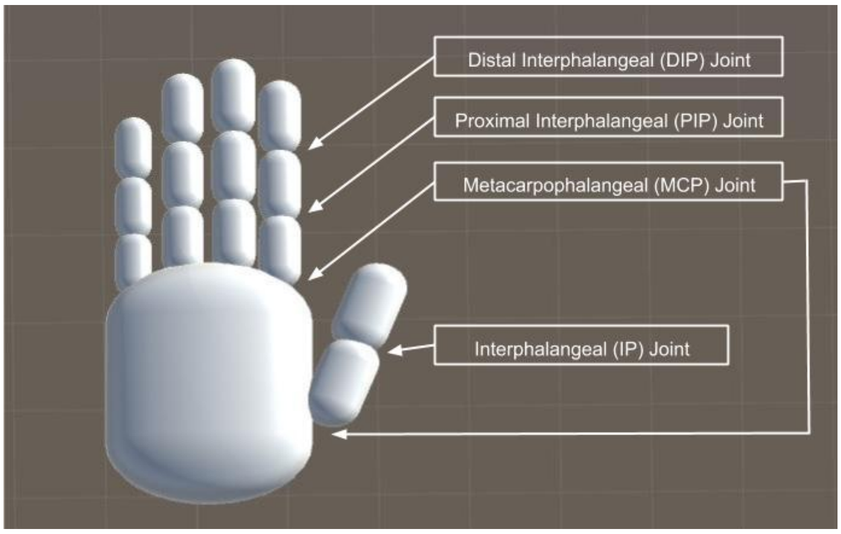


Figure 4.5: The 5DT Data Glove 5 Ultra (Left) and the 5DT Data Glove 14 Ultra (Right)


Distal Interphalangeal (DIP) Joint
Proximal Interphalangeal (PIP) Joint
Metacarpophalangeal (MCP) Joint
Interphalangeal (IP) Joint
Figure 4.6: Joint Names

As shown in Figure 4.7 and Table 4.1 the 5DT Data Glove 14 Ultra supports all possible joint movements for the thumb, but it does not provide sensors to track the flexure of the distal interphalangeal joint. This fact had to be considered in the implementation. To still be able to simulate a natural movement it was assumed that if the proximal interphalangeal joint is flexed the distal interphalangeal joint is flexed as well with the same angle. Despite this solution the prototype does not support every possible finger placement. But it does cover most of the natural looking poses.

Since the 5DT Data Glove 5 Ultra has only one sensor per finger to track the Metacarpophalangeal joint, the approach for the prototype is to apply the same flexure to all

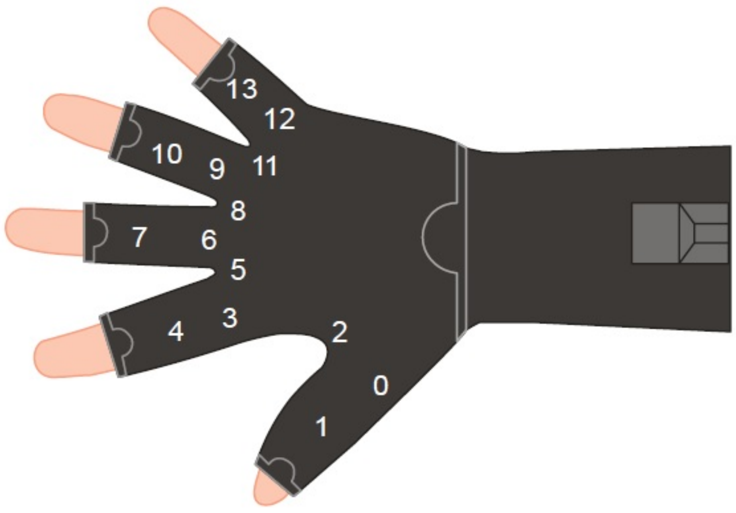


Figure 4.7: Sensor Mapping for the 5DT Data Glove 14 Ultra [13]

following joints of the same finger as well. To correctly apply the input data a simple sample model of a 3D human hand was created. Its hierarchical structure is constructed as follows:

- Wrist joint (Hand)
  - Thumb joint (MCP) -> Thumb joint (IP)
  - indexFingerJoint (MCP) -> indexFingerJoint (PIP) -> indexFingerJoint (DIP)
  - middleFingerJoint (MCP) -> middleFingerJoint (PIP) -> middleFingerJoint (DIP) -> HandTip
  - ringFingerJoint (MCP) -> ringFingerJoint (PIP) -> ringFingerJoint (DIP)
  - pinkyJoint (MCP) -> pinkyJoint PIP) -> pinkyJoint (DIP)

The test scene where the hand tracking prototype was developed can be seen in Figure 4.8. The whole scene consists of a left hand and a right hand for testing the tracking capabilities, and an additional right hand to test animation playback. The root node of every hand object contains two classes, HandController.cs and HandManager.cs.

The HandManager script contains a reference to every finger joint. It also has a checkbox to specify if the model is left or right handed and another one to enable abduction

Sensor	Driver Sensor Index	Description
0	0	Thumb flexure (lower joint)
1	1	Thumb flexure (second joint)
2	2	Thumb-index finger abduction
3	3	Index finger flexure (at knuckle)
4	4	Index finger flexure (second joint)
5	5	Index-middle finger abduction
6	6	Middle finger flexure (at knuckle)
7	7	Middle finger flexure (second joint)
8	8	Middle-ring finger abduction
9	9	Ring finger flexure (at knuckle)
10	10	Ring finger flexure (second joint)
11	11	Ring-little finger abduction
12	12	Little finger flexure (at knuckle)
13	13	Little finger flexure (second joint)

Table 4.1: Sensor Mapping for the 5DT Data Glove 14 Ultra [13]

tracking between the fingers. This feature should only be activated if the Data Glove has the corresponding sensors to track these features. The script also contains references to a few float variables, that can be used to set a desired bend rotation as well as the spread angle between two fingers. By default, the bend rotation is set to 90 degrees and works like euler angles. The rotation will be applied as negative rotation around the x-axis in the local space of the corresponding finger joint. The spread angle will only apply if abduction tracking is activated. It is set to a default value of 45 degrees for every abduction, expect for the the one between thumb and index finger, since this joint naturally as a wider angle than the others.

The script additionally contains three float variables x, y and z to fix potential problems with the rotation of the fingers. Since it is possible that the local rotation of the model's joints does not match the way they are manipulated in the animation process. Every variable again is representative for an euler angle and will be applied to the Metacarpophalangeal joint of each finger.

The HandManager also contains two other important functions which are both used together with the animation scripts described in Section 4.8.

The HandController script holds a reference to the HandManager. Through this reference it has access to the hand joints and all the parameters described above. It also

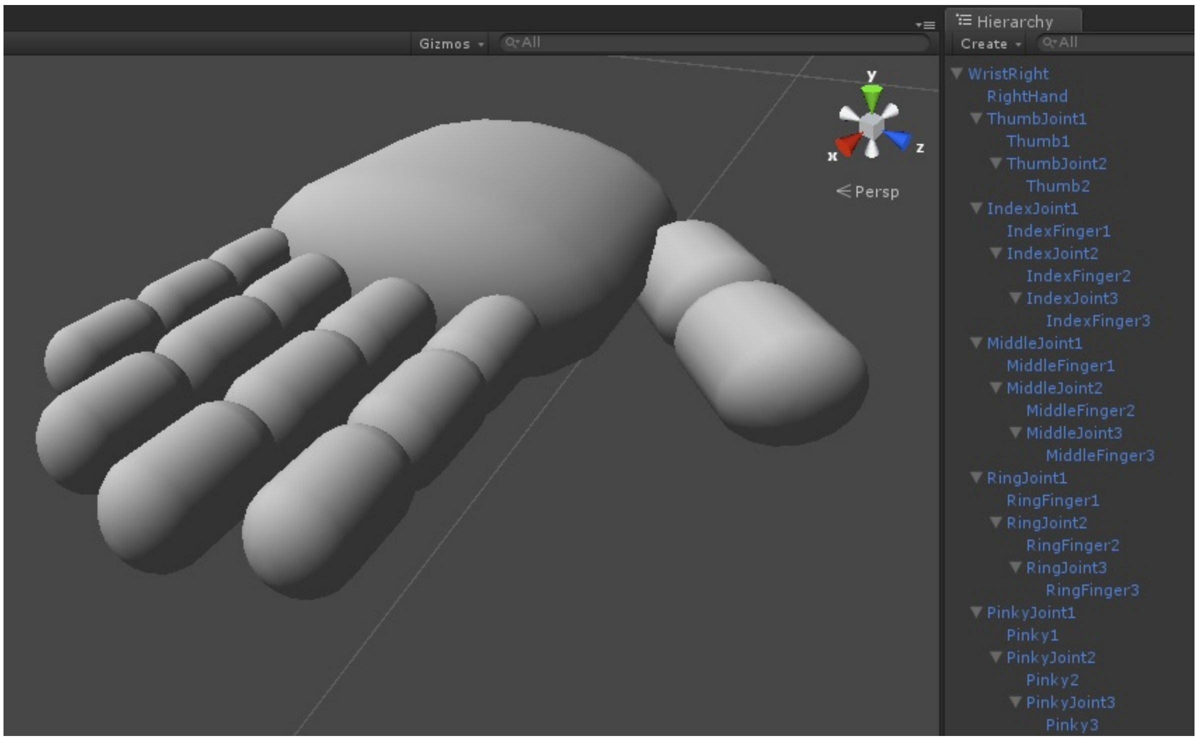


Figure 4.8: Hand Model With Structural Hierarchy

receives the glove input data via an instance of the CfdGlove class. In the Start() method of the script a new instance is created by calling the Open(pPort) function of the CfdGlove class and passing a string parameter containing "USB" plus the desired USB number. If a device is connected to the specified port, the script will call the animateHand(float angle) method for every frame, passing on the bendRotation held by the HandManager.

At the beginning of the animation process the sensor input is refreshed by calling the updateSensorInput() function, which calls the CfdGlove.GetSensorScaled(int index) function for every possible joint. The return values are temporarily stored in global float variables to enable access to them during the animation process.

The next step is to determine how the hand should be animated. This is done by looking at the settings stored in the HandManager script. The stored variables represent abductions, to decide if their abduction values should be considered during the animation. A checkbox for rightHand or leftHand lets the user influence how the abduction variables and bend rotations of the thumb joints are interpreted. Therefore it is not necessary to distinguish between right handed and left handed Gloves, since the checkboxes allow for independent interpretation of the flexure values. This means that a left handed glove could also be used for tracking right handed data if the rightHand check box is selected.

Once these steps are completed, the last step is to apply the sensor data to the corresponding finger joints. To get the correct angle for the rotation, the retrieved float values are multiplied with the passed on bendRotation.

After that, the HandController uses the HandManager to access its referenced GameObjects, representing the finger joints. For each GameObject its localRotation is set by calling Unity's Quaternion.Euler(x,y,z) function.

For the prototype it is assumed that the model's joints are oriented so, that the rotation around the x-axis will let the fingers move down. It is also assumed that there is no upward rotation of the fingers, since the 5DT Data Gloves only detect how much a finger is bent, but not if it is moved up or down. If abduction tracking is activated the y-axis will be set to the corresponding tracking data, otherwise it will be set to a fixed default value. The z-axis is not used because logically it is not possible for the finger joints to be pushed into each other.

4.6 Using Windows Kinect v2.0 for Body Animation

Since the documentation for the Kinect v2.0 SDK does not cover most of the special cases that occurred during the integration process into Unity, a lot of other sources (mostly forum entries) had to be used as support. However the "Kinect v2 with MSSDK" [7], which is available at the Unity Asset Store, covers a lot of the content this project is trying to achieve in terms of body animation.

Although none of its code was used for the implementation of this prototype, the idea of using a Dictionary together with an array of the bones to access them was adopted, as well as the expansion of the bone model by clavicular joints to be able to correctly map the shoulder rotations from the Kinect SDK.

4.6.1 Interpreting Motion Capture Data

To be able to interpret and use the Kinect input data correctly in Unity the following steps have to be performed:

1. First get body data from Kinect sensor. This can easily be achieved by using the BodySourceManager script which is available from the KinectView project in the Kinect Unity package described in Section 4.4.3.

2. Iterate through all available BodyJoints, the GameObjects referenced in the Body-Manager script. For each of these joints get the matching Kinect.JointOrientation.

3. Transform the JointOrientation object from a Kinect 4D Vector (Kinect representation of a Quaternion rotation) to an Unity Quaternion by using JointOrientation2Quaternion(), which takes three arguments. The first argument is the Kinect.JointOrientation object itself. The other two parameters are booleans, that determine if the X or the W component of the JointOrientation should be inverted or not. This is important to support the control flow of the avatar movement. If so desired, it is also possible to make the avatar move like it would be reflected by a mirror. This is achieved by exchanging the left and right extremities and converting the Quaternions to the right dimensions.

