Hand Tracking with Microsoft Kinect v2 and Leap Motion

Practical Course of Data Fusion

Hendrik Bitzmann
Hanna Holderied
Christoph Rauterberg
Maximilian Weß

Jun. Prof. Marcus Baum
Institute for Applied Computer Sciences
University of Göttingen
September 2016

Contents

CHAPTER 1

Introduction

This project report is the result of the participation in the "Practical Course: Data Fusion" in the Summer Semester 2016.

The goal of this project was to construct a scene in a virtual reality in which a user is able to interact with an object using his or her hands and fingers. To achieve this goal we were given two kinds of sensors:

1. A Microsoft Kinect v2[1] and
2. A Leap Motion Sensor[2].

The Microsoft Kinect v2 was released in 2014 and brings an infrared depth sensor, a RGB sensor and 3D motion tracking to the table. The last is especially important for this project. The Kinect can reach between 15 to 30 frames per second (fps).
The Leap Motion sensor uses two monochromatic IR cameras and three infrared LEDs to observe a hemispherical area in front of the device and can reach up to 200fps.

In this report we will highlight our line of work to enable future research to be able to set up such an environment more quickly. Firstly, we will describe all steps that are necessary to achieve a working Kinect environment under Windows and Linux. We will then describe which algorithmic approaches we inferred and in the end chose to achieve the desired goals of this project. We will conclude by discussing our first approach for combining the sensor readings from the Kinect and the Leap Motion Sensor.

[1] https://developer.microsoft.com/de-de/windows/kinect/develop
[2] https://www.leapmotion.com/

CHAPTER 2

Methods

2.1 Setup

Configuration of the Kinect v2

Firstly we have to emphasize that the Kinect sensors (Kinect v1 and Kinect v2) are provided by the Microsoft Cooperation and are therefore for development under the operating system Windows, nowdays preferrably Windows 10. All current solutions for Linux are community based.

There are solutions such as *libfreenect*[1] which provide drivers for Linux - however, we could only achieve a working version of this with Ubuntu and not Arch Linux. To establish a working version of *libfreenect* with the according Python-wrapper *python-freenect2*[2] took quite an amount of work - we would therefore discourage anyone from using the Linux drivers if time is dire.

Furthermore it is important to mention that you need to have an USB 3.0 port if you want to use the Kinect v2, otherwise it will not work. In addition you need moderate computation power. It turned out that a notebook with just HD4000 graphics is not suitable.

The first step for using the Kinect v2 under Windows is downloading the official *Kinect 2.0 Software Development Kit* from Microsoft for Windows[3]. The SDK includes many useful tools and examples. Next step should be starting the Kinect Configuration Verifier from the SDK browser, which will check the system requirements. Every check besides for the USB controller should be successful. In our experience, a warning for the USB port did not turn out to be a problem.

Set up a working development environment

We choose a running start by following the tutorial from [???] . This gave us a working development environment consisting of Unity3D, Visual Studio 2015 and the Kinect SDK, already with a working "Hello World" example.
Unity3d is a game development engine. It is a good choice for our application, because of ease which simple tasks can be executed. The import of assets into the project can be achieved via drag and drop and also linking scripts for event handling and update mechanisms to this assets is handled via drag and drop.

[1] `https://github.com/OpenKinect/libfreenect2`
[2] `https://github.com/OpenKinect/libfreenect/tree/master/wrappers/python`
[3] `https://www.microsoft.com/en-us/download/details.aspx?id=44561`

The editing of scripts/code can be done in Visual Studio 2015, which is the state-of-the-art Windows IDE. We imported the source code into Unity3D and Visual Studio 2015, as a programming language we have chosen C# - simply for convinience.

