University College Dublin
Ireland's Global University

SHAKE THE FUTURE.

P2RV Report

Sign Recognition Project

Alexis Dupuis, Elsa Thiaville, Gustave Bainier & Yicheng Chen

Virtual Reality Specialization

Sign Recognition Project

Alexis Dupuis, Elsa Thiaville, Gustave Bainier & Yicheng Chen

Virtual Reality Specialization

January - March 2019

Summary

1 Introduction

1.1 Project's goal

The principal aim of this project is to create a tool that can identify Irish Sign Language (ISL) in real time and output it in English (speech or text) what was signed. This is a complex task because ISL, which is an official language of the Republic of Ireland, has its own grammar and syntax which is different to spoken languages such as English. Additional information such as facial expressions and distance of the signs from the body also have an effect on the meaning of the sign.

During our first interview with Anthony VENTRESQUE, we set more realistic goals four our project.

Minimal goals :

- The idea is to be able to record actors signing and transpose it to an avatar. With a database of signing avatars, it would be easier to produce content for sign language applications without resorting to an actor.
- With a realistic hardware installation, try to track body skeleton in 3D, hands rigs and if possible facial features.

Additional goals:

- Output joint movements in some exploitable ways for an ulterior sign recognition pipeline to be used.
- Explore on what makes a good avatar for sign language (natural movements, aspect...)

Therefore, we had a few questions before staring our researches:

- Do we need online processing and robustness?
  Online processing is a goal that will eventually be explored so we should aim for it if possible, but offline is fine for a proof of concept in the scope of this project. On of the main interest of online functioning is to have a direct feedback on what's recorded, and whether some part should be re-recorded. Likewise, robustness is one of the main goal to avoid the trouble of recording multiple times.
- Do we need robustness to various illumination/environments?
  It is not required to work "in the wild", since we have control over the recording environment.
- Can we consider for instance data gloves, markers, etc?
  The actor should be free of any embedded hardware if possible.

1.2 Readings

The following terms are often found in the papers and depicts the state-of-the-art in Sign Recognition:

- HamNoSys : a symbol-based language to describe the signs used in Signed Language.
- SiGML : an XML re-transcription of HamNoSys, to be used in computer applications.
- JASigning : a synthetic animation system for deaf signing, written in Java, taking SigML as input and producing motion data for an avatar.

- eSign editor : convert HamNoSys to SiGML.
- AnimGen : animation synthetiser, creating animation frames from SiGML and an avatar description, to be used as input of an avatar renderer.
- ARPToolkit : plug and play application that allows to create rigged avatars directly usable for deaf-signing (correct proportions and bones properties).

Many papers show restrictions in the use of avatars for sign language due to poor acceptation from the deaf community (when opposed to videos of human actors). [5] illustrates how the quality of the human-like movements are crucial for acceptation of this technology:

" Common remarks include "robotic", "unnatural", "stiff" and, as mentioned by one participant, "avatary". This feedback alongside the statistic that 90 % did not think the avatars looked natural demonstrates that there is still a lot of work to be done with regard to the avatars movement. "

Even though this is probably out of the scope of this project, building convincing avatars requires thorough studies on human-like movement. One of the trends in the field of animation is the use of skeleton-less motion capture and rigging, that we will obviously not explore here.
Requirements for a realistic avatar usable for sign language transcription are also described in [1], and are far more advanced than the use of a simple re-targeted rig.

1.3 First observations and predictions on the potential difficulties

Skeleton tracking belongs to the vast field of motion capture (MoCap), and it is always challenging to obtain robust results without markers or suits. The MS Kinect could be useful, but Microsoft has recently abandoned the Kinect v1 and v2 projects which means sources of documentation could come to disappear.

There are many facial features tracking solutions based on OpenCV that should be exploitable. Hand and fingers tracking on the other hand should be very difficult, especially considering the precision required to recognize a sign and the variety of signs used. The use of Leap Motion could be considered since we have control of the recording environment (elimination of IR interference sources can be considered). However, it is restricting in terms of hand positioning and could show lack of robustness to various situations

The synchronization of the many sensors and their outputs are also a potential source of difficulties and issues.

2 Software/Hardware testing and potential methods

2.1 OpenPose

Among the very few open source libraries in this field, OpenPose offers a solution for robust 2D human pose tracking in the wild. It seems very robust and could serve as a great starting point for 3D pose reconstruction.

2.2 Tf-OpenPose

Tf-OpenPose is an adaptation of the core of OpenPose to run with Tensorflow 1.4.1+ (working with CUDA versions <5 and supposedly more computationally efficient). Among other dependencies, it should be noted that Tensorflow only works with Python 64bit.

Our first successful install was on ROS (Ubuntu 16.04) in a virtual machine, this would have been the ideal environment for development but it is way too slow on execution (only half of a CPU available...). Therefore we managed to successfully installed it on Windows, first without the GPU acceleration, then after installing CUDA 4 and cuDNN with a GPU acceleration. This gave us a decent real-time body tracking varying from 2 up to 15 fps on our different computers.

2.3 MS Kinect v1

Alexis started experimenting with the MS Kinect in the hope to find body tracking methods, either standalone or that could be combined with 2D human pose tracking. The developers' toolkit shows attempts at body tracking but the documentation related to Kinect v1 has been lost so it is hard to know more precisely what to expect. Some code is still available that could be used to familiarize with the Kinect and to try to understand what is really possible with this hardware in terms of human body tracking in 3d.

[4] details a skeletal tracking pipeline used by the Kinect to estimate the human pose, meaning even a Kinect alone can produce an estimate of the body pose. We can imagine a fusion of the estimations made by multiple Kinects from multiple angles could obtain a robust estimate of the full body pose.

2.4 Leap Motion

Tested with Unity 18.01, this has yield deceiving results. Massive issues when tracking fingers when the hand is not facing the camera (fingers occluding each others). Some occurrences of false-positives (detecting a hand when there are none) or true-negatives (not detecting a hand that is present). Robustness issues in general and re-detecting a hand after it was lost can make a few seconds and requires to stop to re-position the hand right in front of the device

3 Body tracking

3.1 3D scanning with the Kinect v1

3.1.1 Setting up and testing the Kinect SDK.

Setting up the Kinect 360 on Windows 10 requires to install the Kinect SDK 1.8 by Microsoft (`https://www.microsoft.com/en-us/download/details.aspx?id=40278`). It will install the drivers and Kinect Studio v1.8, the software that enables to use the Kinect on PC. Downloading the Developer toolkit as offered during the installation allows to see example of the Kinect abilities. However, the online documentation seems to be lost.