Each Kinect joint is depending on the rotation of its parent joint. For the mapping this means that a joint's rotation is representative for its parents rotation, for example the rotation of Kinect.JointType.AnkleLeft has to be applied to the models knee joint as well. To be able to map the bones correctly in step 2 the following map was included into the code:

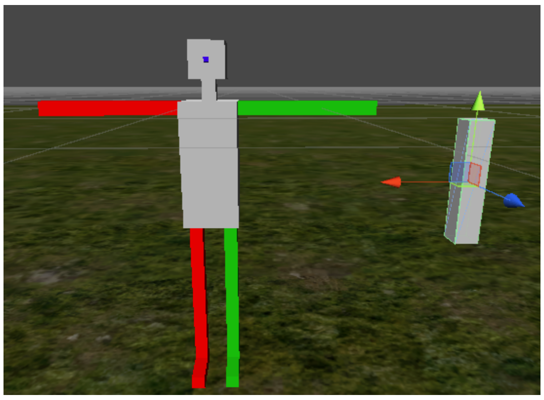
```
private Dictionary<Kinect.JointType, Kinect.JointType> _BoneMap
   =
new Dictionary<Kinect.JointType, Kinect.JointType>()
{
  { Kinect.JointType.AnkleLeft, Kinect.JointType.FootLeft },
  { Kinect.JointType.KneeLeft, Kinect.JointType.AnkleLeft },
  { Kinect.JointType.HipLeft, Kinect.JointType.KneeLeft },

  { Kinect.JointType.AnkleRight, Kinect.JointType.FootRight },
  { Kinect.JointType.KneeRight, Kinect.JointType.AnkleRight },
  { Kinect.JointType.HipRight, Kinect.JointType.KneeRight },

  { Kinect.JointType.HandLeft, Kinect.JointType.HandTipLeft },
  { Kinect.JointType.ThumbLeft, Kinect.JointType.HandLeft },
  { Kinect.JointType.WristLeft, Kinect.JointType.HandLeft },
  { Kinect.JointType.ElbowLeft, Kinect.JointType.WristLeft },
  { Kinect.JointType.ShoulderLeft, Kinect.JointType.ElbowLeft },

  { Kinect.JointType.HandRight, Kinect.JointType.HandTipRight },
  { Kinect.JointType.ThumbRight, Kinect.JointType.HandRight },
  { Kinect.JointType.WristRight, Kinect.JointType.HandRight },
  { Kinect.JointType.ElbowRight, Kinect.JointType.WristRight },
  { Kinect.JointType.ShoulderRight, Kinect.JointType.ElbowRight
     },

  { Kinect.JointType.SpineBase, Kinect.JointType.SpineBase},
  { Kinect.JointType.SpineMid, Kinect.JointType.SpineShoulder},
  { Kinect.JointType.SpineShoulder, Kinect.JointType.Neck },
  { Kinect.JointType.Neck, Kinect.JointType.Head },
  { Kinect.JointType.Head, Kinect.JointType.Head },
};
```

4.6.2 Applying Tracking Data to a 3D Model

This was the most complex issue in the whole body tracking process, since it was not easy to find a model that matches the requirements that are necessary to apply the tracking data correctly, even after the interpretation process.

Since the direction of each joint depends on its own rotation, as well as the orientation of the bones connected to it, applying an animation to a certain body part quickly turned into a very complex process. Incidentally, the best possible way to achieve proper mapping was to test each GameObject individually, until its correct rotation was found. This was done by applying the Kinect JointOrientation to the rotation of each transformation, but not to its localRotation parameter. Doing this had no big impact on the animation data, since all bones were still constricted by their bone hierarchy.

When a new object is created, it needs to be aligned along the y-axis, to make it possible to map the Kinect JointOrientation joint to the bone. After that they can be rotated in the desired direction. As an example, Figure 4.9 illustrates this process for the left arm of the prototype model. In this test scene a cube was created and its size was increased along the y-axis. After that the bone was rotated to the left to match the orientation of the joint data returned by the Kinect. Note the different coordinate systems in (a) and (b).


(a) Rectangle Oriented Along y-axis

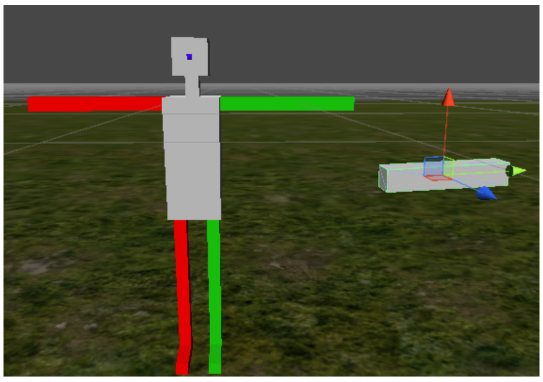

(b) Rectangle Rotated to Match Kinect v2.0 Input for Left Arm

Figure 4.9: Adaptation of Bone and Joint Rotation

4.7 Capturing Facial Feature Points With Webcam

At the time the prototype was built there was no SDK available for Unity, that would have been able to meet the needed requirements. Subsequently, two approaches have been made to achieve accurate and stable face tracking.

First, the attempt was made to write a custom face tracking implementation via OpenCV and include it into Unity with one of the introduced wrappers. However, as mentioned before, it soon turned out to be harder than expected to make the face tracking run stable enough with OpenCV alone.

The second approach was to include an external SDK into Unity, as detailed in Subsection 4.7.2.

4.7.1 OpenCV Approach

In the area of face tracking, feature extraction and feature mapping with OpenCV, there have been multiple approaches, like [53] and [23]. OpenCV gives its users so many liberties that it seemed to be a good fit with the development of this custom prototype. For this reason, it seemed reasonable to try to create a custom face tracking implementation at first. This section details the chosen development approach, and the reasons why the idea of a custom implementation was discarded at the end.

After including basic OpenCv-examples like [2] and [8], the resulting code was modified to fit the application requirements. Additional feature tracking filters were added and optimized. The result of this procedure can be seen in Figure 4.10.

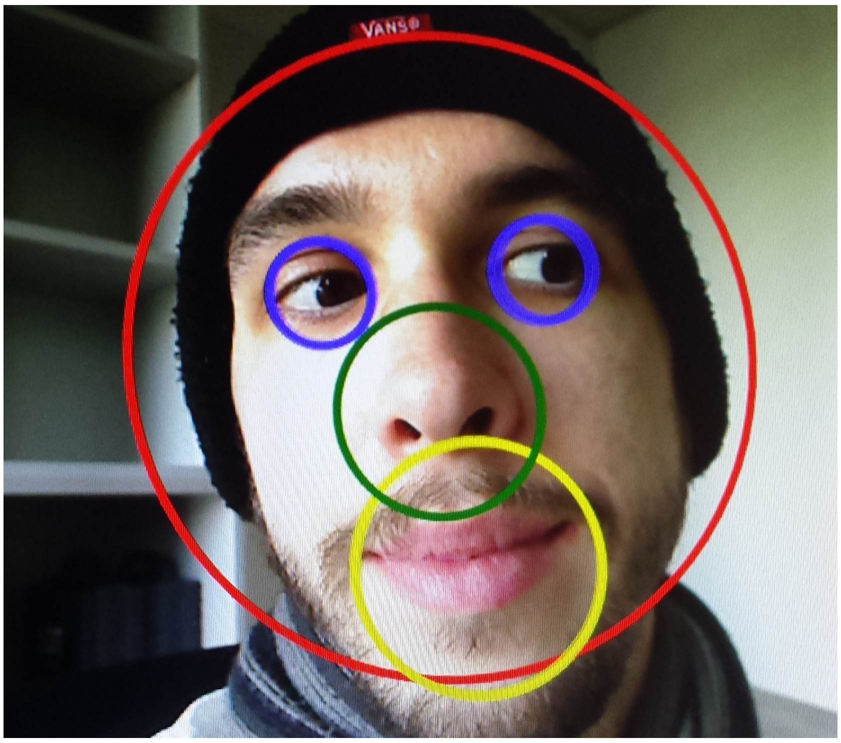


Figure 4.10: Face Feature Tracking With Haar Feature-based Cascade Classifiers

However, the tracking process was still not stable enough for live tracking. Especially in an unevenly illuminated environment. The next steps were to improve illumination, extract Face Feature Points and match them to the face mesh.

In the end the Haar-Cascade files where not stable enough without sufficient training. So the approach was discarded and a professional, more suitable SDK was chosen for the project.

4.7.2 External SDK: CSIRO Face Analysis SDK

As a prerequisite for this face tracker, the following GameObjects needed to be added to the scene:First, the FaceSourceManager, which can hold a reference to a webcam display, for showing the input image from the webcam, and a reference to a Unity Mesh Renderer. The Mesh Renderer uses a material with a custom grayscale shader and converts the webcam input into a grayscale image for processing, for the face tracking.

The FaceRoot needs to be placed in the middle of the face, where the face feature points should be displayed. For animation capturing the FaceManager and the FaceControl scripts must be attached to it.

Unlike the BodyManager described in the previous subchapters, the FaceManager of the prototype does not contain references to any joints. It creates the feature points for the animation at runtime and sets their position according to the tracking data received from the FaceSourceManager. The data is applied to the FaceManager by the FaceControl script, which has a few public variables that have to be specified before playback: The FaceFeatureCount needs to be set to 66, since that is the exact number of features returned by the face tracking SDK. The FaceRoot has to match the created feature points. Optionally, the Head parameter can be used to implement a method for rotating the model's head. The FeatureSize is a 3D Vector, which determines how big the features will be displayed on the screen. By default it has a size of 0.02 FeatureColor is an optional parameter and can be used to apply a material to the feature points. The prototype uses a blue colored material for the face features.

Additionally to the FaceManager and the FaceSource reference, the FaceControl script has some specific parameters that need to be set: GlobalFaceSizeX and GlobalFace-SizeY are parameters that enable the prototype to consider the global size of the face when mapping the features of the tracked face to the FaceRoot. These parameters are necessary, since Unity does not know an object's size in relation to the world anymore, when it is part of a hierarchy. The Prefab parameter specifies the GameObjects which are used to display the referenced face features on the webcam display in the top right corner of the test scene. A snapshot of this can be seen in Figure 5.10. The prototype does not calculate depth values for the face feature points, but provides a DepthValues parameter which can be used to adjust depth values manually. It contains an array for all of the 66 possible feature points.

Should the scale or position of the face still not match the model, despite all configurations, it can be adjusted to the size of the avatar by simply changing the size or the position of the FaceRoot by using the editor window. These changes can be made at runtime, before starting the recording.

4.8 Exporting and Loading Animation Data

Since Unity does not provide the option to export animation data at runtime just by memorizing the transform rotations of each component, it was necessary to implement this part of the application as an individual component.

4.8.1 Data Format

To store the animation data in a way that would make it reusable, a specific file format was created that uses the Unity Quaternion representation to save the animation's transformation rotations. In this file format, each main tracking part (hand, body, face) has its own feature vector, which contains all required values to animate the corresponding region.

The format also contains time stamps for each frame, that mark the absolute time which has passed since the start of the recording process. These timestamps might also be used for queuing, if trying to load the animation progressively, but for the current prototype the animation file needs to be fully loaded into the main memory before the playback starts.

Since not every animation file must necessarily consist of each of the three tracking input sources, each data block is marked by an individual starting mark. A complete list of all marker tokens can be found in Table 4.2.

In this data model it is assumed that every joint that is supported by the prototype is tracked. For this reason, there is an exact amount of separators for each marker. For example, the seventh separator in the "hr" section always represents the second joint of the middle finger on the right hand. However, there are a few exceptions that had to be taken into account.

Token	Usage	Context
"time:"	Calculate time difference	retrieved via Time.time
"root:"	Position of the root element (SpineBase)	
"hr"	Hand tracking right hand	5DT Data Gloves
"hl"	Hand Tracking left hand	5DT Data Gloves
"f"	Face Tracking	Webcam
"b"	Body Tracking	Kinect v2.0
"{"	Begin tracking data section	after "hr", "hl", "f" or "b" marker
"}"	End tracking data section	after last Quaternion entry
"*"	error in getTrackingData function	return value
";"	Seperator	between Quaternion entries

Table 4.2: Vocabulary for the Animation Data File Format

4.8.2 Saving Animation Data

For the hand tracking it was important to distinguish between left and right hand data because of the directions of the abduction Quaternions. For example, spreading the middle finger apart from the other fingers on the right hand is a movement in the opposite direction on the left hand. The flexure of each finger is not affected, because it can be interpreted independently. For this reason two markers "hr" and "hl" are needed so the animation data can be interpreted correctly.

As shown in Table 4.3 the feature vector of the hand tracking data contains all entries listed in Table 4.1 and Figure 4.7. But it also contains additional tracking data about the distal interphalangeal joints for the index, middle, ring and little finger. As mentioned before, these joints are assumed to bend to the same degree as their corresponding proximal interphalangeal joints.

The body tracking data also has some special characteristics. Since it is not sure that every body model possesses all of the currently possible 27 joints, which are provided by the Kinect v2.0, it had to be considered that there might be empty connections, from

Feature Number	Name
0	Wrist joint (root bone for hand animation)
1	Metacarpophalangeal (MCP) joint, thumb
2	Interphalangeal (IP) joint, thumb
3	MCP joint, index Finger
4	Proximal interphalangeal (PIP) joint, index finger
5	Distal interphalangeal (DIP) joint, index finger
6	MCP joint, middle finger
7	PIP joint, middle finger
8	DIP joint, middle finger
9	MCP joint, ring finger
10	PIP joint, ring finger
11	DIP joint, ring finger
12	MCP joint, little finger
13	PIP joint, little finger
14	DIP joint, little finger

Table 4.3: Feature Vector Hand Animation Data

where no Quaternions could be obtained. In the first prototype each body joint got assigned its own marker token, like "head:", "rightKnee:", "leftWrist:" and so on.

This approach worked, but it was an unhandy solution because it made the parsing process slower and unnecessarily difficult. To improve the parsing speed, the feature vectors for empty connections are now filled with dummy Quaternions, that do not affect the rotation.

Table 4.4 shows the feature vector for the body tracking. It contains every joint provided by the Kinect v2.0 SDK and an additional root parameter. This parameter is an optional feature and contains information about where the model was placed in the scene and what its orientation was at the time the animation was recorded. It can be used to reset the animated character to its exact initial position and/or rotation. However, this parameter was mostly kept for debugging reasons.

The feature vector contains no extra fields for the abduction values since they are also contained in the Quaternion rotations, if abduction tracking is activated.