Moving objects in a Virtual Reality

First of all, we needed models for a Unity3D-scene. Patrick Harms provided us with a working *Blender*-Model[4] of a coffe machine, written by one of his students.
To first understand the combination between our C# -Scripts and the Unity-Scene, we started out with simple spheres in a 3D-system. Those objects can be created very simple in Unity. Furthermore we found *Blender*-model that represented Hands in a 3D-space[5].
After integrating these models in a Unity-Scene we were able to compute events:
We used a haptical feedback to emulate the proximity of the hands to the spheres created in the beginning. Whenever the hands were moving closer towards the sphere, the sphere itself would change its color - depending on the *eucledian distance* to the *mesh grid* of the hands. In the Unity-Scene, the 3D-Model of two hands consists of a mesh grid of coordinates for each hand, which represent the shape of the hands, and textures that is rendered over these mesh grids.

To compute the proximity between a sphere and the hands, we could therefore simply compute the euclidian distance between the coordinates of the mesh grid and the coordinates of the sphere and were able to compute a haptical feedback from there on.

Tracking hands with the Kinect v2

The next step was to map the hand movement into the created 3D-Scene. The first sensor we used was the Kinect v2. It comes with a lot of out-of-the-box-examples on how to achieve hand tracking.

However, reality turned out to be a little more tricky than that: As displayed in 2.1, the Kinect naturally only supplies four positions for the skeleton of the hand:

1. The position of the *wrist*
2. The position of the *palm*
3. The position of the *Top of the finger*
4. The position of the *Top of the thumb*

As our initial goal was to be able to recognize several gestures of the whole hand, we now needed more advanced algorithms to estimate the detailed position of the hand from this data. The last approach we found to be quite promissing was [3], coming from Mircosoft itself. Unfortunately, the time in this Practical Course was not enough to establish a complete working version of this algorithm for our project needs.

[4] Blender is the free and open source 3D creation suite.
[5] `https://clara.io/view/3e6923db-407c-4e0c-a253-2f564dfcd152`

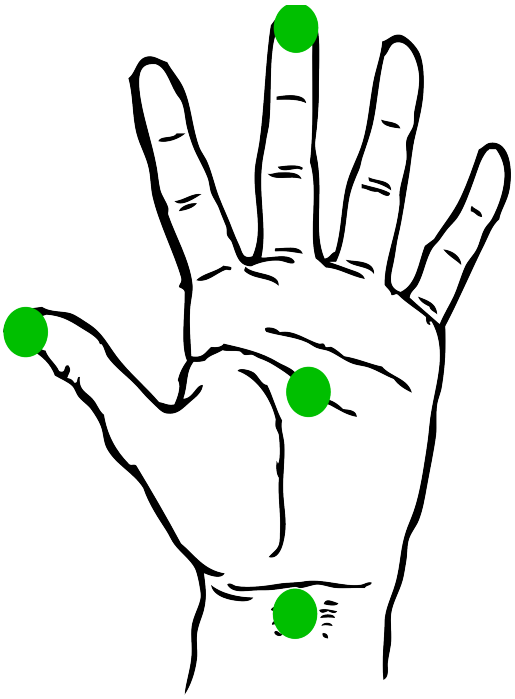

Figure 2.1: The Kinect naturally tracks four points in the hand of the user: The wrist, the palm, the top of the fingers and the thumb.

2.2 Combining the two techniques

We now have a coffee machine in our virtual reality, we got the 3d hand model from.
The hand model has full stretched fingers, which isn't perfect for operating a coffee machine. Therefore we transformed the model to a hand with a pointing finger. We used basic transformation operations in Blender for this. The export can be a bit tricky, because the exported model is not equal to the one you see in blender. For our case this meant, that it wasn't enough to mirror the hand in Blender and export it to get a right hand from a left hand. We also had to invert the normals.
For the positional tracking of the hands we just used the positional information of the palm. The other coordinates were used to coordination of the movement of the hand itself.
To operate the coffee machine we had to track when the hand is touching the coffee machine. We pursued 2 approaches:

1. The first is to track the distances from then hand to the buttons manually. Therefore we pull a the list of buttons and calculate the euclidean distance of the palm,thumb,wrist and tip to these buttons. We used this to give some indication how close the hand is to the buttons. For the actual button push we used the build in collision detection in unity. To make this work we had to add a Box Collider to the Buttons and enable the isTrigger property. We also added a Mesh Collider to the hands, as well as a rigid body. If the rigid body interacts with the Box Collider, then it will trigger certain events: onTriggerEnter, onTriggerStay and onTriggerExit. We wrote a script for the buttons which will colorize the button green onTriggerEnter and return to the original color onTriggerExit.

