Computer Vision Project – Motion Capture

Luca Cazzola, Alessandro Lorenzi

a.y. 2023/2024

1 Introduction

The following report describes a comprehensive computer vision project that focuses on several critical tasks related to motion capture. First, we examine standard output files from a motion capture system, visualizing human skeletons and rigid bodies in 3D using Python. We then tackle the problem of flickering in rigid body motion caused by marker occlusions, using techniques such as the Kalman filter and the Particle filters. Finally, we integrate motion capture data into Unreal Engine to animate a character and project the skeleton data onto a 2D image plane, applying the necessary geometry.

2 Tasks

In the following sections we analyze the approaches used to solve the individual tasks and we also present the results obtained.

2.1 Task 1

We have employed a structured approach to read, store and visualize data from CSV, C3D, and BVH files, each of which containing different motion capture data elements.
For the **CSV files** we extracted the x, y, and z coordinates for each joint of the skeleton, along with their connections. The data was then plotted in a 3D visualizer using the Matplotlib library, showing the relative positions of various bones and joints (such as the head, neck, shoulders, ...) and also both the connections along the markers and the trajectories of the skeleton. Similar to the skeleton, we read the rigid body data using a function to extract the marker positions, visualizing in 3D the markers and their connections. We have used and updated the `optitrack.csv_reader` [10] library. Figures (fig. 1) and (fig. 2) show one-frame plot result.
In the **BVH file** we parsed the file to retrieve the relative rotations, global positions, edges (bone connections), offsets (bone lengths), and joint names [8]. We store the data in variables and export in file. Then using the **C3D file** we read the file and extracted a tuple containing a list of dictionaries with the marker information for each frame and the labels: a list of marker labels. We store them in variables and export in file.

2.2 Task 2

To address the issue of flickering in the motion of the rigid body due to occlusions, two different filtering methods were implemented: **Kalman Filter** (KF) and **Particle Filter** (PF).
The **KF** uses the physical laws governing the motion to estimate the state of the system. It predicts the next state based on the current state and corrects the prediction using the observed data. We initialize the filter with the positions (x, y, z coordinates) of the markers over time, setting the measurement matrix as an identity matrix, the uncertainty in the motion model and the uncertainty of measurements too. The filter iterates through each frame of the input data, retrieves the current position, predicts the next state using the KF, checks and corrects for missing data using predictions, and finally it updates the KF model with corrected measurements (point) and reshapes and updates the position arrays with corrected data.
In the **PF** particles are initialized uniformly within the determined limits based on `non-NaN` values found in the initial marker positions. Each particle also receives an initial velocity within the specified

range, ensuring they start with a diverse initial state. The PF iterates through each frame predicts the positions of markers whose values are missing, using the positions of the particles that best fit the current observations and it updates the positions of all markers. In particular, we implement the filter such that it adds random noise to the positions and velocities of particles to ensure exploration and prevent particle degeneracy, updates particle positions based on their velocities, calculates the error between each particle's position and its associated marker's position and it assigns weights to particles based on these errors, with higher weights indicating better fits to the observed data. Then the resampling selects particles based on their weights to form the next generation of particles, ensuring particles with higher weights have a higher chance of being selected. We have successfully implemented and applied both the filters to the rigid body and analyze the improvement trough plotting comparison (fig. 3) (fig. 4) (fig. 5) (fig. 6) [3]. Both 2 filters are simple ways to mitigate the markers loss due to occlusion, perhaps the PF's stochastic nature and higher computational cost makes it a bit worse compared to the KF solution, which is both faster to compute and retains better the body's geometry.

2.3 Task 3

Inside an **Unreal Engine 5 (UE5)** virtual environment we want to achieve 3D to 2D projection of joint positions onto the camera plane. UE5 makes it possible to interact with level (scene) components either via C++ code, or **blueprints** (visual representation of code functions via node graphs). We properly modelled the scene inserting 2 core blueprints, one containing our main actor and the other containing the camera (fig. 7). Using the **animation retargeting** feature provided by UE5 we were easily able to map the provided animation onto another free skeleton from **Adobe Mixamo** characters. After properly positioning actors in the scene a **LevelSequencer** component allows to capture a video using the virtual camera. Using the blueprint engine together with [4] we implemented a script to extract data in **Json** format (available for visualization at [5]) :

- **bones** ($\forall$.frames, $\forall$.bones $\in$ skeleton) : $\{X, Y, Z\}$ position, $\{X, Y, Z, W\}$ rotation (quaternion), bone name.
- **skeleton base frame** ($\forall$.frames) : $\{X, Y, Z\}$ position, $\{X, Y, Z, W\}$ rotation (quaternion).
- **camera** (once, camera is fixed) : $\{X, Y, Z\}$ position, $\{X, Y, Z, W\}$ rotation (quaternion), FOV, aspect ratio.