Feature Number	Name
0	Body root node
1	Head
2	Neck
3	Spine Shoulder
4	Spine Mid
5	Spine Base
6	Clavicle Left
7	Shoulder Left
8	Elbow Left
9	Wrist Left
10	Hand Left
11	Clavicle Right
12	Shoulder Right
13	Elbow Right
14	Wrist Right
15	Hand Right
16	Hip Left
17	Knee Left
18	Ankle Left
19	Foot Left
20	Hip Right
21	Knee Right
22	Ankle Right
23	Foot Right
24	Hand Tip Left
25	Hand Tip Right
26	Thumb Left
27	Thumb Right

Table 4.4: Feature Vector Body Animation Data

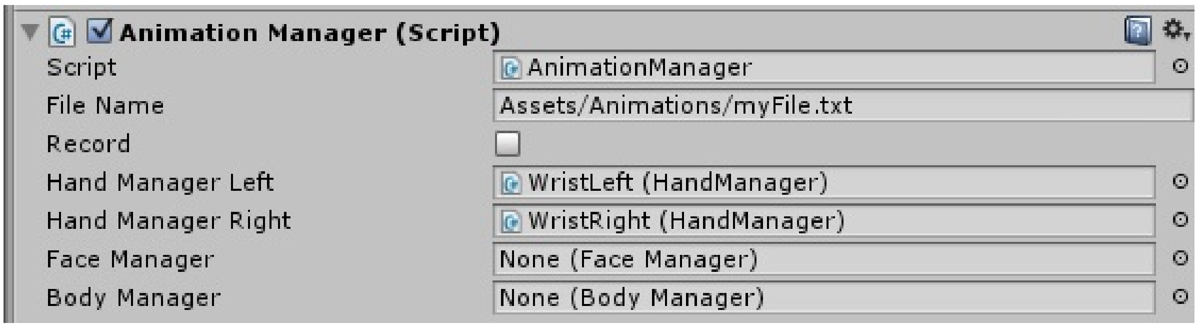
Since the face features are stored as 3D vectors instead of Quaternions the face feature vector is slightly different from the body and hand feature vectors. Furthermore, the getFeatures() - method of the FaceManager only returns a predefined set of GameObjects. Their position can be retrieved from the transform of each element. The feature vector holds all 66 feature points provided by the face analysis SDK. The mapping of the feature points can be found in the official SDK documentation[7].

[7] `http://face.ci2cv.net/doc/#sec-4-3`, Accessed: 2015-07-02

Finally, to be able to create and store these feature vectors in a file an AnimationManager script was added to the project. This script can hold a reference to each possible input source that is supported by the defined data format. The different sources are:

- Left Hand (5DT Data Glove)
- Right Hand (5DT Data Glove)
- Face (Webcam)
- Body (Kinect v2.0)

To make the script easier to use all the essential configurations were added to it as public variables. The script can simply be added to a GameObject and from there it is possible to configure the settings. Figure 4.11 shows a snapshot of how the script is displayed in the Unity Inspector.



Animation Manager (Script)

Script	AnimationManager
File Name	Assets/Animations/myFile.txt
Record	☐
Hand Manager Left	WristLeft (HandManager)
Hand Manager Right	WristRight (HandManager)
Face Manager	None (Face Manager)
Body Manager	None (Body Manager)

Figure 4.11: Interface of the AnimationManager

The location and filename is stored in a string variable and can be specified via the File Name field. If the Record button is checked the AnimationManager will write the animation data of the referenced ManagerObjects to the specified file. The Manager-Objects can be added to the script by dragging the GameObject, which contains the matching manager script, onto the corresponding field.

The AnimationManager will then call the getTrackingData() method every frame, for all referenced manager scripts. This method uses the getFeatures() method of the manager script and stores the string representation of these features individually for each current frame. The code below shows what this function looks like in the HandManager script:

```
public Quaternion[] getFeatures(){
  Quaternion[] features = new Quaternion[15];
  features[0] = Base.transform.rotation;

  features[1] = Thumb1.transform.rotation;
  features[2] = Thumb2.transform.rotation;

  features[3] = IndexFinger1.transform.rotation;
  features[4] = IndexFinger2.transform.rotation;
  features[5] = IndexFinger3.transform.rotation;

  features[6] = MiddleFinger1.transform.rotation;
  features[7] = MiddleFinger2.transform.rotation;
  features[8] = MiddleFinger3.transform.rotation;

  features[9] = RingFinger1.transform.rotation;
  features[10] = RingFinger2.transform.rotation;
  features[11] = RingFinger3.transform.rotation;

  features[12] = Pinky1.transform.rotation;
  features[13] = Pinky2.transform.rotation;
  features[14] = Pinky3.transform.rotation;

  return features;
}
```

Note that in this implementation the Base variable is the Unity GameObject which holds the reference to the root joint. It is equivalent to the wrist joint of the body tracking process returned by the Kinect, therefore it is only needed if the hand tracking is done without body tracking.

An example of the animation data for one captured frame is given below:

```
time:2.17249
root:(0.07061455, -0.23192610, 1.44451500)
hl{(-0.05069533, -0.35007570, 0.41551870,
   0.83798670);(0.58886610, 0.21946390, 0.34244610, 0.69842930);
   ... }
hr{(-0.34291020, -0.26722300, 0.69032680,
   0.57832010);(-0.47619380, -0.12531200, 0.77615680,
   0.39384990); ... }
f{(0.00531995, -0.02025503, 0.00000000);(0.00487498,
   0.01646760, 0.00000000); ... }
b{(-0.02991204, 0.94905940, -0.09112739,
   -0.30014610);(-0.02991204, 0.94905940, -0.09112739,
   -0.30014610);(-0.03580621, 0.93773010, -0.09231110,
   -0.33295510); ... }
```

It shows a snapshot of the captured animation data of one frame as captured by the prototype. The format is organized in the following order:

- time: for the time passed in seconds since the prototype was started.
- root: the current position of the root node, which in this example is the spine base joint of the model
- hl... for the left hand, with the previously explained feature vector for the hand tracking
- hr... for the right, also with the hand feature vector
- f... for the face tracking data. As mentioned before, the entries are 3D Vectors instead of Quaternions and only represent the position of the face features.
- b... represents the body tracking data. Joints that are not tracked are filled with empty Quaternions.

During playback all entries are interpreted as float values. They are stored with eight digits by default. If higher or lower accuracy is desired the number of digits can be specified in the AnimationManager scripts.

Each marker token is interpreted individually, so if for any reason the tracking of an input source fails it has almost no effect on the animation file. The position of the model simply stays the same until the tracking from the failed source can be resumed. The following examples demonstrate how the captured animation data changes when certain gestures are made while capturing.

Hand Tracking Data Example

Flat Hand

```
time:4.59343
hr{(0.00000000, 0.00000000, -0.70710690,
   0.70710680);(-0.13360840, 0.39066280, -0.42435270,
   0.80588850);(-0.57123850, 0.56331650, -0.12396470,
   0.58394710);(0.01196110, 0.70700570, -0.01196110,
   0.70700570);(-0.48917750, 0.51059310, 0.48917760,
   0.51059320);(-0.70686930, 0.01832482, 0.70686930,
   0.01832481);(0.00152866, 0.70710520, -0.00152866,
   0.70710520);(-0.46338300, 0.53411270, 0.46338300,
   0.53411280);(-0.70024610, 0.09826287, 0.70024610,
   0.09826288);(0.00082776, 0.70710640, -0.00082776,
   0.70710640);(-0.49941440, 0.50058510, 0.49941450,
   0.50058510);(-0.70710650, 0.00082776, 0.70710650,
   0.00082776);(0.08160488, 0.70238210, -0.08160488,
   0.70238210);(-0.17296770, 0.68562540, 0.17296770,
   0.68562540);(-0.40502400, 0.57961680, 0.40502400,
   0.57961680);}
```

Fist

```
time:5.052942
hr{(0.25879680, -0.84076630, 0.04211672,
   0.47366960);(-0.18032550, -0.73440750, -0.09367839,
   0.64757490);(-0.49766060, -0.57153880, -0.47060960,
   0.45188980);(-0.52707860, -0.33480530, -0.02366283,
   0.78072670);(-0.92093590, -0.22509930, -0.24896750,
   0.19804800);(-0.80143090, 0.01008493, -0.33548890,
   -0.49502950);(-0.53945930, -0.33438550, -0.02899783,
   0.77222370);(-0.91489220, -0.23213070, -0.24242490,
   0.22431700);(-0.83547850, -0.01447018, -0.33532830,
   -0.43511080);(-0.54078280, -0.33433520, -0.02957126,
   0.77129750);(-0.92702400, -0.21667790, -0.25633020,
   0.16724970);(-0.77622300, 0.02650484, -0.33459220,
   -0.53368890);(-0.45839600, -0.33560250, 0.00504987,
   0.82293320);(-0.73356120, -0.31300200, -0.12117850,
   0.59096030);(-0.90164730, -0.24471220, -0.22971810,
   0.27272400);}
```

The first thing to be noticed in this comparison is that some quaternions, for example the first one, hold values near or equal to zero when a flat hand is recorded. The values change drastically when a fist is formed. This is because the initial position of the hand joints is the flat hand. When forming a fist the fingers are bent and a negative rotation towards the palm of the hand is applied to the joints.

Body Tracking Data Example

Body in T-position

```
time:2.788506
root:(0.00000000, 0.00000000, 0.00000000)
b{(0.00000000, -0.00000033, 0.00000000,
   -1.00000000);(0.00000000, 1.00000000, 0.00000000,
   -0.00000016);(0.00000000, 1.00000000, 0.00000000,
   -0.00000016);(0.00000000, 1.00000000, 0.00000000,
   -0.00000016);(0.00000000, 1.00000000, 0.00000000,
   -0.00000016);(0.00000000, 0.00000000, 0.70710690,
   0.70710680);(0.00000000, 0.00000000, 0.70710690,
   0.70710680);(0.00000000, 0.00000000, 0.70710690, 0.70710680);
   ... }
```

Body is Recognized by the Kinect Sensor

```
time:2.844301
root:(0.01320315, -1.01690200, 2.71933200)
b{(-0.00457114, 0.94712690, -0.17332110,
   -0.26998110);(-0.00457114, 0.94712690, -0.17332110,
   -0.26998110);(-0.02046368, 0.89589760, -0.23083320,
   -0.37903170);(-0.01640655, 0.95615300, -0.14715380,
   -0.25268200);(-0.01130806, 0.99048640, -0.06972072,
   -0.11810090);(-0.43366350, 0.70312290, 0.51666250,
   0.22497590);(-0.59852040, 0.51073230, 0.60890510,
    0.10079960);(-0.77742050, 0.23510120, 0.53364530,
    0.23572810); ... }
```

This example shows how the values of the body tracking change when the Kinect sensor recognizes the performing actor in two consecutive frames. The first data set shows the model in its initial T-position. After recognizing the actor the model jumps to the relative position and rotation. This is why the position of the root element and the rotation of the body joints change so drastically. Of course, only a representative selection of the tracked body joints are shown in this example, since the data set would consume too much space.

Face Tracking Data Example

Mouth Opened, Grinning

```
time:7.86412
f{ ... (-0.01371785, 0.00098656, 0.00000000);(0.01709121,
    0.00571176, 0.00000000);(0.04596011, -0.00100981,
    0.00000000);(0.06547271, 0.02471725, 0.00000000);(0.07359099,
    0.05573834, 0.00000000);(0.05986268, 0.07775412,
    0.00000000);(0.04181895, 0.09604469, 0.00000000);(0.01709826,
    0.10589030, 0.00000000);(-0.00909295, 0.09692510,
    0.00000000);(-0.02936864, 0.07849124,
    0.00000000);(-0.00985637, 0.03156788,
    0.00000000);(0.01696140, 0.03108522, 0.00000000);(0.04127914,
    0.02983860, 0.00000000);(0.03916199, 0.05202315,
    0.00000000);(0.01733847, 0.06481667,
    0.00000000);(-0.00701104, 0.05341836, 0.00000000);}
```

Mouth Closed

```
time:8.50118
f{ ... (-0.01176107, 0.00427570, 0.00000000);(0.01872471,
    0.00870007, 0.00000000);(0.04733468, 0.00171613,
    0.00000000);(0.06736776, 0.02654896, 0.00000000);(0.07676663,
    0.05669533, 0.00000000);(0.06203578, 0.07639766,
    0.00000000);(0.04363808, 0.09244032, 0.00000000);(0.01971341,
    0.10149400, 0.00000000);(-0.00587023, 0.09383619,
    0.00000000);(-0.02674355, 0.07803033,
    0.00000000);(-0.00768339, 0.03478444,
    0.00000000);(0.01894353, 0.03430741, 0.00000000);(0.04330693,
    0.03255747, 0.00000000);(0.04127171, 0.05004216,
    0.00000000);(0.01963017, 0.06167261,
    0.00000000);(-0.00456566, 0.05190928, 0.00000000);}
```

In this last example representative face tracking points of the mouth are shown. The other features are not included since the contours of the face do not change much during capturing, which makes sense considering that the face is tracked from a fixed position. The first vector shown in this dataset represents the right corner of the mouth. The first value is the x-coordinate and the second is the y-coordinate. When the mouth is open the x-value is further away from the center of the face but the y-value is closer, because the corners of the mouth goes up and apart. The values for z are zero because they are optional and can be modified via the scripts in the editor.

4.8.3 Loading Animation Data

Each manager class was extended by a setRotations function. This function is used together with the AnimationReader script. Each part of the model (left/right hand, body or face) that is going to be animated must have its own AnimationReader script attached to it. The AnimationReader script needs to hold a reference to the manager script of its GameObject, so that the animation data can be passed to the manager script every frame.

The following code snippet shows the setRotation function for the hand animations. This function works similarly for the hand and body animation scripts. Only the setRotation method for the face animations looks differently, since instead of joints, the face animations are defined by the position of their feature points, which are stored in 3D vectors and not in Quaternions.

The code below shows the setRotation function which is used together with the HandManager script:

```
public void setRotations(Quaternion [] r){
        if (r.Length != 15)
                return;


        interpolate (r);

        Thumb1.transform.rotation = newFrame [1];
        Thumb2.transform.rotation = newFrame [2];

        IndexFinger1.transform.rotation = newFrame [3];
        IndexFinger2.transform.rotation = newFrame [4];
        IndexFinger3.transform.rotation = newFrame [5];

        MiddleFinger1.transform.rotation = newFrame [6];
        MiddleFinger2.transform.rotation = newFrame [7];
        MiddleFinger3.transform.rotation = newFrame [8];

        RingFinger1.transform.rotation = newFrame [9];
        RingFinger2.transform.rotation = newFrame [10];
        RingFinger3.transform.rotation = newFrame [11];

        Pinky1.transform.rotation = newFrame [12];
        Pinky2.transform.rotation = newFrame [13];
        Pinky3.transform.rotation = newFrame [14];

}

private void interpolate(Quaternion [] r){

        for (int i = 0; i<15; i++) {
                newFrame[i] = Quaternion.Slerp(lastFrame[i],
                                        r [i],
                                           smoothFactor*Time.deltaTime);
        }

                lastFrame = newFrame;
```

```
}
```

As mentioned before, the animation file needs to be fully loaded before replay starts. Consequently, a preloaded set of Quaternions is passed to the setRotations function every frame. However, if these Quaternions are applied to the model without prior interpolation, the animation will appear choppy and clipped. For this reason the interpolate function is called every frame before the animation data is passed to the transform rotations.