This great page: `https://homes.cs.washington.edu/~edzhang/tutorials/index.html` offers an alternative to the lost documentations and serves as a tutorial on the 4 basics of Kinect: RGB stream, Depth stream, RGB+D handling (to output a coloured pointcloud of the environment for instance), and Skeleton tracking. You can download the source codes for OpenGL for each tutorial.

The installation requires to set up a Visual Studio environment with OpenGL. I used VS2015 Enterprise, and the NuGet package manager (Tools→ NuGet Package Manager→ Manage NuGet packages for Solution) to install nupengl.core. The projects also requires GLEW and freeglut. Installation trivias on Windows are explained during the tutorials. As detailed, you should:

- Get GLEW binaries for Windows here: `http://glew.sourceforge.net` by clicking on "Binaries Windows 32-bits and 64-bits".
- Get freeglut from here: `https://www.transmissionzero.co.uk/software/freeglut-devel/` by clicking on "Download freeglut 3.0.0 for MSVC".
- Put the x64 dlls in C:/Windows/System32 and the x86 ones in C:/Windows/SysWOW64
- Put the lib files at the same place as opengl.lib on your computer (there might be two of them under the v8.1 and the v10 folders if ou are on windows 10).
- Put the .h files of the include folders where GL.h and other OpenGL headers are on your computer (same remark as above).

At that point the skeleton tracking code might still ask for a SDL.lib. This shouldn't be the case since we are using OpenGL instead of SDL. To solve this edit the *KinectTutorial4.vcxproj* file to remove the two lines referencing *SDL.props*.

The skeleton tracking project outputs both arms as an example. The tracking works fine but is not really precise, and suffers from robustness issues when both arms are close to each other

and hard to distinguish. However, it is computationally efficient and immediate (no initialization or noticeable convergence phase). The code makes the output rotates to see the 3D, but you can change the rotation speed at the last line of *rotateCamera()* in *main.cpp*. Here are some example images of the tracking in action:

Figure 1: Arms tracking with the Kinect : front view (the arms bones appears in red).

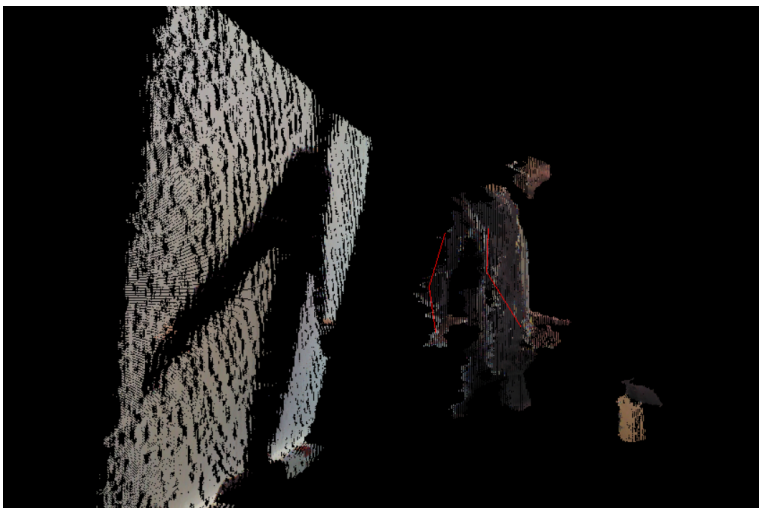

Figure 2: Arms tracking with the Kinect : side view (the arms bones appears in red).

The 3D poses of the 20 joints can easily be collected: see Figure 13.

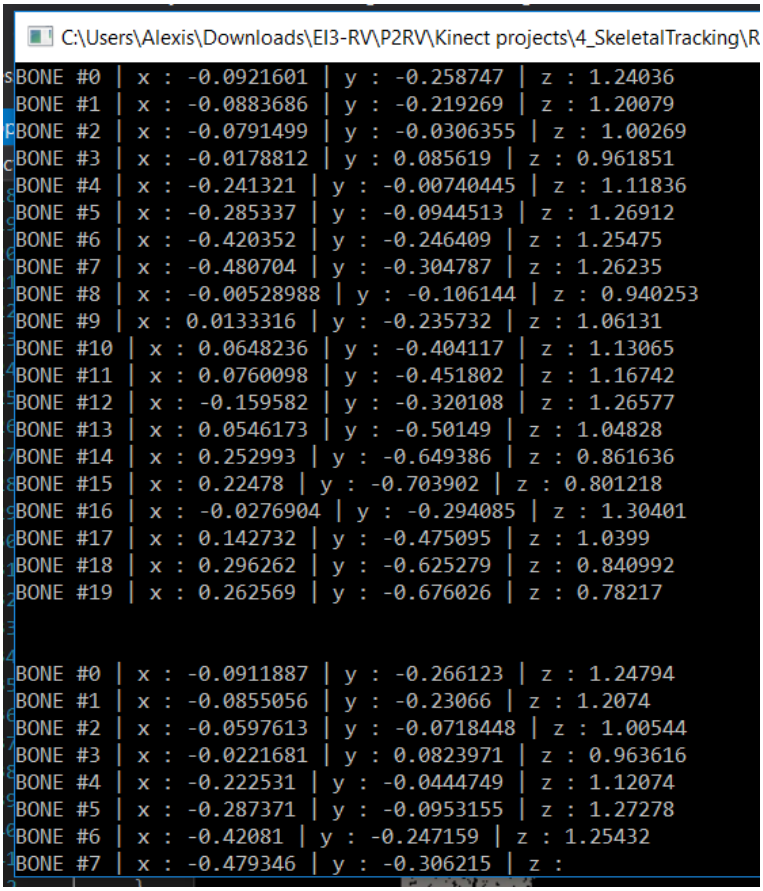

Figure 3: Body tracking with the Kinect : 3D pose data output.

The tutorials also provides the necessary code to stream depth maps, which means tf-OpenPose's output could be used to determine a 3D pose manually by identifying which depth pixels belongs to which body joint.

The next step would be to try writing a Blender plugin that read 3D joints positions and outputs an animated rig.

3.1.2 Complete motion capture framework with Kinect and NI Mate.

NI Mate is a software that deals with Kinect's skeleton tracking. It basically does cleanly what we saw in the section above, and for both Kinect v1 or v2. Other sensors are also available, and a paid version allows to combine multiple sensors into a single robust 3D skeleton output. It also comes with a Blender plug-in that transfer those joint positions into Blender. By using those, we can easily reconstruct a rig usable for Blender avatar animation. This plug-in might also be useful to understand how to transfer data into Blender. Some retro-engineering could be very useful for writing our own plug-ins, and have the comfort to work with Blender for the avatar animation.