Moving locally with python, we used **openCV** functions to compute the 3D to 2D projection of joints. UE5 and openCV use 2 different coordinate systems: UE5 is **left handed** while openCV is **right handed**. Change of basis can be performed by using **homogenous transformation matrices** as in [7]. Knowing the world coordinate references to both the camera and joints we just need to extract the **camera intrinsics** and compute the projection. Camera intrinsics are generally extracted via camera calibration, but as this is a controlled environment with no distortion we can directly compute them with some algebra. computing the transformation [9] for each point we're able correctly displace the skeleton on the image plane (fig. 8).

2.4 Bonus Task

Instead of evaluating results on Matplotlib or open3d it would be much better to forward data to a more suitable environment such as Blender. To achieve so we used as basis [1] [2] which provides a framework to build skeleton aware neural networks by interacting with Blender python APIs. There was actually not much work to do here as the repository is quite well structured. We just needed to understand the framework and feed it with our data (fig. 9).

3 Conclusions

We have successfully solved all the presented tasks related to the field of motion capture. In particular, we first analyzed and plotted human skeletons and rigid bodies in 3D, reading data from different types of input files. Then, by implementing both the Kalman filter and the Particle filter, we solved the problem of flickering due to occlusion. Finally, we interacted with Unreal Engine 5 to get the joints projected from 3D to 2D. We also connected our data to the Blender environment.

3.1 Code & additional material

Code available at [6]. A less formal presentation about the project containing also **videos and animated gifs** available at [3]

References

[1] Kfir Aberman et al. "Skeleton-Aware Networks for Deep Motion Retargeting". In: *ACM Transactions on Graphics (TOG)* 39.4 (2020), p. 62.

[2] Kfir Aberman et al. "Unpaired Motion Style Transfer from Video to Animation". In: *ACM Transactions on Graphics (TOG)* 39.4 (2020), p. 64.

[3] Luca C. *Luca C. portfolio*. June 2024. URL: https://www.lucazzola.it/mocap.html.

[4] Epic Games Inc. *Json Blueprint Utilities Plugin*. Feb. 2022. URL: https://www.unrealdirective.com/tips/json-blueprint-utilities-plugin.

[5] Luca C. & Alessandro L. *Blueprints*. Dec. 2024. URL: https://blueprintue.com/blueprint/_qn_vgvc/.

[6] Luca C. & Alessandro L. *Github repository*. June 2024. URL: https://github.com/lorenzialessandro/CV-project.

[7] Dario Mazzanti. *Change of Basis*. 2021. URL: https://www.dariomazzanti.com/uncategorized/change-of-basis/.

[8] Giulia Martinelli MotionCaptureLaboratory. *BVH reader*. 2024. URL: https://github.com/mmlab-cv/MotionCaptureLaboratory/tree/main/python/BVH_reader.

[9] openCV team. *openCV documentation - projectPoints()*. 2021. URL: https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html#ga1019495a2c8d1743ed5cc23fa0daff8c.

[10] Garth Zeglin. *optitrack.csv_reader*. 2016. URL: https://github.com/cmuphyscomp/hmv-s16/blob/master/python/optitrack/csv_reader.py.

A Appendix

A.1 (Task 1)

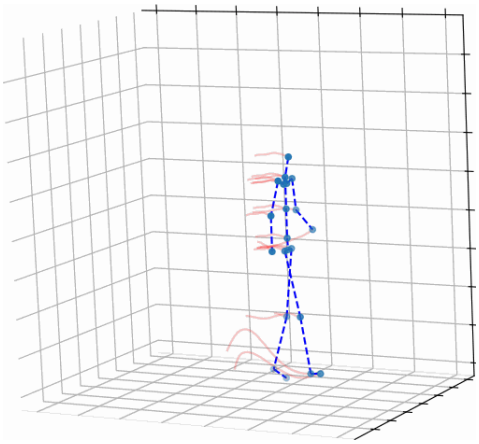

Figure 1: Skeleton plot from CSV file

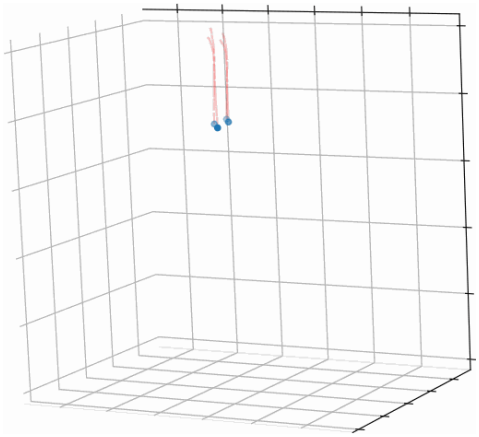

Figure 2: Rigid body plot from CSV file

A.2 (Task 2)