The interpolation is performed by using Unity's Quaternion.Slerp function with a predefined smooth factor on the data from the previous frame and the current frame. Subsequently, the computed frame is stored as the new previous frame for the next iteration circle. Only after that the new frame data is applied as the new joint rotations. At the beginning of each iteration process the received data is compared to the size of the feature vector of the corresponding model as an additional precaution to ensure that the animation process runs smoothly.

4.9 Issues and Adaptation

Because of the number of different technologies and input sources used to create this prototype, many conflicts and problems arose during the development process. This section will cover the biggest complications, which needed to be solved during the development of the prototype. Some of these problems had direct consequences for the implementation, but there were also some compatibility issues with the hardware.

4.9.1 Kinect Related Issues

Sometimes an error occurrs with the Kinect, where it reports, that there are not enough USB controller resources. This is a problem that can occur when a USB hub is used, that operates with a power source which is not strong enough.

4.9.2 Bit Version Conflicts

At the beginning the prototype was developed with UNITY 4.6.5, which requires the pro version of Unity to be able to include and load external libraries. This changed with the release of Unity 5.

Unfortunately, the .dll files provided by the 5DT Glove SDK were only available in a x32 precompiled version. Testing proved that these files functioned correctly with the x32 bit version of Unity 5, and as a consequence the whole prototype was forced to run in 32bit mode instead of 64bit. Needless to say, that meant that the .dlls of the included SDKs needed to be x32 versions as well. For example, the OpenCVSharp .dlls needed to be recompiled for 32bit.

However, after contacting the 5DT support they provided a 64bit build of the .dll files and the project was rebuilt as a x64 version. Unfortunately, the face tracking had presently still unresolved issues in 64bit mode, so in the end the project had to be changed back to x32 again.

4.9.3 Unity Build Settings

For the current version of the prototype it is not possible to build an executable which runs outside of the Unity editor. If the project is built anywhere else, the dependencies for the face tracking component seem to get lost and the .dll files are not loaded correctly.

However, this only concerns the face tracking module. When it is removed from the project the build with only hand and body tracking works correctly.

4.10 Quick Start Guide

The easiest way use the prototype is by importing it as a Unitypackage. The only other prerequisites are that the previously mentioned system requirements are met and that Kinect v2 for Windows SDK is installed.

After importing the package the project structure should look like in Figure 4.12.

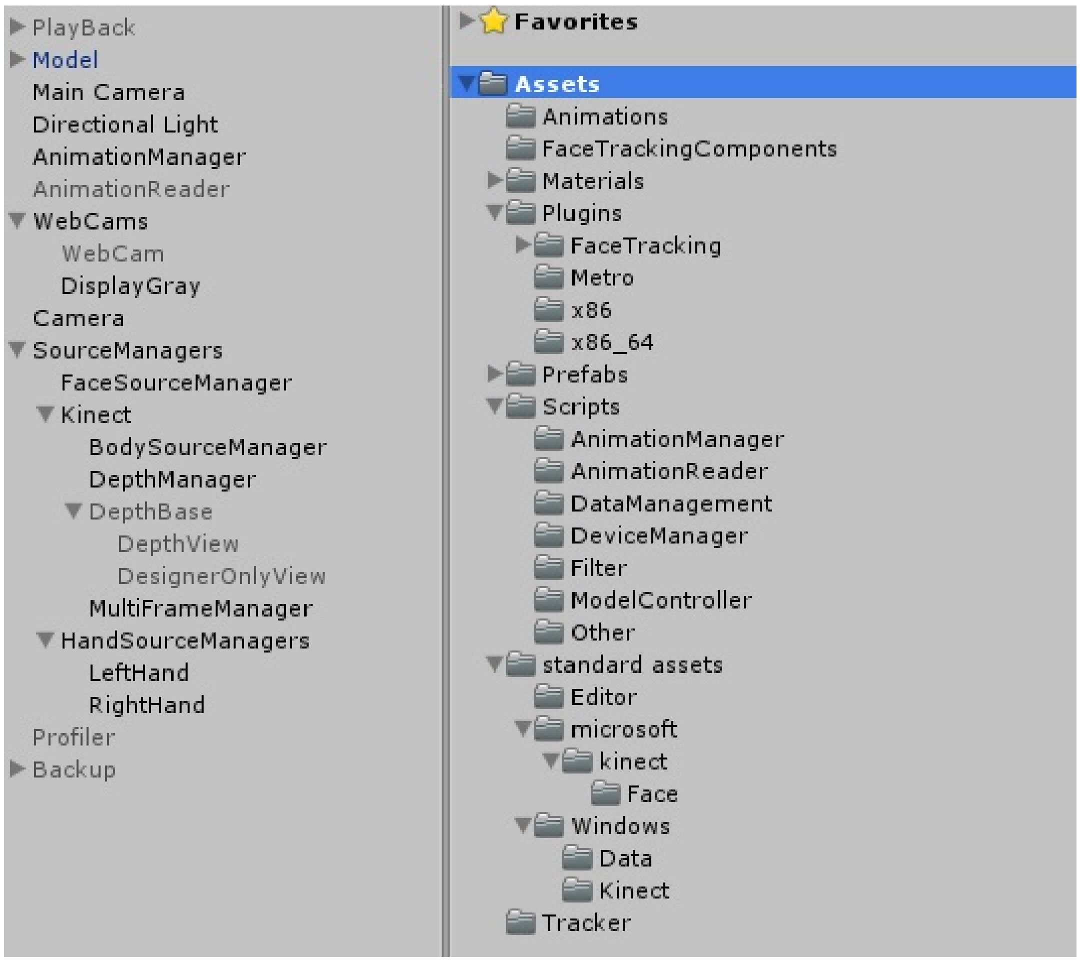


Figure 4.12: Project Hierarchy

The left column shows the objects in the scene. The PlayBack object is the demo model that works together with the AnimationReader object to load and play animation files. They are both disabled by default, but can be activated in the inspector window of the Unity editor. The Model object is used for displaying the tracked animations. They can be recorded by using the AnimationManager object, which holds references to the Model's manager scripts for retrieving the animation data.

There are two cameras in the scene. The MainCamera is used for filming the model during playback and the other Camera shows the helper screen for the face tracking.

The SourceManagers object contains all GameObjects with the scripts for controlling the input devices attached to them. The Kinect object also contains the DepthManager and the MultiFrameManager which can be used to configure the Kinect tracking. Also, the input from the Kinect's depth camera can be displayed via the DepthBase object, but this can impact the runtime performance. For this reason this feature is turned off by default. The other objects, Profile and Backup, are used for debugging.

The right column shows the assets folder of the prototype. The folder structure is organized as follows:

- Animations: default location for recorded animations
- FaceTrackingComponents: prefabs and shader used for face tracking
- Materials: materials for the demo model
- Plugins: the .dll files for Kinect, 5DT Data Gloves and the CSIRO Face Analysis SDK
- Prefabs: predefined models of the body, hands and the complete demo model
- Scripts: scripts, as described in the component diagram
- Standard assets: scripts imported via the Kinect SDK
- Tracker: specification files for the face tracking SDK

5 Results and Evaluation

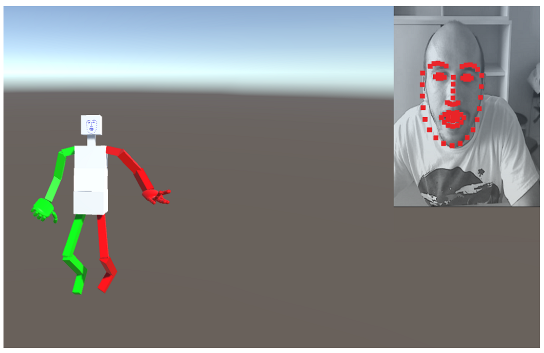
At the beginning of this section an overview of how well the body, face and hand tracking work together to generate synchronized full body animation data will be given. After that in this section the prototype will be evaluated in terms of performance and animation quality. All the evaluations have been performed on the same computer. The tests have also all used the same Unity scene. For evaluating the single components in section 5.2, all other input sources were excluded before the test was started.

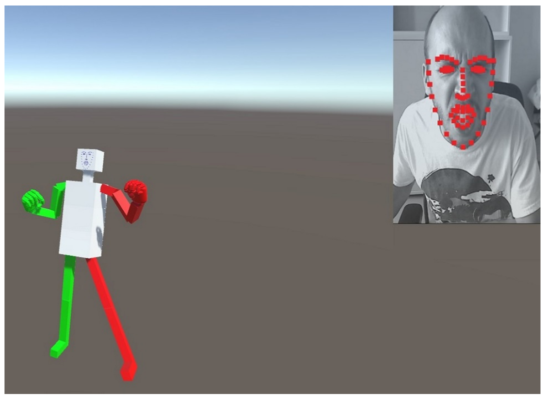
The animation quality analysis is not based on a statistical analysis. It just points out general aspects that influence the animation output.

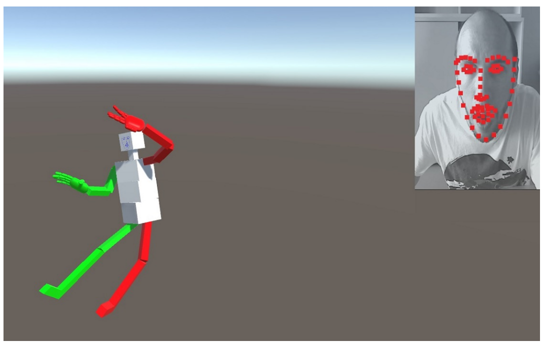
5.1 Animation Capturing

Since the prototype can track the facial expressions and detailed finger movements of an actor, while simultaneously recording his body movements, there is a broad variety of possible animations that can be recorded. Some example animations are shown in Figure 5.1.

The actor can move freely inside the field of view of the Kinect sensor. Walking around and jumping is also not a problem, since the position of the root node is retrieved from the Kinect.SpineBase bone and applied to the model. Different kinds of emotions such as joy, fear and anger can be animated by recording the body language, detailed finger movement and the actor's facial expressions. It is also possible to capture more complex motions like dancing or fighting moves. Tracking errors can still occur, but recording the animations and correcting errors afterwards is still considerably more efficient than manually creating such detailed animations.


(a) Jumping


(b) Angry


(c) Afraid

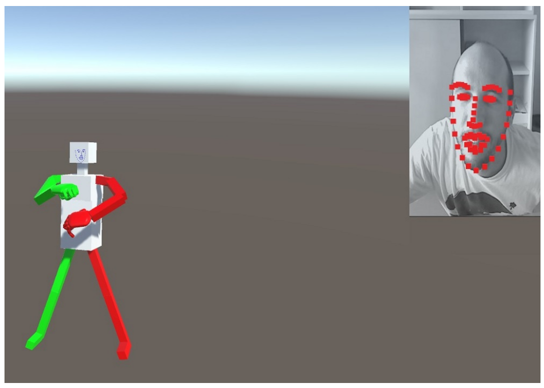

(d) Dancing

Figure 5.1: Capturing Examples

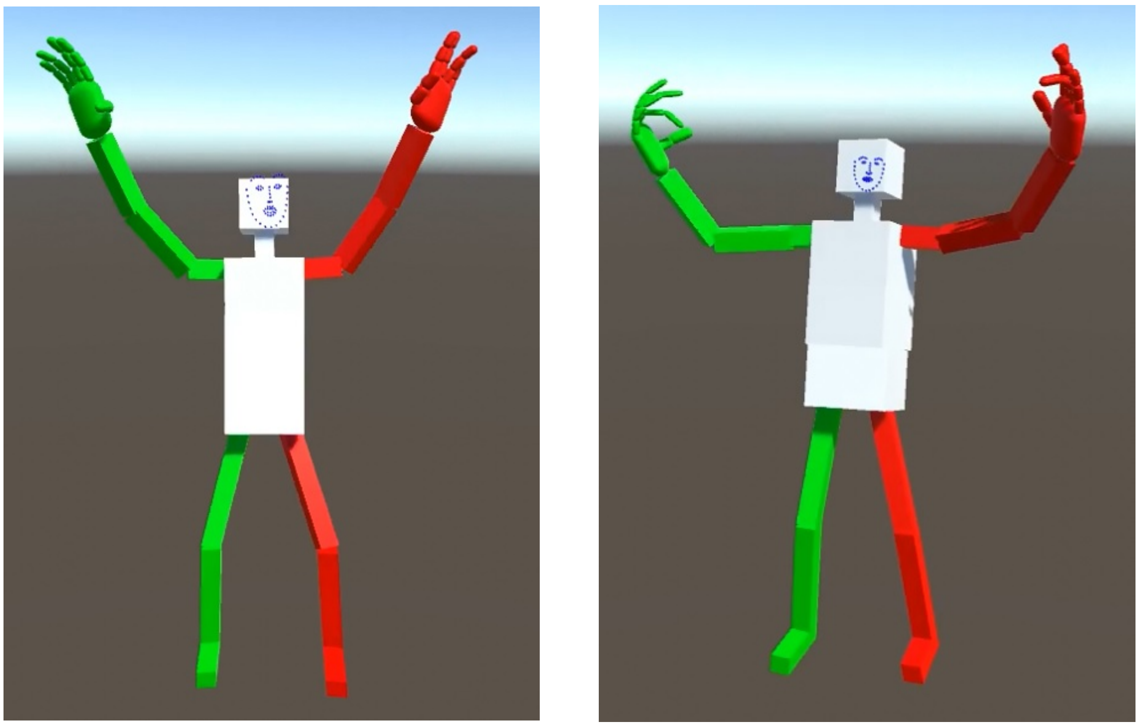
The tracking process can be initiated at the same time the prototype is started, but it takes some time to calibrate the input sources. It is highly recommended to do so before recording any animations, because otherwise some input sources might not be tracked at all or might have a heavy impact on the run time performance or the animation quality.