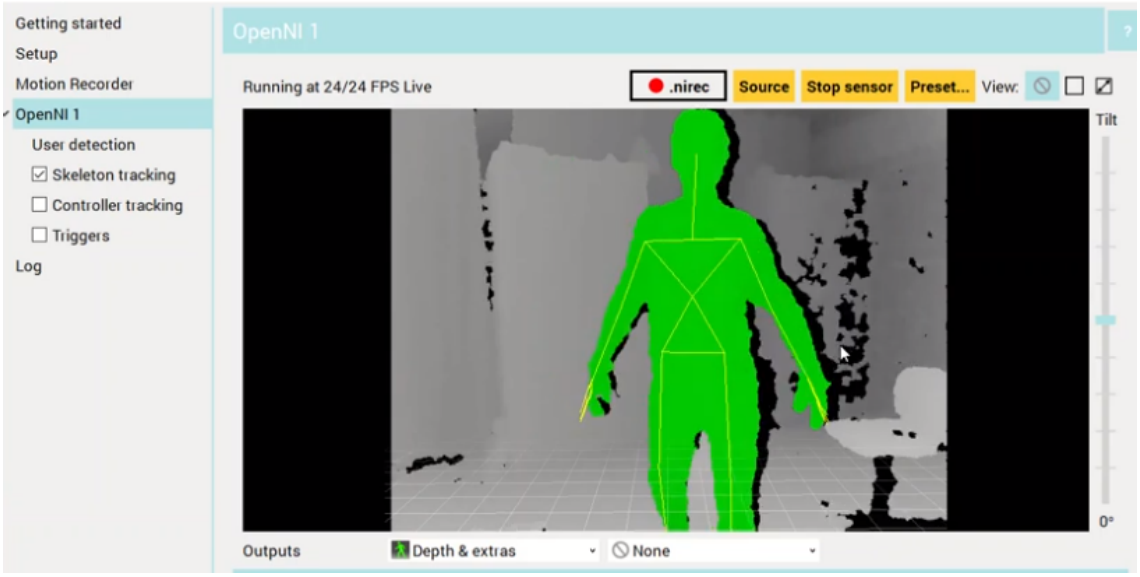

Figure 4: NI Mate allows to directly use the Kinect output -including the skeletal tracking- without having to code from the SDK as we did in the previous section.

NI Mate provides almost everything to do basic motion capture from a simple Kinect. I thus set up and tested the whole pipeline: from Kinect's input to Blender rig animation. A video of the result can be visioned here: `https://www.youtube.com/watch?v=vwcxfD92a9c` .

As you can see this is fairly robust for "homemade" MoCap. It only glitches with the legs at moments when they are not fully seen. Better results might be obtained with Kinect v2 and/or by using the payed version of NI Mate with multiple sensors.

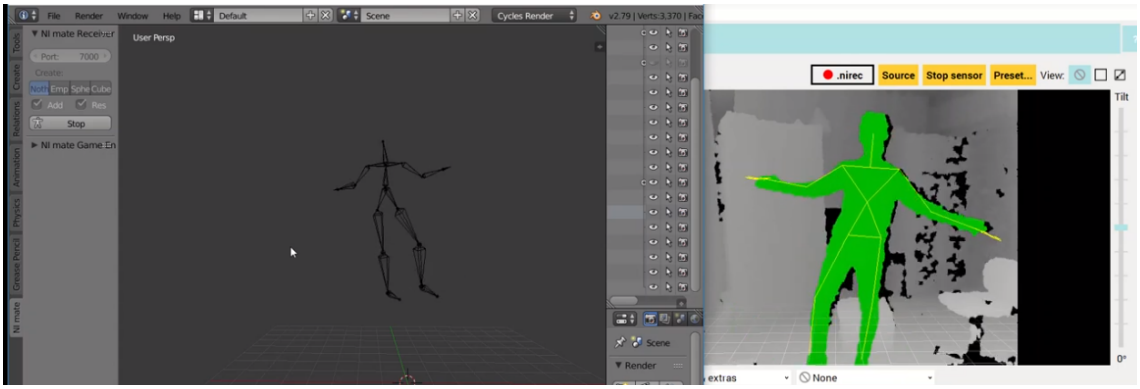

Figure 5: Using the blender plug-in we can output joints pose in Blender. We then create an avatar rig and retarget NI Mate's joints to the rig's joints to animate.

We then created an avatar in low-poly using Blender and linked it to the animated rig to

illustrate those results. The avatar is convincing and the Kinect is able to track correctly as long as the movements are not too complicated (from a curled up position for instance) and we are mostly facing the camera (spinning in place looses tracking of some bones, mainly because of the occlusion). A video of the result can be seen here: `https://youtu.be/qprEROIUefg`

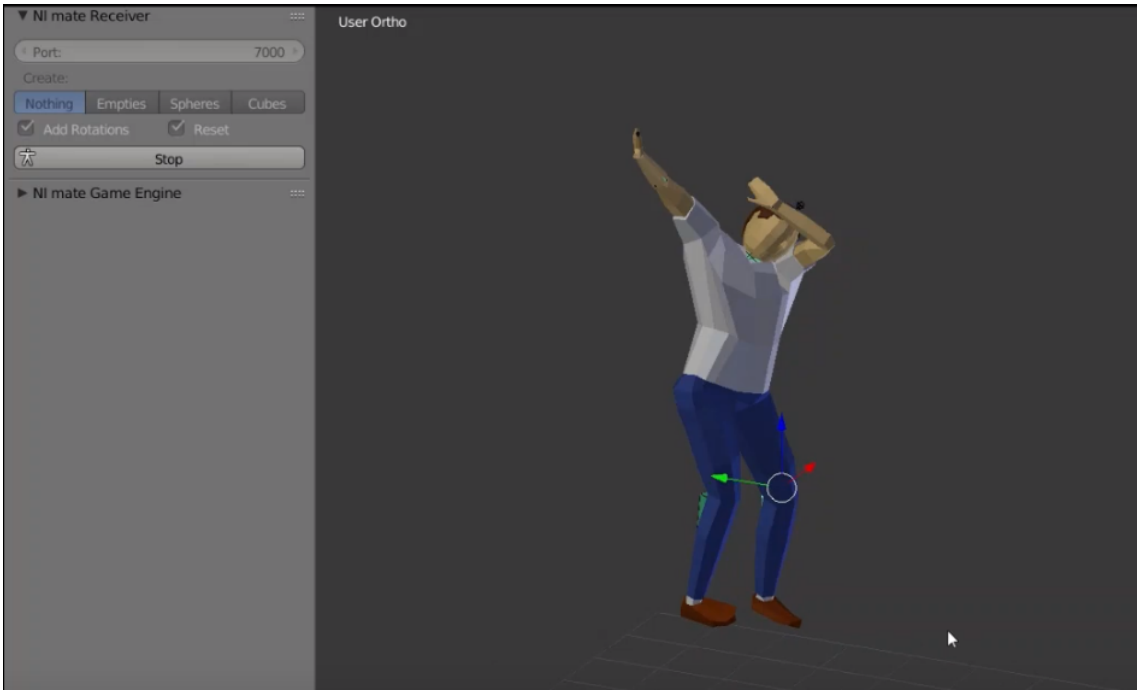

Figure 6: By adding a low-poly avatar that we link to our rig, we can illustrate a full body-tracking pipeline using the Kinect v1.