2. Wo ist denn der zweite? :-D

2.3 Problems

Frame Selection

The Kinect v2 switches between 15 to 30 frames per second, depending on the lighting conditions in the room where it is used. Sometimes there are single frames in which the Kinect detects the hands in wrong positions or alignments. That made our hand models jump or shake from time to time and it became very difficult to move the hand models close to the relatively small buttons like that.

Due to that we implemented a frame selection: We saved the hand positions of seven following frames and calculated the median of those. This made sure that single wrong detected hand positions did not make the hand model jump uncontrolled, but of course also made the reactions slower and more sluggish. We tried the selection with less than seven frames as well, but seven seemed to be a necessary amount to smooth out irregularities.

Another possibility we did not implement is to have a window of seven frames and build the median of it. With each new incoming frame this window could be pushed forward one more frame, so the overall displayed frame rate will remain the same.

Pushing the buttons

The function to colorize the buttons when a hand is coming close is quite buggy at the moment. For some reason the calculation which button is closer changes frequently. This could be due to difficulties to identify the hand to which the distance should be measured in the moment.
To make a more stable but equally simple solution we could add bigger Box Colliders around the buttons and colorize the buttons on the onTriggerEnter event.This wouldn't mess with the other function, because the onTriggerEnter event of the inner Box Collider would remain intact.

Unstable hand orientation

The hand orientation is unstable in certain situations. Firstly we had a problem when we got to close to the kinect sensor. In this case the orientation gets more and more unstable, which is kind of expected due to the working distance of the kinect. We also had the problem that sometimes the up vector of the hand changes sign and therefore the hand will do a 180° rotation around the wrist.

2.4 Experiences with the Leap Motion

As already mentioned above we just had little time to test the Leap Motion, but still we made some experiences with it. We installed V2 Desktop, a Leap Motion SDK with several predefined scenarios to test the leap and its features. This one is plug-and-play and easy to use: We put the leap in front of the keyboard and opened the software. After choosing a scenario, 3D models of your hands are displayed on the screen and the movement of your hands and fingers are tracked a lot more accurately than with the Kinect v2.

The only problem we spotted is when you turn your hands with the palm facing each other and the thumbs pointing up - so that the fingers closer to the leap hide the other ones. Some motions are not

Figure 2.2: Look of the 3D hand models used in the Leap Motion software [1]

tracked correctly than. This problem could probably be solved by fusing tha data of the Kinect with the leaps data and both sensors filming from different angles.

2.5 Fusion/Switching between Kinect and Leap

Our first attempt was to switch between the both sensors so we can use the more accurate Leap when a persons hands are close enough to be detected by the Leap sensors and otherwise use the Kinect data.

This is the easiest way to have the advantages of both technologies fused in one project, but of course it probably is not the best one.

A problem with this approach is to fit both coordinate systems together, so the recorded data can be adjusted properly in the right positions relative to the coffee machine. Therefore you would also need a static setup which we didn't have due to limited space.

Another possibility would be to really fuse both sensor data to increase the accuracy in all three dimensions, especially to prevent the hand models from being unstable while the hands are rotated. Ideally it would also be possible to import the already implemented, completely movable 3D hand model coming with the Leap Motion software, so you can really track and see the movements of the single finger joints.

CHAPTER 3

Results

In the Practical Course Data Fusion we successfully established a development environment for the use of a kinect v2 sensor and programmed a demo application, where we are able to control a coffee machine with our tracked hands in a virtual environment. For the setup of the environment we had to install the kinect v2 SDK, Visual Studio 2015 Express and Unity3d, which is decribed in 2.1. The general procedure to make the coffee machine operable is described in 2.2. We were able to show that it is possible to create a small virtual reality application with the kinect v2 in a short amount of time using freely available libraries and tools.