The 5DT Data Gloves sometimes initially produce random values, but after making fists and opening the hands again this issue disappears.

It takes some time for the Kinect sensor to recognize the body of the actor. The tracking process can be supported by taking a T-position or slowly moving the arms up and down. It is also important that no other person is in the field of view of the sensor while recording, because otherwise it is possible that the other person is tracked as well. This results in tracking loss of the actor, and the system then tracks the movements of the second person.

When the face tracking is started, at the initial search for the face or when the tracking of the face is lost, run time performance is greatly impacted. Once the face is recognized and the feature points are displayed correctly the run time performance increases to an average of 30 FPS. It is also very unlikely that the tracking of the face is lost while recording, unless there is a major disturbance, like an object blocking the view of the face.

Configuring all input sources takes between 10 to 30 seconds, after that the recording can be started at any time. Figure 5.2 illustrates what animations look like when all input sources are synchronized by the prototype (a) and in contrast to that, what it would look like if all animations would be recorded separately and synchronized manually (b), which can easily result in synchronisation errors.



(a) Synchronisation While Tracking (b) Synchronisation After Tracking

Figure 5.2: Comparison of Synchronisation Methods

In (a) the model raises its hands and shows a surprised expression while turning right to the camera. In contrast (b) shows, that the model has not completely rotated to the camera yet, the hands are still facing inwards and the surprised face expression is delayed as well. The animations should look the same, but because synchronisation after recording is much harder, the animation is less accurate.

5.2 Performance

This section evaluates the performance of the prototype. For the analysis the CPU usage, rendering information and memory usage were observed. The following environment was used for the tests:

- Unity 5.1.1f1 (32-bit)
- Windows 8.1 Pro
- RAM 12,0 GB
- AMD Phenom^{TM} II X4 955, 3.20 GHz
  - L1 Cache Size: 128 KB
  - L1 Cache Count: 4
  - L2 Cache Size: 512 KB
  - L2 Cache Count: 4
  - L3 Cache Size: 6144 BK
- AMD Radeon^{TM} HD 7800 Series

All figures were created by taking snapshots from the Unity Profiler. In some of these snapshots the left columns with CPU Usage, Rendering and Memory are highlighted, but this does not influence any of the displayed values.

The vertical line represents the selected frame on the horizontal time axis.

A detailed list of the parameters from each section and how to properly interpreted them can be found in the official Unity documentation[1]. For the evaluation the most significant parameter is the execution time of the scripts in milliseconds. It is important, because many features have to be calculated from the captured input images of the body and face tracking methods.

[1] `http://docs.unity3d.com/Manual/Profiler.html`, Accessed: 2015-06-30

For the test, first every component was evaluated individually. Then the components were tested together with different settings. Additionally, the playback performance of the AnimationReader was tested in an isolated environment.

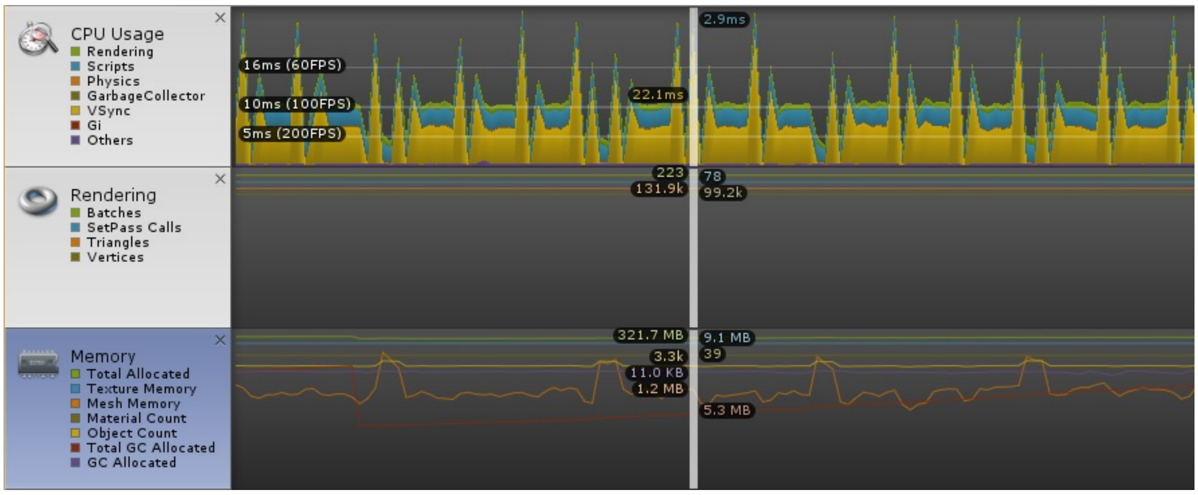


Figure 5.3: Performance of Hand Tracking Only

First the 5DT Data Gloves were tested. For the evaluation shown in Figure 5.3 both gloves were used. The chart shows that the execution time for the scripts is very short, because the input is directly retrieved from the gloves themselves and therefore no further calculations have to be performed. The input data can be directly applied to the referenced hand models. As a result, the average frames per second (FPS) are higher than 60, which is very good and ensures a lag and jitter free playback. There is no change in the rendering graph, since all objects remain in the scene - they are simply rotated around the calculated axes.

The mesh memory spikes in the Memory graph are a side effect of using the profile inside the Unity editor. They do not occur if the project is built and run outside of the editor. This also applies for all other graphs in this section.

Next, the body tracking with the Kinect v2.0 sensor was evaluated. The results are plotted in Figure 5.4. The CPU Usage graph shows a drastic increase in execution time for running the scripts. At the beginning however, the duration is always low and the FPS are at 60+ since the sensor has not recognized a body yet. After detecting the body, the FPS drop to a level around 30 to 40, which is still quite reasonable. At the end of the test the script execution time shows a rapid and constant increase. This is caused by switching the Kinect source reader from SeperateSourceReader to MultiSourceReader.

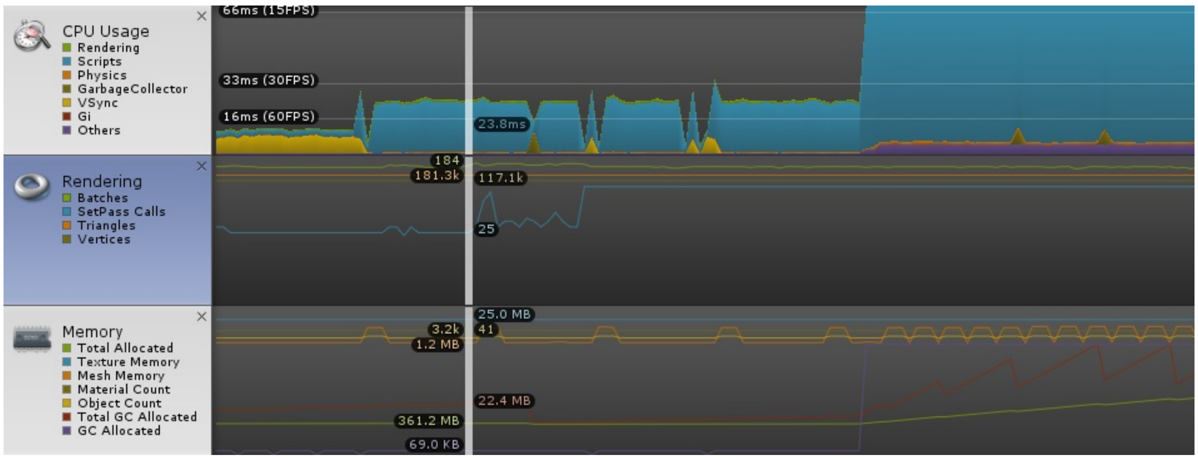


Figure 5.4: Performance of Body Tracking Only

If the MultiSourceReader option is active the scripts have to wait for all frames retrieved by the sensor to be ready in order to synchronize them.

The effects of the switch can also be seen in the Memory graph, because at the same time the script execution time increases there is a big increase of the total allocated memory and the allocated memory in the garbage collector rises as well.

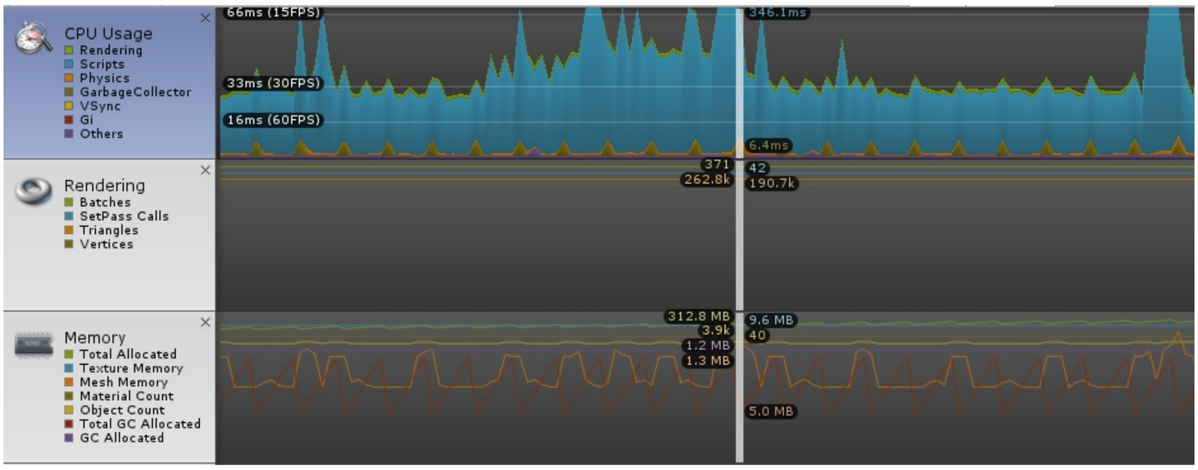


Figure 5.5: Performance of Face Tracking Only

Tested alone the face tracking runs with an average FPS around 30 most of the time, as can be seen in Figure 5.5. If the tracking is lost, which can happen due to sudden changes in lighting or if an object covers large parts of the face, the execution time of the scripts can drastically increase until the algorithm recovers. This can delay the whole system significantly with an execution time of 316.1 ms or longer per frame.

Since the wrapper for the face tracking is not optimized the Memory section shows the occurrence of frequent peaks for the allocated memory in the garbage collector, because some temporary data storages are not regularly freed.

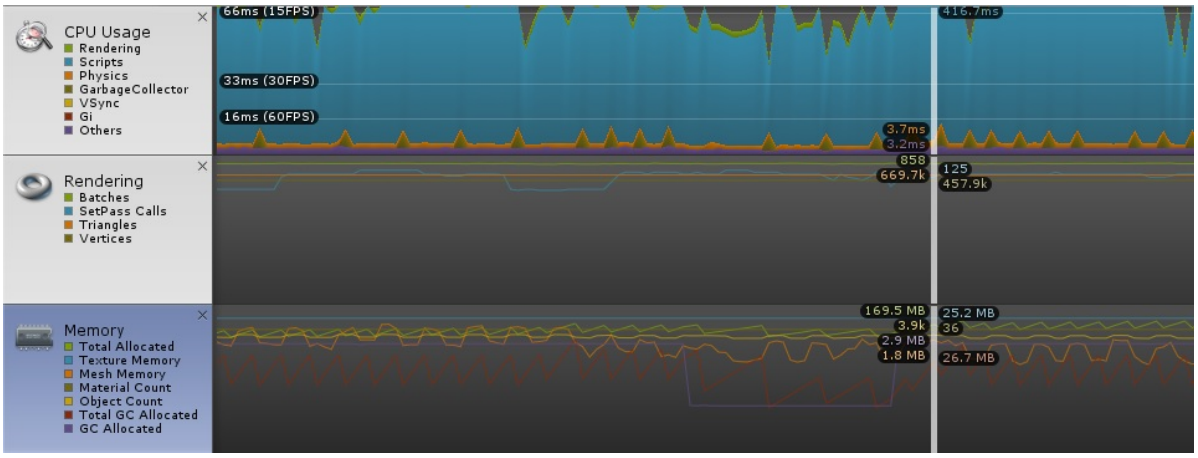


Figure 5.6: Performance of All Sources With MultiSourceReader

Next, Figure 5.6 shows the first setting that tested tracking all three input sources simultaneously. Here the SourceReader configuration for the Kinect is set to Multi-SourceReader which causes additional delay when running the scripts. This is the worst possible setting for tracking smooth animation data. The CPU Usage graph shows that the frame rate has dropped to below 15 FPS for most of the test. The marker indicates a frame where the face tracking algorithm is trying to recover from tracking loss. The CPU Usage graph shows that recovery in a test with three input sources takes even longer, which is a result of the CPU simultaneously computing the Kinect input and waiting for its sources to synchronize.

The drop of the GC Allocated graph in the Memory plot shows the time span in which the Kinect sensor lost tracking of the body.

A better approach is shown in Figure 5.7. For this test the Kinect SourceReader was set to SeperateSourceReader, so that the body tracking data could be passed on to the ModelController component as soon as it was captured by the sensor. To assure animation tracking that is completely lag free the process still needs to be optimized a bit for future releases, but a slight improvement in terms of FPS can already be seen. Since the settings for the face and hand tracker were not changed, the spikes indicating longer computation durations caused by the face tracker losing the tracking are still very high.