3.1.3 Inter-Process Communication and re-creating the Kinect tracking pipeline from scratch.

Interest of IPC for the project

In the previous subsection, we used NI Mate to create a complete body-tracking pipeline. However, we don't have much control on some parts of this pipeline because we ue the software NI Mate to deal with the Kinect data and its export to Blender. We would like to be able to create our own full pipelines for motion capture in real-time. This requires handling the communication between various processes

(1) "We have a C++ algorithm that extracts Kinect's data and produce an exploitable output, we want to create a Python plug-in that collects this output to display 3D markers in Blender. How can these two process communicate in real-time?"

(2) "We have two cameras used to estimate the pose of a body in 2D. We use these estimates to triangulate the 3D pose. These cameras are linked to 2 different computers that need to share their output in real-time, how do we achieve this ? "

Those are questions that need to be answered to go further in the project. This problem is called Inter-Process Communication (IPC) and is an extremely vast subject with many potential solutions. For this project, we choose to create a python server that the various processes and plug-ins will connect to as clients. Data messages will be sent through sockets: via localhost for *(1)* and via internet for *(2)*. The goal of studying IPC was to be able to theoretically turn every working solution into a full pipeline. However, due to a lack of time, only the Kinect solutions were turned into full pipelines.

Sending Kinect data to a python server.

We choose to re-use the work done in subsection 3.1.1 and to answer the question in *(1)* to illustrate the functioning of our IPC tools. This leads to the creation of a full pipeline using Kinect skeletal tracking, similar to the work done in 3.1.2 but entirely controlled by our own codes.

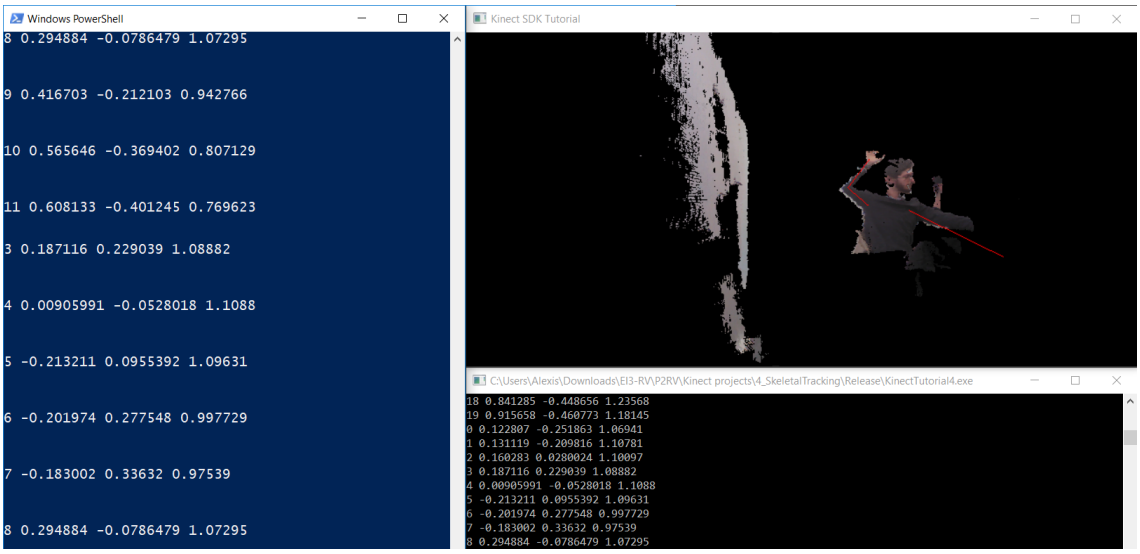

Figure 7: We send the joints poses tracked by the Kinect on Visual Studio to a python server through sockets. Note the integer added to identify the joints in the beginning of each message.

A first python script called "server.py" is used to test the reception of the data. Our skeletal tracking project from 3.1.1 has then been modified to add a publisher function. This function connects as a C++ client to the python server and send the joints poses. The data have to transit

from C++ to python and through the socket (*i.e.* in bits) without being altered. We use string messages (converted to binary during transition) and do some more conversions on both ends.

When re-using this code, you also have to change the IP used by the socket (line 272 in *Kinect-Tutorial4.sln*) to your local IP. Be sure to first launch the python server and then run the Visual Studio project.

Creating a Blender script.

We want to use Blender as we did with NI Mate. The idea is to create our own python plug-in that should be run in the Blender console. It has to play the role of the python server, and also animate 3D markers corresponding to the various body joints. There are many specificities in this script due to the fact that it should:

- Control elements in Blender: this is done using the *bpy* library that gives access to a specific syntax and set of functions.
- Act as a server: this is done as described in the section above using the *socket* library.
- Be able to run without blocking the Blender renderer, since we want to visualize the results in real-time. For this reason, it should be run in a dedicated thread on our GPU, that we spawn and use with the *threading* library.

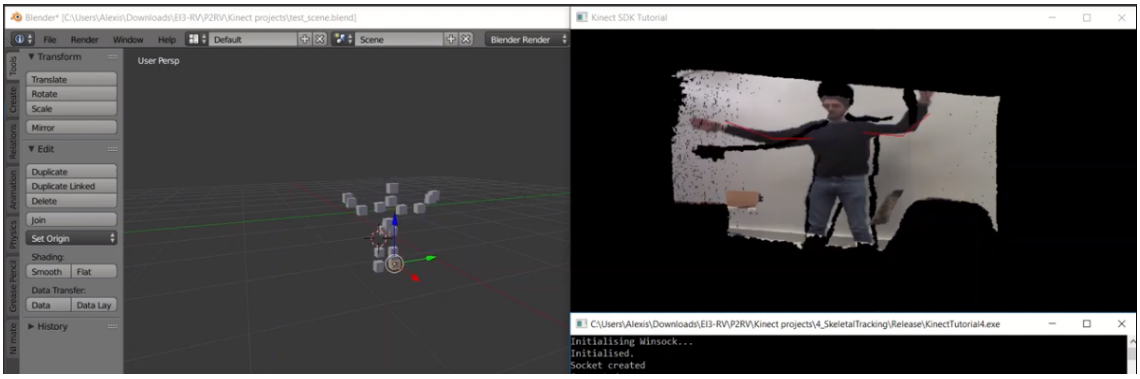

Figure 8: Using a Visual Studio project to extract Kinect's data and our own blender script, we can animate joints in Blender just as well as when using NI Mate.