- screenshot kinect usage of CM
- screenshot leap usage of CM
- precision of the used technologies

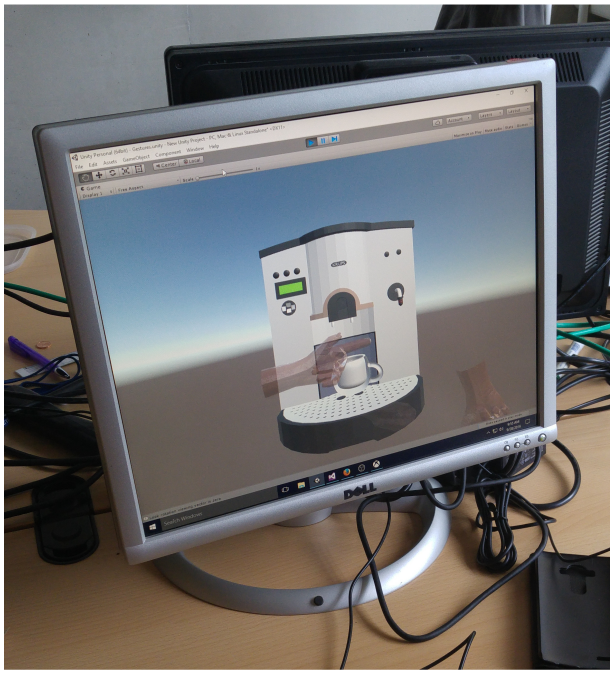

Figure 3.1: The finished scene in the 3D-Scene. The Hands can be seen in front of the Coffee Machine.

Figure 3.2: A user interacting with the scene.

CHAPTER 4

Discussion

Apart from the problems mentioned above we were astonished how quickly we were able to use our setup, move hand models via the Kinect in Unity3D and implement collision detection. Even though some things were a bit tricky, the software works quite well and can - for easy purposes - be almost used as plug-and-play. Unfortunately this can't be said for the software available for Linux PCs, Windows is in the lead here.

Of course there are several possibilities to further improve our project, some of them already mentioned above as well, e.g. the different frame selection or the fused usage of the Leap and Kinect. Our setting also might have been not ideal, e.g. the placement of the Kinect next to and not on top of the screen and the relatively small distances between the Kinect sensors and the hands.

It would also be a nice thing to stream the image of the screen to a smartphone and use a Google Cardboard to create the full virtual reality effect. A Google Cardboard is a cheap device where you can put your smartphone in and - by holding it in front of your eyes and with the help of the build-in sensors of the phone - experience virtual reality. This could improve the usage of objects like our coffee machine even more and make it feel and look a bit more real.

The whole virtual reality business is becoming increasingly important and is used more and more in our everyday life, so there are lots of projects and researches going on concerning this topic. Just to mention a few: there is the so called social virtual reality with its virtual reality chatrooms, of course there are computer games using VR, but it could also be used for medical treatments, e.g. exposure therapies [2].

CHAPTER 5

Conclusion

- possible to fuse data usefully
- possible to create a usable interaction this an object

CHAPTER 6

Additional

Bibliography

[1] Hand Models of the Leap Motion - Playground Flower Scene. Website. http://blog.leapmotion.com/wp-content/uploads/2014/11/playground-flower-scene.png; september 28th 2016.

[2] Hand Models of the Leap Motion - Playground Flower Scene. Website. http://www.techrepublic.com/article/10-ways-virtual-reality-is-revolutionizing-medicine-and-healthcare/; september 28th 2016.

[3] David Joseph Tan, Thomas Cashman, Jonathan Taylor, Andrew Fitzgibbon, Daniel Tarlow, Sameh Khamis, Shahram Izadi, and Jamie Shotton. Fits like a glove: Rapid and reliable hand shape personalization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5610–5619, 2016.