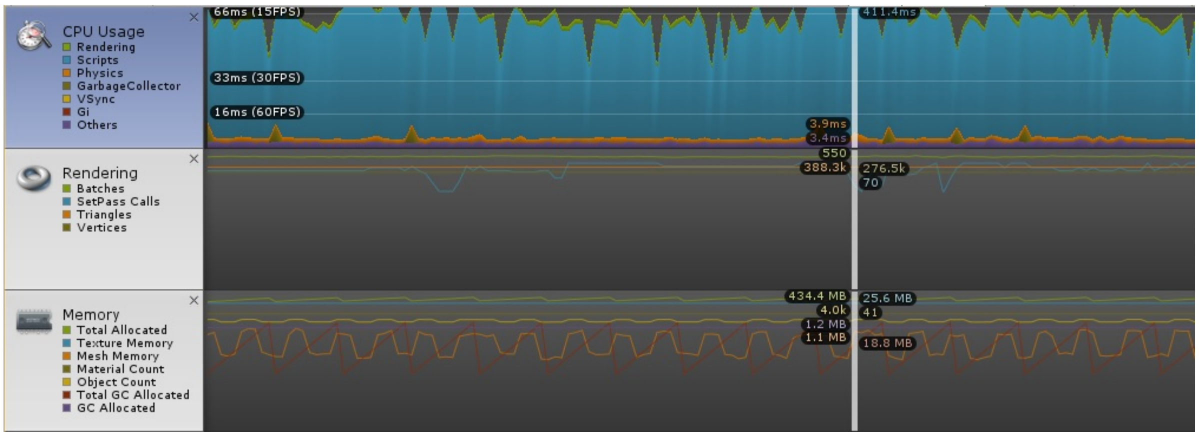


Figure 5.7: Performance of All Sources With SeperateSourceReader

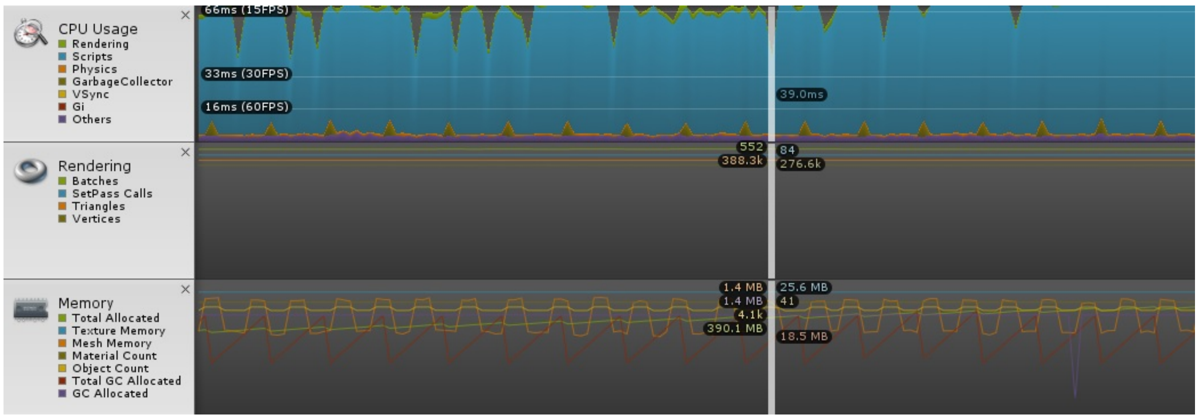


Figure 5.8: Performance While Recording Animations (of All Three Sources)

For the last test, tracking with all input sources, the same settings as for the previous test were chosen, but additionally live animation capturing was activated. The results are shown in Figure 5.8. This graph is very similar to the test chart without animation capturing, since face and body tracking still take up a considerable amount of the resources. The AnimationManager writing the animation data to the animation file reduces the computation speed slightly, but it has no significant influence on the overall performance of the prototype.

Finally, Figure 5.9 shows the playback of a captured animation. The reason the script execution duration chart spikes at the beginning is that the animation file is first loaded, parsed and interpreted when the scene is started. After that all information about rotations and positions of the model's joints are stored in the main memory and can be accessed via the AnimationReader component. After that, the data is simply applied to the model every frame, which is done almost instantly.

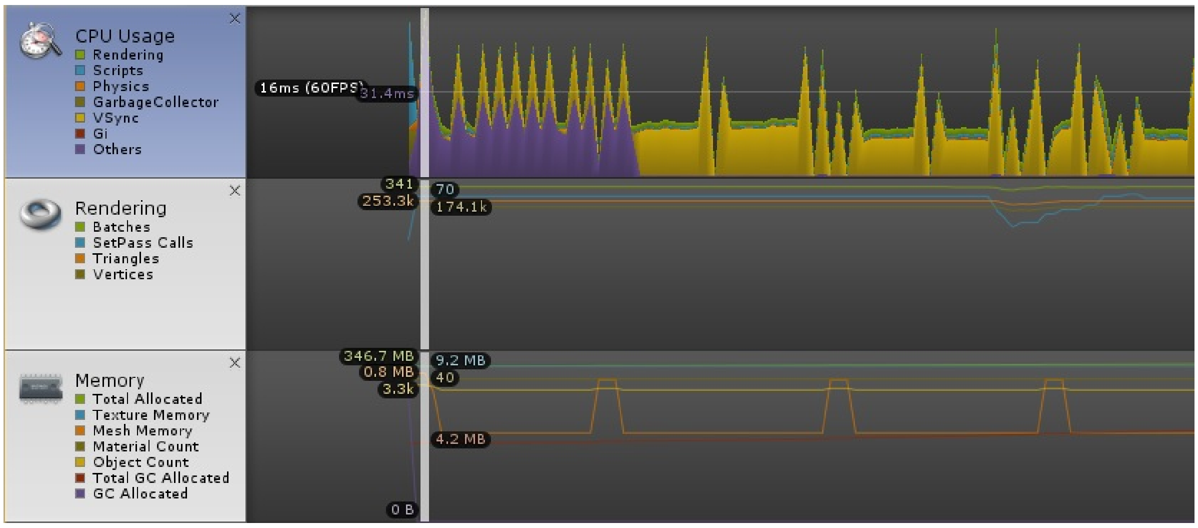


Figure 5.9: Performance Animation Playback

Concluding this section it can be said that, as long as only the body tracking data is used for the model animation it is more efficient to use the SeperateSourceReader with the Kinect tracking.

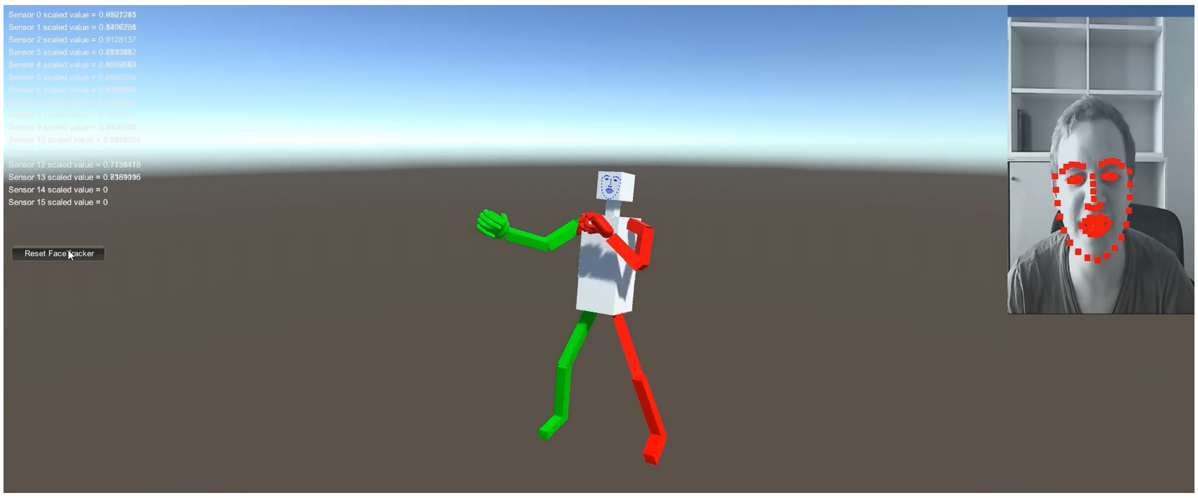
For the face tracking it is important to ensure that the tracking is not lost during the performance capture process, since face recognition takes a lot of time and can cause a heavy drop of frame rate. Therefore the calibration should be done before the recording process is started.

Using the body and face tracking together causes the system to drop below 24 FPS and should be improved further to ensure the generation of smooth animation data of high quality standard. But the performance of the hand tracking is sufficient, since it is fast and has no heavy impact on the frame rate. It needs no further improvement.

The playback of the animation files might take long because of the inefficient parsing, but it is still efficient enough if the animation file does not get too big.

5.3 Animation Quality

Not only the FPS and memory usage are crucial when it comes to creating animations. Therefore this sections reflects on how well the prototype can actually interpret and map the device input to the 3D models. Figure 5.10 shows the current test scene of the prototype.



```
Sensor 0 scaled value = 0.[illegible]
Sensor 1 scaled value = 0.[illegible]
Sensor 2 scaled value = 0.9128137
Sensor 3 scaled value = 0.[illegible]
Sensor 4 scaled value = 0.[illegible]
Sensor 5 scaled value = 0.6906318
Sensor 6 scaled value = 0.[illegible]
Sensor 7 scaled value = 0.[illegible]
Sensor 8 scaled value = 0.[illegible]
Sensor 9 scaled value = 0.[illegible]
Sensor 10 scaled value = 0.[illegible]
[illegible]
Sensor 12 scaled value = 0.[illegible]
Sensor 13 scaled value = 0.[illegible]
Sensor 14 scaled value = 0
Sensor 15 scaled value = 0
```

Figure 5.10: Tracking Environment Inside Unity

In the top left corner the input data from the 5DT Data Gloves is displayed. Below that, there is a button to reset the face tracking. This button is needed to prompt the initial face detection as soon as the user is positioned and ready. The webcam pictures and the face feature points are displayed on an extra screen to the right.

Once the Kinect sensor recognizes a body, the avatar automatically jumps to the correct position on the screen and mirrors the movements of the tracked user.

5.3.1 Hand Tracking Errors

Since the hand tracking uses a mechanical tracking algorithm it is very stable and a loss of tracking data is not to be expected. Nevertheless it is still possible that issues occur with the received data. One example can be seen in Figure 5.11. Note how in the picture the flexure of the fingers does not match the model on the screen.

This can happen if,

- the hand of the user is too small or too big to fit the glove, which is very unlikely since the gloves are made of elastic materials,
- the glove is not worn correctly, or

- the sensors inside the glove are not properly placed.

Otherwise the tracking should work just fine.

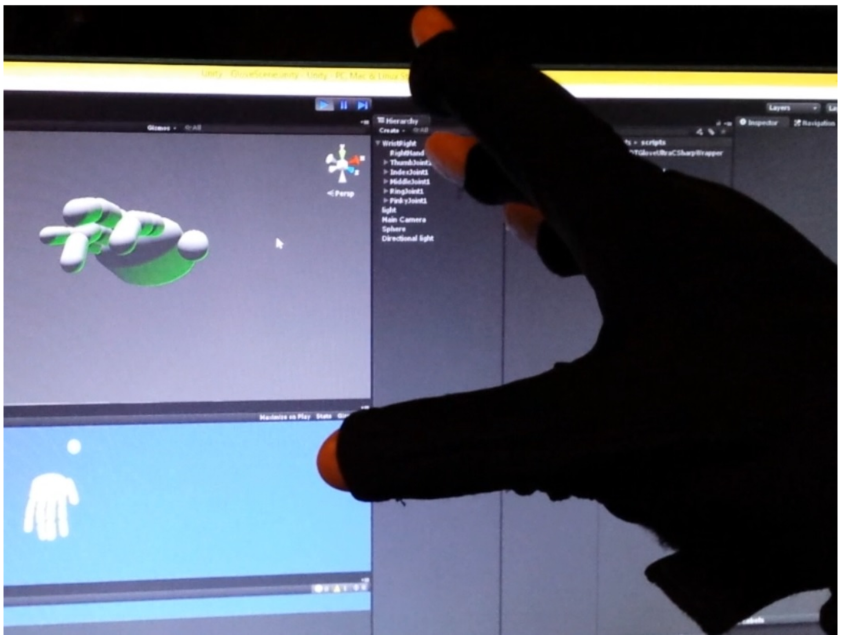


Figure 5.11: Tracking Errors With 5DT Data Gloves

5.3.2 Body Tracking Errors

As shown in Figure 5.12, it can happen that the Kinect loses track of some of the user's body parts, which mostly occurs when the depth sensor doesn't recognize from which part of the body the depth information is retrieved. This mainly happens when body parts are overlapping, e.g. when hands are held in front of the body or when the knee is pulled up to the chest.

Consequently, another reason for tracking errors is when a body part disappears from view completely, e.g. holding hands behind the back or setting one foot in front of the other.

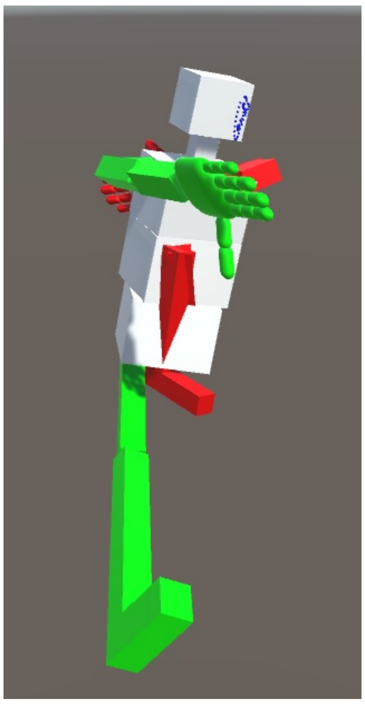


Figure 5.12: Tracking Errors With Kinect

5.3.3 Face Tracking Errors

With the face tracking procedure there are also some issues which have to be considered. The first one is that the initial tracking often fails because the user is not in an optimal position. Other issues are dealing with users wearing glasses with thick frames, users with beards and mistaking the collar for the edge of the face.

Figure 5.13 shows how a false initial tracking looks like. However, once the face is recognized correctly the tracking is very stable and unlikely to be lost again. At least as long the lighting does not drastically change and there are no objects between the face and the camera.