The result can be seen in video here: `https://youtu.be/zUHTrmORSrg`

3.2 Combining tf-openpose with the Kinect.

Instead of using the Kinect skeletal tracking, we can think of a way to use Kinect's depthmap stream and tf-openpose's 2D joints tracking to spawn a pointcloud corresponding to the joints position in 3D (obtaining the depth on the depthmaps).

tf-pose-estimation outputs JSON files containing 1 array for each detected skeleton. This array ("pose_keypoints_2d") indicates the positions of the joints in pixels on the image. The joint i is detected at the pixel of coordinates (u_i, v_i) (with $(0, 0)$ in the top left corner).

- $u_i = pose\_keypoints\_2d[i]$
- $v_i = pose\_keypoints\_2d[i + 1]$

To determine the 3D point (x_i, y_i, z_i), we start by determining the 2D point (x_i, y_i):

1. Calibrate the webcam to get intrinseque params (pinhole model).
2. Use this camera model to get x_i, y_i from u_i, v_i.

Pipeline:

1. Run tf-openpose with publish mode to output JSON files :

   python run_webcam.py –model = mobilenet_thin –resize=640x480 –camera=0 –output_json output/

   tf-openpose doesn't have a publish mode by default. Using the modifications proposed by user ArashHosseini on github, we can output the results in this JSON stream. See `http://tinyurl.com/y364pn8c`.

   Something to note here: this usually outputs a stream of JSON files numeroted in their names. However, the frequency of these publication is expected to be lower than the frequency of our Kinect program, meaning we can't just ask our Kinect project on VS to seek for a new JSON file at each frame, since there might generally not be one already. We use a little trick by modifying the publisher script in tf-pose-estimation to always publish in the same JSON file. We will then always open this same file, and sometimes work multiple times in a row with the same 2D pose even though the depth pose of the Kinect has changed slightly. If tf-pose-estimation run at an acceptable framerate (10+ fps should be good) we won't have

the time to move our body fast enough between two updates of the JSON file for this to create pose errors. This trick is the best solution anyway, along with potentially reducing the Kinect inputs framerate.

2. Create and run a Kinect project that can stream the depthmap. It also needs a function that parses the JSON files created from tf-openpose. We then get the depth of the pixel at u_i, v_i in the depthmap to get z_i. Finally, we compute a full 3D point from these using Kinect build-in functions, and send them as a client through a socket.

3. Run a Blender plug-in as a server, similarly to what was done in 3.1.3.

3.2.1 Streaming Kinect's RGB camera in tf-openpose.

Our first issue is that the Kinect is not recognised as a webcam, so tf-openpose can't use it to stream image data. Note that this alone will force us to search for difficult solutions on a hardware that has completely been abandoned and that is heavily outdated in terms of software support (and also a bit outdated in terms of characteristics). On the other hand, the Kinect v2 is directly usable as a webcam and still has great support (even though this is believed to change in the years to come, where Kinect Azure will probably be the go-to solution...). This is an example of how hardware limitations actually increased the difficulty of this project even though it also reduced the expectations on the results...

There was apparently a fix existing to register the Kinect as a webcam (KinectCam.ax) but it seems to be outdated and doesn't work on our environment. If we use an external camera, the intrinsic parameters + the extrinsic (transform between this camera and the Kinect's camera) will lead to imprecisions. The joints are then usually detected slightly next to their real position, leading to frequent incoherences.

We could stream the Kinect manually in our VS project, using NUI_IMAGE_STREAM and some openCV conversion:
`https://stackoverflow.com/questions/28072576/displaying-kinect-streams-using-opencv-c`

But then we don't know how to send this to tf-openpose. This could however be used for determining the Kinect's intrinsic parameters...

Alternatively, we could re-write tf-openpose partially to allow Kinect streaming, and hopefully both programs can stream the Kinect simultaneously. This imply that we need to stream the Kinect RGB data in python (the Kinect SDK is strictly C language).

A first failed attempt with pykinect.

For this, we can use the **pykinect** package. Unfortunately, it is now only available as a Visual Studio tool from PTSV (Python Tools for Visual Studios). So we need to start a new python project from visual studio and to install PTVS.

Despite thorough researches, there doesn't seem to be any tutorial or even a good set of examples on how to use pykinect. There may be multiple reasons for this. First, it was often used as a part of pygame to develop full Kinect applications (including windows, displays, interactions etc...). Second, it is not used a lot anymore since Kinect v2 and pykinect2 are available now. And of course, the main reason is that Microsoft doesn't sell Kinects anymore. Since Visual Studio is also a Microsoft product, any real documentation seems to have been lost.

Anyway, we found on an old forum a first example of RGB streaming via the kinect, that we can use as inspiration. But to be honest, we abandoned here. The last thing we want is the complexity of Visual Studio for something as straightforward as python. Somewhere in the beginning of our script, VS started to complained it didn't knew any basic python library, because apparently you should give him a link to a python environment or whatever. Having no idea of why it was so difficult to write and run python when the Bloc-Notes text editor and a cmd window can do it just fine, we decided to seek for an alternative.

Alternatives: OpenNI or OpenKinect.

OpenNI is a set of open source tools to use the Kinect. It seems to have a python linkage available that can be easily installed using pip. But first we need to install OpenNI2 (and NiTE2), then do pip install openni to do the bindings. Alternatively, we can use **OpenKinect**, that is a little more specific than OpenNI since it focuses only on the Kinect while OpenNI also works with other depth sensors.

For some reason, the OpenNI.org domain seems to have been bought by Apple (!) since it now leads to Apple.com... The download links can still be found with some research (`https://structure.io/openni`) but with not much instructions.

We first tried to work with OpenKinect since it still has its wiki available: `https://openkinect.org/wiki/Getting_Started` `https://openkinect.org/wiki/Python_Wrapper`

But apparently the CMake version used then isn't compatible with current versions of CMake, making the wiki install instructions not usable as-is.

We thus decided to try our chance with **OpenNI**. We tried to display the RGB stream using OpenCV in the simplest way. Snippets found online only focus on depth stream, so we had to guess how to use a similar syntaxe for RGB stream. Note that the Kinect uses RGB while OpenCV uses BGR, so some conversion needs to be done before displaying if we want real colors.