Concluding this chapter it can be said that the prototype still needs to be improved in terms of performance. Concerning the animation quality there are also some issues to be resolved, but if the previously mentioned suggestions for improved tracking are implemented the number of tracking errors would be reduced significantly.

However, it is also an option to remove the remaining tracking errors from the animation file recorded by the present prototype, after the capturing is complete.

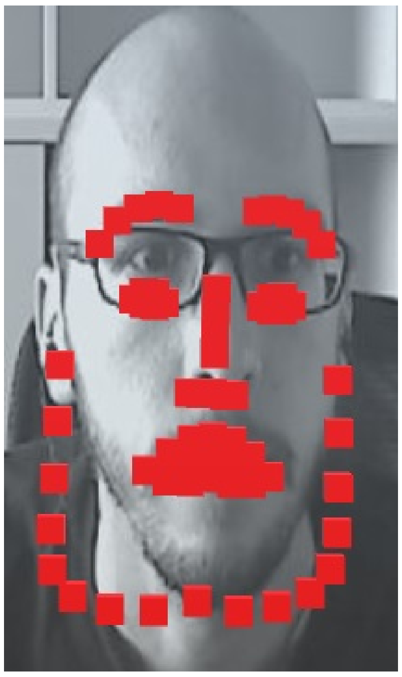


Figure 5.13: Webcam Tracking Errors

6 Discussion and Limitations

In this section the prototype will be compared to other capturing systems. This is done by listing the main criteria and evaluating if these can be met or not. After this the strengths and limitations of the prototype will be discussed.

At the end of the chapter there will be a list of possible improvements that could be made to increase the performance of the prototype, and the animation quality.

6.1 Comparison

This section compares the prototype to some of the tracking solutions presented in Chapter 2. In Table 6.1 the main criteria are listed. The importance of some of these criteria was only realized during the development of the prototype. Others were inspired by [3], but adapted for the evaluation of this thesis.

Each entry of the table has a value between 0 and 2. The following list shows what these codes stand for:

- 0: the feature is not supported
- 1: the feature is partly available or not well supported
- 2: the feature is fully supported

For the affordability feature a 2 was given for prices under €3.000,– a 1 for prices under €5.000,– and a 0 for everything with prices of €5.000,– and up.

After comparison, it can be said that the prototype still needs to be improved to keep up with professional solutions, especially in terms of runtime performance, tracking stability

	Prototype	Fastmocap	Face Plus	Facerig	Control VR
Face tracking	2	0	2	2	0
Body tracking	2	2	0	0	2
Exact hand tracking	2	0	0	0	2
Affordable	2	2	2	2	2
Noise resistant	1	1	1	1	2
Distance does not influence accuracy	0	1	0	0	2
Haptic feedback	0	0	0	0	1
Runtime performance	0	2	2	2	2
Established animation formats	0	2	2	0	2

Table 6.1: Comparison of Different Tracking Systems

and exportable animation formats.

However, the prototype is an affordable solution that is capable of simultaneously tracking exact hand movement, and body and face features, which is not possible with any of the other solutions. Also, since it is only the first version of the prototype there are still many things that can be improved. A detailed list of possible improvements is given at the end of this chapter.

6.2 Strengths and Limitations

This section will cover some of the main strengths and limitations of the prototype which were pointed out during the comparison.

6.2.1 Expandable All in One Tracking

One of the greatest achievements of the prototype is clearly that it combines hand, body and face tracking in one development environment and makes it possible to individually adapt the tracking methods to the user's needs.

All the components are designed in such a way that they can be expanded with new code or replaced with new components. This enables the prototype to be extended by new tracking devices. It is also possible to change the way the input data is interpreted, by only exchanging one script in the ModelController component. The DataManagement module is also extendible and can be added to all new controller components as well.

6.2.2 Costs

All prices and exchange rates[1][2][3] were taken from the official websites by 2015-06-18.

Device	Price
5DT Data Glove 5 Ultra (Right)	€873,45 ($995)
5DT Data Glove 5 Ultra (Left)	€873,45 ($995)
Kinect for Windows v2 sensor	€199.99
Logitech HD Pro Webcam C920	€99.99
TOTAL	€2046,88

Table 6.2: Estimated Costs for the Equipment

As Table 6.2 shows, the biggest part of the expenses comes from the 5DT Data Gloves. If they could be exchanged with cheaper tracking devices the price for the tracking system could be greatly reduced.

6.2.3 Low Runtime Performance

With approximately 15 FPS, in its current state, the prototype is too slow for live capturing. If only body and hand or hand and face are tracked simultaneously the FPS rise to around 30.

In order to get tracking data in good quality with enough FPS it would be necessary to further improve the runtime performance of the prototype. Suggestions about how this could be accomplished are made in the next section.

[1]`http://www.logitech.com/de-at/product/hd-pro-webcam-c920`, Accessed: 2015-06-18
[2]`https://www.microsoft.com/en-us/kinectforwindows/purchase/v2sensor.aspx`, Accessed: 2015-06-18
[3]`http://www.5dt.com/?page_id=34`, Accessed: 2015-06-18

6.3 Possible Improvements

Since this is the first version of the prototype and its main goal was to see if it is even possible to include all this functionality in one project, there are a lot of potential ways by which its performance could be improved. The following section discusses a few adjustments that could be made to achieve a higher animation quality.

6.3.1 Increasing Runtime Performance

The project is not optimized for runtime performance yet. One way to improve the performance would be to move calculations from the CPU to the GPU or to make a x64 build of the project. Also, the prototype was only tested inside the Unity editor. An optimized release could bring a performance boost as well.

Alternatively, the tracking could be done separately on two different devices. For example one PC could be used for performing the face tracking and another one for tracking hand and body movement. The recorded input data would need to be synchronized afterwards.

6.3.2 Adding and Improving Filters

Each ModelController provides the option to add a filter component. The current prototype does not include any filters. Adding filters would be an easy way to improve the tracked animation quality, but the additional calculations could also slow down the prototype.

Another approach for improving the prototype would be to check the tracking state of each Kinect JointOrientation before applying its rotation to the joints of the model. Restrictions could be added that check if the rotation of a joint changes rapidly between two frames. If it does, the difference between the rotations could be interpolated. Or another approach would be to keep the rotation of the first frame until the tracking has recovered. A combination of these two methods would also be possible. On the other hand, if the tracked rotation values change very often, it can be interpreted as jitter, in which case the rotations should be smoothed.

Alternatively it would also be possible to add algorithms for removing jitter from the animations after they were recorded. The advantage of this approach would be, that it would not influence the run time performance during the tracking. There is literature available [49] about how the depth accuracy of the Kinect for Windows v2 can be evaluated and improved. Another approach [41] shows how to measure body joints and how to use the collected information to integrate this data into a motion capturing system.

6.3.3 Joint Restrictions

Adding joint restrictions is a similar approach to adding filters. The main difference is that checks are only performed if the actual joint orientation is physically possible for the human body. For example, if applying the tracking data for the new frame would turn the head farther than humanly possible, the transformation would be discarded.

Therefore a restriction database could be added to the project. Then a check could be performed in every frame, if the joint rotations are valid or not.

6.3.4 Using More Input Devices for Body Tracking

Extending the current prototype by other or additional tracking devices is another possibility to increase the animation quality. One approach shown in [14] uses multiple Kinects for body tracking. This improves the tracking quality significantly.

On the other hand, increasing the number of tracking devices could negatively affect the run time performance and the FPS.

6.3.5 Adding Colored Markers to Improve Tracking

The current prototype only relies on markerless visual and mechanical tracking. Optical beacons could be easily added to the project, since OpenCV is already included into the prototype. Such beacons could be used to ensure that the hands are tracked from the right side and therefore prevent unnatural hand rotations caused by tracking issues with the Kinect.

Then again, adding additional processing steps to the prototype also could mean an extended script execution time.

6.3.6 Using Kinect Face Tracking for Head Rotation

The current version of the prototype does not support head rotation. This is mainly because of the fact that for unknown reasons Unity continued to crash when it was tried to access face tracking data from the Kinect input.

If this issue would be solved the animations would look much more natural. This would be very beneficial for the animation quality.

6.3.7 Improving Environmental Conditions

Since two third of the tracking is done by optical motion capture, improving the lighting of the environment could have a positive influence on the stability of the tracking and therefore increase the quality of the captured animation data.

The only negative aspect is that this improvement would also increase the equipment costs.

6.3.8 Export Format

The only data format which is currently supported by the prototype is the one specially developed for it. This file format is not supported by any other applications, which means that it can't be exported easily and thus makes it almost impossible to edit the animation data after recording.

A possible solution would be to replace the current file export format with a more common format; for example the .FBX file format.

7 Conclusion

In this thesis, it was shown that it is possible to combine multiple input sources inside the Unity environment, to achieve a low budget performance capturing system, which can capture, export and import animations at runtime.

The chosen input sources might not be an optimal composition to track detailed animations on a professional level, but there are many new technologies like the ones introduced in Chapter 2 that could be included in future releases of the prototype for more promising results. Since the prototype was developed in a way that can be extended by as many input sources as needed, it is possible to try new approaches of motion capturing with different devices.

However, even if the tracking data has some issues like optical tracking problems with the Kinect v2.0 and the webcam, as well as inaccurate sensor input from the 5DT Data Gloves, the data can be manually adapted and corrected after the tracking process. This would still save a lot of work and be more cost efficient than creating all animations by hand.

After giving a general overview of the problem field and showing solutions for individual issues, the general aspects of the work were explained in Chapter 3. Also, the core technologies were presented that were used for developing the prototype and an overview of how and where they were used in the prototype was given.

Chapter 4 illustrated how the structure of the prototype was planned and how each of the tracking components was developed and integrated into the project. The main issues that occurred were covered, as well as possible solutions and workarounds.

In Chapter 5 the prototype was evaluated in terms of runtime performance, rendering and memory usage. A special focus was also put on achieving sufficient animation quality. Subsequently it was shown that the prototype still has issues with its runtime performance, especially with meeting the required FPS, if all three tracking sources are used simultaneously.

Finally, the prototype was compared to other capturing solutions in Chapter 6 and its strengths and limitations were evaluated. Based on these results possible improvements and extensions might be envisioned for future versions of the prototype.

Concluding this work it can be said that even though the prototype is not suited for professional use yet, it has the potential to overcome its shortcomings in the near future and to be an affordable and adaptable solution for capturing performance based animation data.

Bibliography

[1] 5dt data gloves, c# driver (ms .net 2.0 and upwards, mono and unity). http://www.5dt.com/?page_id=34. Accessed: 2015-04-21.

[2] Cascade classifier. http://docs.opencv.org/doc/tutorials/objdetect/cascade_classifier/cascade_classifier.html. Accessed: 2015-04-21.

[3] Control vr. http://controlvr.com/. Accessed: 2015-04-17.

[4] Development tools and languages, kinect for windows sdk 2.0. https://msdn.microsoft.com/en-us/library/dn799271.aspx. Accessed: 2015-04-21.

[5] Documentation, csiro face analysis sdk. http://face.ci2cv.net/doc/. Accessed: 2015-06-27.

[6] Face plus. https://www.mixamo.com/faceplus. Accessed: 2015-04-17.

[7] Kinect v2 with ms-sdk. https://www.assetstore.unity3d.com/en/#!/content/18708. Accessed: 2015-04-22.

[8] Opencv eye tracking with c#. http://www.prodigyproductionsllc.com/articles/programming/opencv-eye-tracking-with-c/. Accessed: 2015-04-21.

[9] Preparing your own character. http://docs.unity3d.com/Manual/Preparingacharacterfromscratch.html. Accessed: 2015-05-04.

[10] Press downloads. https://unity3d.com/public-relations/downloads. Accessed: 2015-07-03.

[11] Technical documentation and tools, kinect for windows v2 essentials. https://www.microsoft.com/en-us/kinectforwindows/develop/downloads-docs.aspx. Ac-

cessed: 2015-04-22.

[12] Xsense. `https://www.xsens.com/`. Accessed: 2015-07-24.

[13] 5DT (Fifth Dimension Technologies). *5DT Data Glove Ultra Manual v1.3*, January 2011.

[14] Seongmin Baek and Myungyu Kim. Real-time performance capture using multiple kinects. In *Information and Communication Technology Convergence (ICTC), 2014 International Conference on*, pages 647–648, Oct 2014.

[15] J.C. Barca, G. Rumantir, and R. Koon Li. A new illuminated contour-based marker system for optical motion capture. In *Innovations in Information Technology, 2006*, pages 1–5, Nov 2006.

[16] R. Belaroussi and M. Milgram. Face tracking and facial feature detection with a webcam. In *Visual Media Production, 2006. CVMP 2006. 3rd European Conference on*, pages 122–126, Nov 2006.

[17] D. Casas, M. Tejera, J. Guillemaut, and A. Hilton. Interactive animation of 4d performance capture. *Visualization and Computer Graphics, IEEE Transactions on*, 19(5):762–773, May 2013.

[18] Chenyang Chen, Mingmin Zhang, Kaijia Qiu, and Zhigeng Pan. Real-time robust hand tracking based on camshift and motion velocity. In *Digital Home (ICDH), 2014 5th International Conference on*, pages 20–24, Nov 2014.

[19] H. Chen, Gang Qian, and J. James. An autonomous dance scoring system using marker-based motion capture. In *Multimedia Signal Processing, 2005 IEEE 7th Workshop on*, pages 1–4, Oct 2005.

[20] Taehoon Cho, Jin-Ho Choi, Hyeon-Joong Kim, and Soo-Mi Choi. Vision-based animation of 3d facial avatars. In *Big Data and Smart Computing (BIGCOMP), 2014 International Conference on*, pages 128–132, Jan 2014.

[21] Jongmoo Choi, Y. Dumortier, Sang-Il Choi, M.B. Ahmad, and G. Medioni. Real-time 3-d face tracking and modeling from a webcam. In *Applications of Computer Vision (WACV), 2012 IEEE Workshop on*, pages 33–40, Jan 2012.

[22] Ahmed Elgammal, Bodo Rosenhahn, and Reinhard Klette, editors. *Human Motion – Understanding, Modeling, Capture and Animation*. Springer Berlin Heidelberg, 2007.

[23] Xianghua Fan, Fuyou Zhang, Haixia Wang, and Xiao Lu. The system of face detection based on opencv. In *Control and Decision Conference (CCDC), 2012 24th Chinese*, pages 648–651, May 2012.

[24] Youngmo Han. 2d-to-3d visual human motion converting system for home optical motion capture tool and 3-d smart tv. *Systems Journal, IEEE*, 9(1):131–140, March 2015.

[25] Ye Hu, Wenguang Jin, and Feng Ni. An efficient wireless sensor network for real-time multiuser motion capture system. In *Communication Technology (ICCT), 2012 IEEE 14th International Conference on*, pages 155–160, Nov 2012.

[26] Haoda Huang, Jinxiang Chai, Xin Tong, and Hsiang-Tao Wu. Leveraging motion capture and 3d scanning for high-fidelity facial performance acquisition. In *ACM SIGGRAPH 2011 Papers*, SIGGRAPH '11, pages 74:1–74:10, New York, NY, USA, 2011. ACM.

[27] Li Jia, Miao Zhenjiang, Cheng Hengda, and Zhang Dianyong. Unsynchronized markerless motion capture with sharp illumination changes. In *Image Processing (ICIP), 2010 17th IEEE International Conference on*, pages 1529–1532, Sept 2010.

[28] Dimitrios I. Kosmopoulos and Fillia Makedon. Erratum: A method for online analysis of structured processes using bayesian filters and echo state networks. In *Computer Vision – ECCV 2012. Workshops and Demonstrations*, pages E1–E1. Springer Science Business Media, 2012.

[29] A. Kyme, S. Se, S. Meikle, G. Angelis, W. Ryder, K. Popovic, D. Yatigammana, and R. Fulton. Markerless motion tracking of awake animals in positron emission tomography. *Medical Imaging, IEEE Transactions on*, 33(11):2180–2190, Nov 2014.

[30] Dongxiao Li, Chen Sun, Fangqin Hu, Dongning Zang, LiangHao Wang, and Ming Zhang. Real-time performance-driven facial animation with 3ds max and kinect. In *Consumer Electronics, Communications and Networks (CECNet), 2013 3rd International Conference on*, pages 473–476, Nov 2013.

[31] Yangmi Lim, Jinsu Kim, and Jinseok Chae. Risa: A real-time interactive shadow avatar. In *Multimedia, 2007. ISM 2007. Ninth IEEE International Symposium on*, pages 112–122, Dec 2007.

[32] Guo-Shiang Lin and Tung-Sheng Tsai. A face tracking method using feature point tracking. In *Information Security and Intelligence Control (ISIC), 2012 International Conference on*, pages 210–213, Aug 2012.

[33] Yebin Liu, C. Stoll, J. Gall, H.-P. Seidel, and C. Theobalt. Markerless motion capture of interacting characters using multi-view image segmentation. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 1249–1256, June 2011.

[34] Kim Doang Nguyen, I-Ming Chen, Song Huat Yeo, and Been-Lirn Duh. Motion control of a robotic puppet through a hybrid motion capture device. In *Automation Science and Engineering, 2007. CASE 2007. IEEE International Conference on*, pages 753–758, Sept 2007.

[35] C. Ott, Dongheui Lee, and Y. Nakamura. Motion capture based human motion recognition and imitation by direct marker control. In *Humanoid Robots, 2008. Humanoids 2008. 8th IEEE-RAS International Conference on*, pages 399–405, Dec 2008.

[36] Chia ping Chen, Yu-Ting Chen, Ping-Han Lee, Yu-Pao Tsai, and Shawmin Lei. Real-time hand tracking on depth images. In *Visual Communications and Image Processing (VCIP), 2011 IEEE*, pages 1–4, Nov 2011.

[37] Jason M. Saragih, Simon Lucey, and Jeffrey F. Cohn. Deformable model fitting by regularized landmark mean-shift. *Int. J. Comput. Vision*, 91(2):200–215, January 2011.

[38] Jason M. Saragih, Simon Lucey, and J.F. Cohn. Real-time avatar animation from a single image. In *Automatic Face Gesture Recognition and Workshops (FG 2011), 2011 IEEE International Conference on*, pages 117–124, March 2011.

[39] M. Schroder, J. Maycock, H. Ritter, and M. Botsch. Real-time hand tracking using synergistic inverse kinematics. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 5447–5454, May 2014.

[40] Toby Sharp, Cem Keskin, Duncan Robertson, Jonathan Taylor, Jamie Shotton, David Kim, Christoph Rhemann, Ido Leichter, Alon Vinnikov, Yichen Wei, Daniel Freedman, Pushmeet Kohli, Eyal Krupka, Andrew Fitzgibbon, and Shahram Izadi. Accurate, robust, and flexible real-time hand tracking. CHI, April 2015.

[41] Jungsu Shin, Kyeong-Ri Ko, and Sung Bum Pan. Automation of human body model data measurement using kinect in motion capture system. In *Consumer Electronics (ICCE), 2015 IEEE International Conference on*, pages 88–89, Jan 2015.

[42] J. Smisek, M. Jancosek, and T. Pajdla. 3d with kinect. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 1154–1160, Nov 2011.

[43] Guanhong Tao, Shuyan Sun, Shuai Huang, Zhipei Huang, and Jiankang Wu. Human modeling and real-time motion reconstruction for micro-sensor motion capture. In *Virtual Environments Human-Computer Interfaces and Measurement Systems (VECIMS), 2011 IEEE International Conference on*, pages 1–5, Sept 2011.

[44] M. Tejera, D. Casas, and A. Hilton. Animation control of surface motion capture. *Cybernetics, IEEE Transactions on*, 43(6):1532–1545, Dec 2013.

[45] Xiaolong Tong, Pin Xu, and Xing Yan. Research on skeleton animation motion data based on kinect. In *Computational Intelligence and Design (ISCID), 2012 Fifth International Symposium on*, volume 2, pages 347–350, Oct 2012.

[46] Chengkai Wan, Baozong Yuan, and Zhenjiang Miao. Model-based markerless human body motion capture using multiple cameras. In *Multimedia and Expo, 2007 IEEE International Conference on*, pages 1099–1102, July 2007.

[47] Xin Wang, Qing Ma, and Wanliang Wang. Kinect driven 3d character animation using semantical skeleton. In *Cloud Computing and Intelligent Systems (CCIS), 2012 IEEE 2nd International Conference on*, volume 01, pages 159–163, Oct 2012.

[48] C. Wong, Zhiqiang Zhang, B. Lo, and Guang-Zhong Yang. Markerless motion capture using appearance and inertial data. In *Engineering in Medicine and Biology Society (EMBC), 2014 36th Annual International Conference of the IEEE*, pages 6907–6910, Aug 2014.

[49] L. Yang, L. Zhang, H. Dong, A. Alelaiwi, and A. El Saddik. Evaluating and improving the depth accuracy of kinect for windows v2. *Sensors Journal, IEEE*, PP(99):1–1, 2015.

[50] M. Zabri Abu Bakar, R. Samad, D. Pebrianti, and N.L.Y. Aan. Real-time rotation invariant hand tracking using 3d data. In *Control System, Computing and Engineering (ICCSCE), 2014 IEEE International Conference on*, pages 490–495, Nov 2014.

[51] Wenbing Zhao, Hai Feng, R. Lun, D.D. Espy, and M.A. Reinthal. A kinect-based rehabilitation exercise monitoring and guidance system. In *Software Engineering and Service Science (ICSESS), 2014 5th IEEE International Conference on*, pages 762–765, June 2014.

[52] Victor Brian Zordan and Nicholas C. Van Der Horst. Mapping optical motion capture data to skeletal motion using a physical model. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, pages 245–250, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[53] Min Zuo, Guangping Zeng, and Xuyan Tu. Research and improvement of face detection algorithm based on the opencv. In *Information Science and Engineering (ICISE), 2010 2nd International Conference on*, pages 1413–1416, Dec 2010.

Abstract

Transferring an actor's movements directly onto a virtual 3D model is an effective approach to creating animations. But even better is being able to not only transfer movements, but also expressions onto the model. These make the animations seem even more realistic and natural to the viewer. But creating these kinds of high class animations is complex and time consuming. The goal of this thesis is the development of a functional prototype, that efficiently realizes such a performance capturing system. It should enable the user to directly transfer complex motion sequences onto a 3D character by using live motion capturing, and if so desired, to even record these animations for later use. The recorded animations are supposed to be transferrable onto any humanoid 3D model. The capturing process can be segmented into three basic steps: the whole body, or rather the skeleton, of the user is tracked by a Kinect v2. Hand and finger movements are registered via 5DT Data Gloves. The face is recorded with a webcam, while feature points are extracted from it to facilitate the transfer of expressions and emotions onto the face of the model. The prototype was developed using the Unity game engine and will likely be available as an Asset Package to enable easy import. For the import to function correctly it is essential to use Unity version 5 or later, because otherwise the included plugins are not supported. The prototype was constructed in such a way, that some of the components are interchangeable, so it is possible to constantly keep adding new tracking methods. As a result it is possible to customize the prototype and even to easily extend and improve it. The prototype was performance tested with Unity's analytical tools and compared to other tracking solutions through a qualitative analysis. The results of these tests are documented in this thesis.

Zusammenfassung

Bewegungen eines Akteurs direkt auf ein virtuelles 3D Modell zu übertragen ist eine effektive Art um Animationen zu generieren. Doch es gibt noch die Möglichkeit mehr als nur den Körper, sondern auch Emotionen auf Modell übertragen. Die dadurch gewonnenen Animationen wirken noch realistischer und sind künstlich sehr schwierig und Zeitaufwändig in einer solchen Qualität zu erzeugen. Das Ziel der Arbeit ist die Entwicklung eines lauffähigen Prototypen der ein solches Performance capturing system realisiert. Es soll einer Person, durch live Capturing, ermöglich werden komplexe Bewegungsabläufe und Emotionen auf einen 3D Charakter zu übertragen und diese, falls gewünscht, aufzunehmen. Diese Animation Daten sollen auf ein beliebiges humanoides 3D Modell anwendbar sein. Der capturing Prozess kann im Wesentlichen in drei Hauptbereiche unterteilt werden. Der ganze Körper, beziehungsweise das Skelett, des Acteurs wird mit Hilfe einer Kinect V2 getrackt. Bewegungen der Finger werden durch 5DT Data Gloves aufgezeichnet. Das Gesicht wird mit einer Webcam gefilmt. Dabei werden Feature Punkte extrahiert um durch diese Bewegungen/Emotionen auf das Gesicht des des Modells zu übertragen. Der Prototyp wird für die Unity Engine entwickelt und voraussichtlich in der Endversion als Asset Package verfügbar sein, um einen einfachen Import zu gewährleisten. Damit der Import problemlos funktioniert ist es wichtig, dass mindestens Unity version 5 verwendet wird. Andernfalls werden die verwendeten Plugins nicht unterstütz. Des Weiteren ist der Prototyp so entwickelt, dass Komponenten beliebig geändert werden können. Auch das Hinzufügen neuer tracking verfahren ist möglich. Dadurch ist es möglich den Prototypen individuell an zu passen und zu verbessern. Mit den Analysetools aus Unity wurde der Prototyp im Hinblick auf seine Leistungsfähigkeit getestet. Außerdem wurde er anhand einer qualitativen Analyse anderen Trackinglösungen gegenübergestellt. Die daraus gewonnenen Resultate sind in der Arbeit dokumentiert.

Acknowledgements

To begin with, I would like to thank Univ.-Prof. Dipl.-Ing. Dr. Helmut Hlavacs for all the support during my time at the university and during the completion of my master's thesis. For technical support I would like to express my appreciation to Mag. Ewald Hotop. Also, I am particularly grateful for the assistance given by Christopher Helf and Daniel Martinek, who greatly helped me when I had issues with Unity. Rebecca Wölfle was also very helpful by supporting and proofreading my thesis.

Special thanks goes to my fellow students Magdalena Stallinger, Thomas Rotter, Manfred Ranner and Benjamin Nussbaum for always being there for me and for the great time we had together all these past years.

Finally I would like to thank all my friends, especially Florian Nichtawitz, my parents Franz and Maria Wagner and my dear brother Gabriel for all their support and for believing in me.

HANNES WAGNER

LEBENSLAUF

ANGABEN ZUR PERSON

Wohnort	Wien
E-Mail	hannes.wagner123@gmail.com

AUSBILDUNG

Okt.2012 – jetzt	MSc, Universität Wien Medieninformatik (Master)
Okt.2009 – Jun.2012	BSc, Universität Wien Medieninformatik (Bachelor)
Sep.2003 – Jun.2008	Matura, BHAK Horn Ausbildungsschwerpunkt Informationsmanagement und Digital Business

ZIVILDIENST

Jan.2009 – Okt.2009	Sanitäter und Einsatzfahrer Rotes Kreuz (Bezirksstelle Horn)

PERSÖNLICHE FÄHIGKEITEN

Muttersprache Deutsch

Weitere Sprachen

	VERSTEHEN		SPRECHEN		SCHREIBEN
	Hören	Lesen	An Gesprächen teilnehmen	Zusammenhängendes Sprechen	
Englisch	C1	C1	B2	B2	B2
Französisch	A2	A2	A1	A1	A1

Führerschein B

Computerkenntnisse

Betriebssysteme	Microsoft Windows 8 (und frühere bis XP), Linux (Ubuntu), Android
Programmiersprachen	Java, C#, C++, Python, R
Skriptsprachen	Javascript, Groovy (Basics)
DB	MySQL
Web und XML-based	(X)HTML, PHP, CSS, XML, XSLT, XSD, XPath, XQuery, XSL-FO, SMIL (Basics), YAML (Basics), REST, JavaScript (jQuery), Java (JSP, Servlets), JSON, AJAX
Software und Tools	Eclipse, MS Office 2013, VS2013 (und frühere), Matlab 2013b Unity, Basics: Blender, Wolfram Mathematica 7 und 8, Enterprise Architect, Metaio
Sonstiges	Xtext, EMF, Ogre 3D, FFmpeg, LaTeX