Once we were able to stream the RGB data image by image using OpenCV and OpenNI, we created a new python script in tf-openpose that would use the kinect stream. We successfuly managed to use the Kinect as an input to tf-openpose:

python run_kinect.py –model = mobilenet_thin –resize=640x480 –output_json output/

Unfortunately when testing our whole pipeline, it appeared that browsing a Kinect channel was locking it, meaning that tf-openpose could not stream the RGB data in the same time as our VS project. We then updated the project to use only the depth stream (before this, the color stream was only used to color the pointcloud so this is only for aesthetics and could be left aside). However, the issue still remained. Our guess is that OpenNI completely locks the Kinect. If we launch the VS project first it works fine but tf-openpose won't find the Kinect. Conversely, if we launch tf-openpose first, the Kinect depth stream in Visual Studio appears empty. Basically this whole pipeline is bound to fail and we worked for nothing.

3.2.2 Using an additional webcam.

Back to stage 0, the easiest path at that point was to use a separate webcam for tf-openpose, but to do some sort of camera registration to reproject pixels positions in tf-openpose's output as if they were seen by the Kinect camera.

To calibrate the webcam, we use the python library calibtools that allow calibration on a video. This is very straightforward and avoid re-coding something as basic as camera calibration. However, since the Kinect is not detected as a webcam, we will have to use our previous streaming script and turn it into a video recorder using OpenCV's functions. All of this being done, we now have 2 videos for both our cameras (webcam and Kinect's RGB camera) and calibtools gives us the calibration parameters we needed.

We go back to the VS project and code the conversion of pixel coordinates thanks to these parameters. After a quick test, it appears that the VS project will freeze pretty quick. There is multiple potential reasons for this: either the GPU can't handle both tf-openpose and the OpenGL display, or tf-openpose and our VS project try to access simultaneously to the JSON file causing

an error.

It may be surprising to see the program not clearly crash or send any message if it was because of the simultaneous file-access, so we will assume its the first reason and disable OpenGL display to see if there are any improvement. Obviously we need some kind of feedback so this is a good time to go one step further and turn our VS project into a C++ client, sending all these joints data to a python server as we did for Kinect's skeleton tracking project.

Creating the Blender plug-in.

First results show that the whole pipeline communicates correctly and without crashing, and the output data seems to be coherent. The next step is to turn the server into a Blender visualisation script as we did before. However, tf-openpose is still making the computer suffer and it's hard to tell if we will be able to display anything in Blender this time.

Baware: if the program seems to hang forever after the "Hello" message was sent and received, it is most likely because output.json is empty. Be sure that it is not before testing without activating tf-openpose.

One issue we haven't address yet is that we don't really know which is the joints order in tf-openpose's output. That means even if the Blender script works, it will assign joints randomly as for now. The pipeline seems to hold at first with cubes moving when we do before the camera. However, it seems that:

1. The load on the GPU seems to be too heavy because the cubes stop responding after a moment, as if something had frozen (but the coordinates are still correctly sent).
2. Since the joints are randomly assigned, this looks nothing like a human obviously for now.
3. There were a lot of cubes that didn't seem to be moving: either they were simply the many joints not detected by tf-openpose, or they were detected joints that ended up on the wall behind me instead of on the correct point on my skeleton in the Kinect depth map (meaning our by-camera solution is still bad).

To assign each joint correctly, we need to guess the order each joints are sent in inside the JSON files. See figure 9. We then make the link with the identifiers sent through the socket messages by checking which identifier corresponds to which joint (see 10).

tf-pose-estimation/tf_pose/common.py

```
class CocoPart(Enum):
Nose = 0
Neck = 1
RShoulder = 2
RElbow = 3
RWrist = 4
LShoulder = 5
LElbow = 6
LWrist = 7
RHip = 8
RKnee = 9
RAnkle = 10
LHip = 11
LKnee = 12
LAnkle = 13
REye = 14
LEye = 15
REar = 16
LEar = 17
Background = 18
```

Figure 9: Joints order in tf-openpose's output.

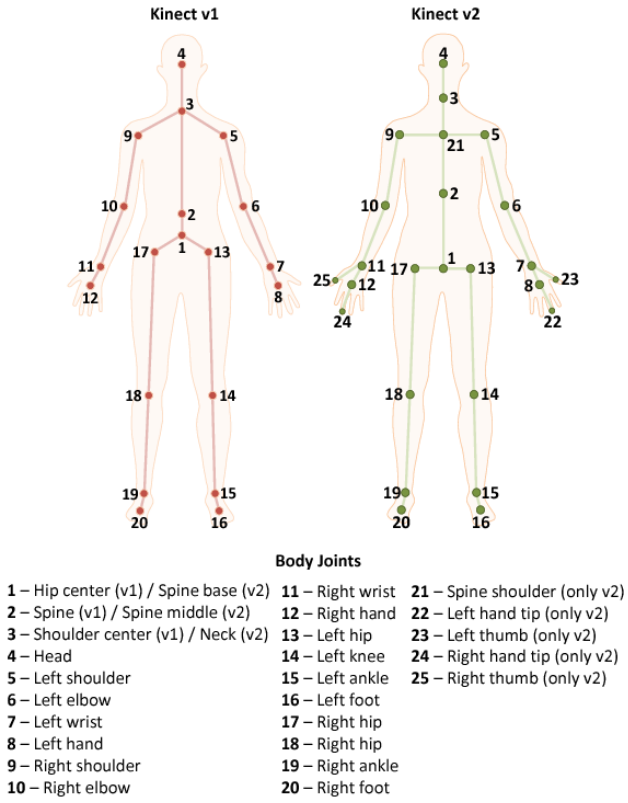

Figure 1. Body joints tracked by Kinect v1 and Kinect v2.

Figure 10: Joints identifiers in the Kinect SDK.

3.3 2D scanning and machine learning to guess the 3D pose

3.3.1 From a video to a 2D pose using a fork of tf-OpenPose

As previously noted, tf-OpenPose gives us decent results in terms of pose scanning. Yet, one problem is still remaining : it is only a 2D pose !

With the intend of using the output data of tf-OpenPose to guess the 3D pose, we use a forked version of the project (`https://github.com/ArashHosseini/tf-pose-estimation`) that outputs one JSON file per frame with the coordinates of every joint detected. It should be noted that this forked version only works with web-cam input, and thus we had to edit a bit the source code to make it able to read video.

Our data to estimate the 3D pose will be some videos where the whole body of someone is visible, because it will help the deep learning algorithm to have every joints at disposal. That is also why Gustave uses a video instead of the camera stream : he does not have enough space in his room to capture his whole body from his laptop.

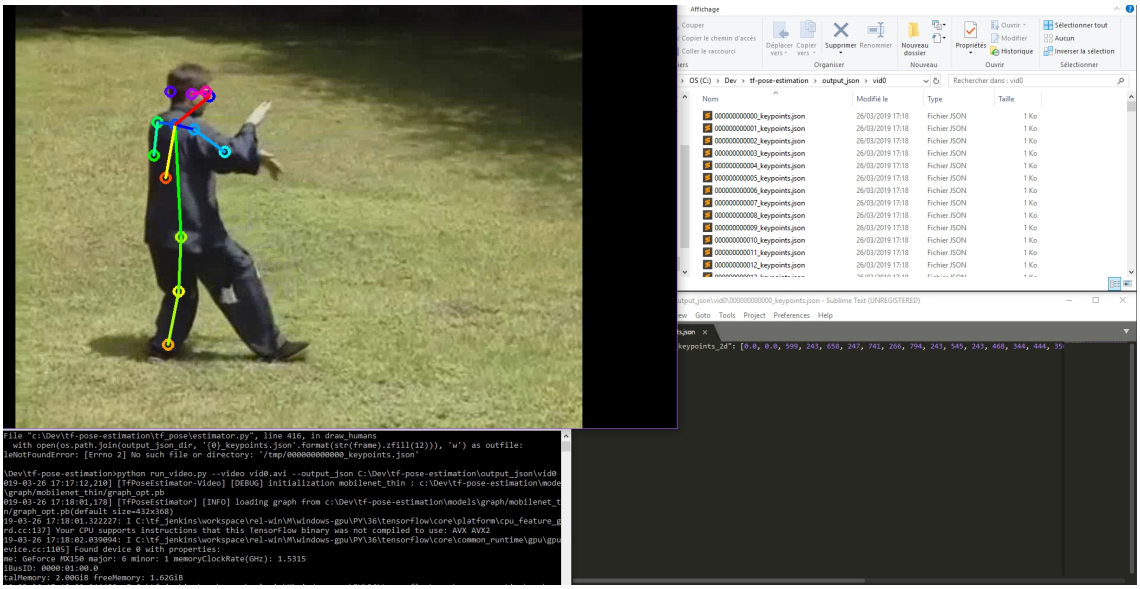

Figure 11: Body tracking from a tai-chi-chuan video with the forked version of tf-OpenPose with JSON data output on the right.

3.3.2 From a 2D pose to a 3D pose using 3d-pose-baseline

From this JSON output, Julieta Martinez, Rayat Hossain, Javier Romero and James J. Little suggest *"A simple yet effective baseline for 3d human pose estimation"* to guess the 3D pose [2]. We will be using their algorithm, an open-source and relatively simple deep feed-forward neural network, to transpose the 2D pose to a 3D one. This algorithm is clearly not the most efficient

that exist in the field, but it has the advantage of being very fast to train (<1 hour with GPU acceleration on Gustave's laptop), which is good for fast prototyping.

It should be noted that their code came with a few mistakes that we had to tackle, because they were working with inputs where the JSON joints were all perfectly in place, and the algorithm either crashed or guessed absurd poses when some joints were missing (see example below). We managed to tweak the source code so the algorithm is a bit more robust, mainly by initializing some variables at some default values.

Here are some example images of the 3D pose reconstruction in action:

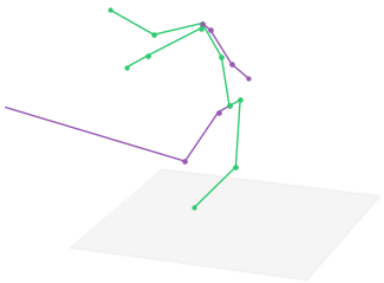

Figure 12: An incorrect 3D pose estimation due to an incomplete JSON joints input

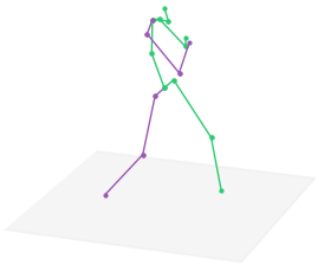

Figure 13: A more proper 3D pose estimation

A video of the result can be visioned here:

`https://www.youtube.com/watch?v=0XWj-baDvjY`

If we managed to get a robust output from the algorithm (it always outputs a 3D pose estimation), we can see than it is still very shaky and jumpy in its movements, and sometimes mistakes a joint to put in the front with a joint to put in the back, and sometimes the opposite.

At this point, the method is clearly not reliable enough to be useful in the context of a serious 3D pose reconstitution, but it could become way better in a near future ! As a matter of fact, during our project, a new algorithm was created by Keze Wang, Liang Lin, Chenhan Jiang, Chen Qian and Pengxu Wei that showed significantly better results by mixing the 2D output pose with the corresponding frame of the video. [3] Unfortunately due to the newness of this algorithm we only came across it recently and did not find the time try it properly.

3.4 3D scanning with multiple cameras

The third solution to do the body tracking, as far as we concerned, is using multiple cameras which recorded the movements from different perspectives, after that, we tried to integrate these different images to create the 3D model in the virtual environment.

We assumed that the user was in the center of the room, the camera no.1 was right in front of the user and the camera no.2 was in the right side of the user. We would like to combine these two images to have a full view.

With this idea, first of all, we wrote a script in Python that would handle the transforms between the first camera's frame (also assimilated to world's frame) and the second camera's frame. Absolutely, to fulfill this function, we provided the position of the camera in the world coordinate system and the direction of the camera, and the script would create a transform matrix and handle the transforms. the script also includes a JSON parser to get the tf-openpose outputs, and basic display in matplotlib (which is actually difficult because it should run uninterrupted in real-time).

The computer vision's problem behind this triangulation consists in extracting the point's coordinates in each camera frames from the pixel coordinates using the pinhole camera model, then projecting these points in a common frame (world's frame) and resolving the equation. This imply to either develop the matrix multiplication by hands to isolate the unknown (point's depth in world's frame), or to use an abstract solver that can handle equation resolution with unknown variables (something Matlab can do very well, but that is pretty tricky in python for instance).

Unfortunately we couldn't possibly test this as we only had one computer able to run tf-openpose fast enough. Even if we did have two computers, tf-openpose doesn't run at fixed fps, so synchronising the output streams is basically impossible. At this point of the project, it left us no choice but give up this idea. However, a possible solution could be to create an OpenCV script that would collect and concatenate the camera streams. then a single instance of tf-openpose would run this stream and detect the person on both angles as if it was two persons. By changing the way we parse the JSON output, it would be possible to use this single JSON output to do the triangulation from both views.

4 Hand and face tracking

4.1 Face tracking

We had found an algorithm that was trained to recognize one's emotions but Anthony has reminded us that the important was not collecting the emotions themselves but collecting the emotions features. We are not looking to know that one is astonished but rather that one's eyebrows are raising.

4.1.1 Emotion detector using Tensorflow.

After 60 epochs of training for the algorithm (it took almost two days on Elsa's computer -which is slow), Elsa ran the emotion detector algorithm on her computer.

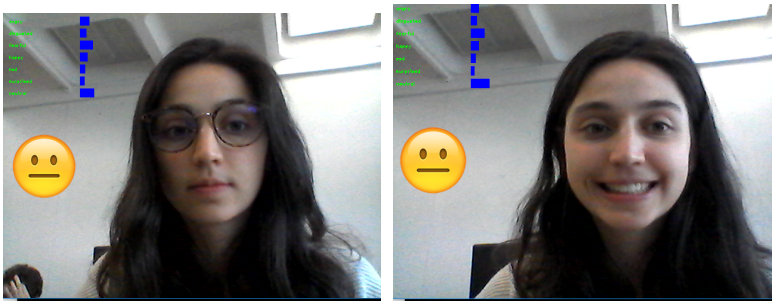

Figure 14: Neutral expression recognition : SUCCESS ; Smiling expression recognition : FAILED

It seems that it is not working very well. Apart from a neutral expression, nothing else is detected on Elsa's computer.
After this unsuccessful experiment, we looked into the code to see how the emotions were detected. It seems that the algorithm is only training from an image data set and not recognizing facial features. We looked for another algorithm that actually detects facial features and stores it.

4.1.2 Facial features detection

The installation process to use this algorithm was quite thorough. In order to compile, it required OpenCV, numpy, dlib and imutils. It seemed that Python was enough to make it work. yet, in the requirements list Anaconda was needed. We had to install all the requirements by hand, the command line *pip install requirements.txt* gave an error because the versions needed were not compatible with TensorFlow's.

This algorithm is extracting facial features coordinates from an image and storing it. One can access this "facial map" via Python indexing (assuming that zero-indexing with Python since the image is one-indexed):

- The mouth can be accessed through points [48:68] on the figure below
- The right eyebrow through points [17,22]
- The left eyebrow through points [22,27]
- The right eye through points [42,48]
- The left eye through points [42,48]
- The nose through points [27,35]
- The jaw through points [0,17]

Figure 15: Facial features coordinates

With a predefined image we had the following results :

Figure 16: Facial features detection.

The image is being stored on a file "images" in the project folder. We modified the call to the function in order to take a different image. We tried it with a picture taken with a computer's camera, where the resolution is lower. Elsa had her glasses on and still we had good results.

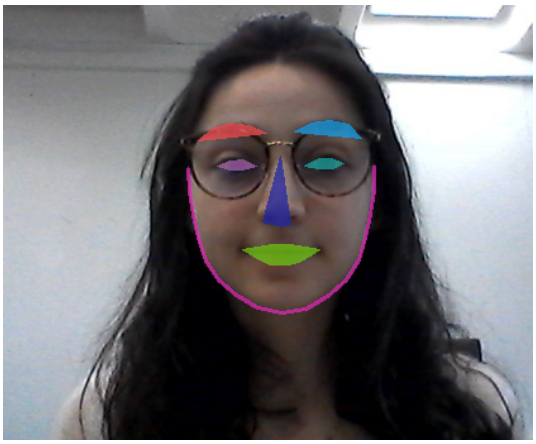

Figure 17: Facial features detection.

We then tried with two people. The algorithm will detect the two faces' facial features but only one by one. However, this will not be a problem since the avatar would be translating one person sitting in a room.

Figure 18: Facial features detection.

The next step in this facial features recognition process is to be able to recognize it in a video capture.We only would have to change the input data. Then we would have to communicate the data we collected to the server in order to represent those facial features on the avatar.

4.2 Hand tracking

An important point in sign language coding is the finger position. We have explored solutions with the Leap Motion. Yet, it has not been very effective. Indeed, we have encountered a lot of "false positive" situations with this device. Even though no hand was there, one was detected. We think it could be linked to the environment in which we tested it.

A solution would be to use OpenPose with the option "hand tracking" activated. It is supposed to be included in OpenPose but we have not been able to test it since OpenPose is not running on our computers.

5 Resources

The code used for the Kinect projects and for the beginning of the work on triangulation is available here:

`https://github.com/alexis-dupuis/Signed_Language_Detection`

- Kinect RGB streaming : this folder contain utilities used to stream the Kinect on python and to calibrate the cameras with calibtools.
- Kinect tracking projects : this folder contains the Visual Studio projects used for the motion capture with Kinect. It also contains the Blender scene used for our own tests.
- Mocap with NIMate and Blender : contains the Blender scene with the retargeted rig used with NIMate.
- Python server scripts : contain the scripts that should be used as servers in the diverse Kinect projects.
- tf-openpose : contains the run_kinect.py script that one can use in tf-openpose to stream the Kinect RGB data for 2D pose estimation.
- Triangulation with tf-openpose : contains the beginning of the work done for triangulation from 2D poses.
- videos : contains the videos linked in this report, in case they get removed from YouTube.

6 Conclusion

To sum up, we thought that this project was truly interesting and we really wanted to do the best we could to have consistent results. However, we lacked materials to do so. Only Alexis had the hardware to run OpenPose and use the Kinect. But in the end, even with these difficulty, we were ready to use it for the inter-process communication.

Moreover, the topic of this project is currently a research subject which made it hard to give original results. With the materials we had, we were well aware that we would not be able to bring exploitable results for Mr.Ventresque. We learned a lot, but unfortunately what we did is not likely to be used again.

Many thanks to Mr.Anthony Ventresque who has supported us during the whole project. He always tried to help us solve the problems and encouraged us to keep going even if we were not fully satisfied with our work.

References

[1] Vince Jennings, Ralph Elliott, Richard Kennaway, and John Glauert. Requirements for a signing avatar. *4th Workshop on the Representation and Processing of Sign Languages: Corpora and Sign Language Technologies*.

[2] Javier Romero Julieta Martinez, Rayat Hossain and James J. Little. A simple yet effective baseline for 3d human pose estimation. 2017.

[3] Chenhan Jiang Chen Qian Keze Wang, Liang Lin and Pengxu Wei. 3d human pose machines with self-supervised learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence (T-PAMI)*, 2019.

[4] Pushmeet Kohli and Jamie Shotton. *Key Developments in Human Pose Estimation for Kinect*, pages 63–70. 01 2013.

[5] Robert G. Smith and Brian Nolan. Emotional facial expressions in synthesised sign language avatars: a manual evaluation. *Universal Access in the Information Society*, 15(4):567–576, Nov 2016.