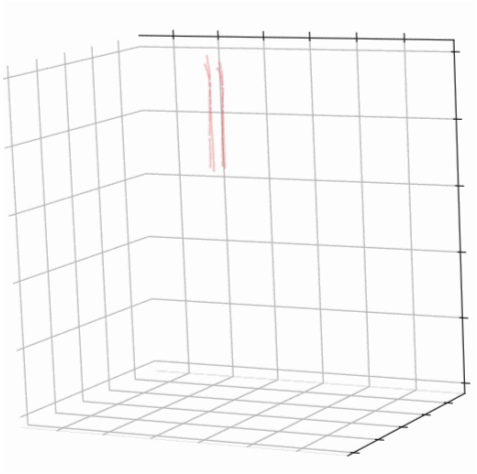

Figure 3: Rigid Body (original)

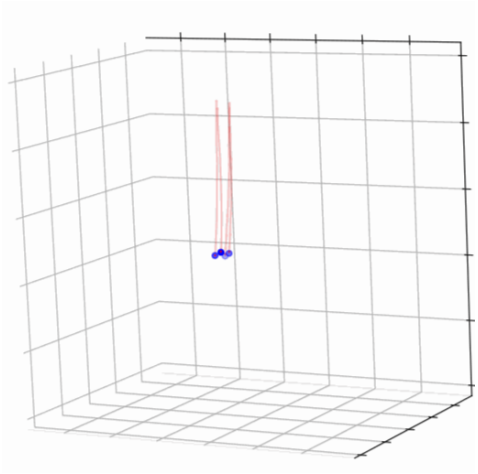

Figure 4: KF on Rigid Body

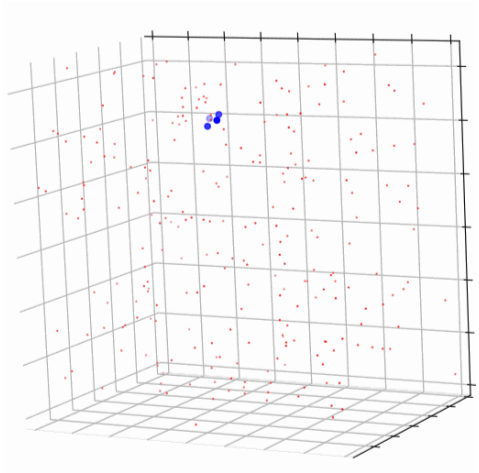

Figure 5: PF on Rigid Body (frame 0)

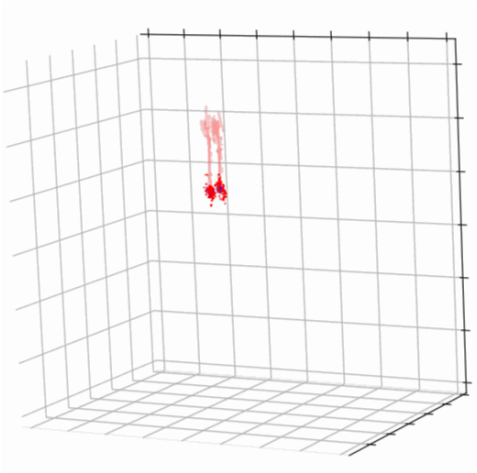

Figure 6: PF on Rigid Body

A.3 (Task 3)

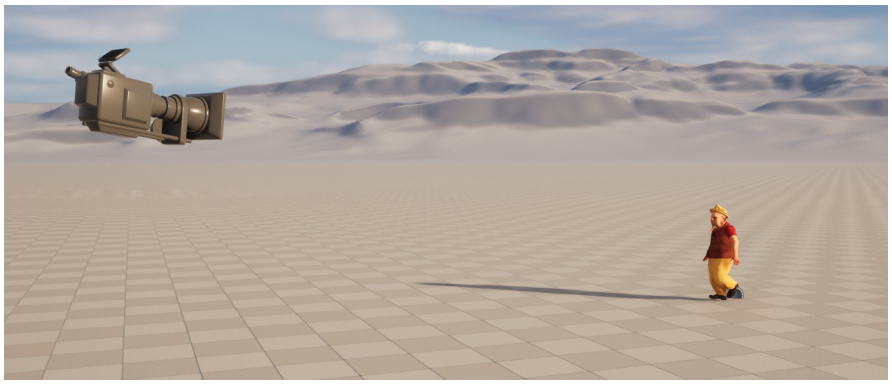

Figure 7: Scene structure

Figure 8: Skeleton projection

A.4 (Task extra)

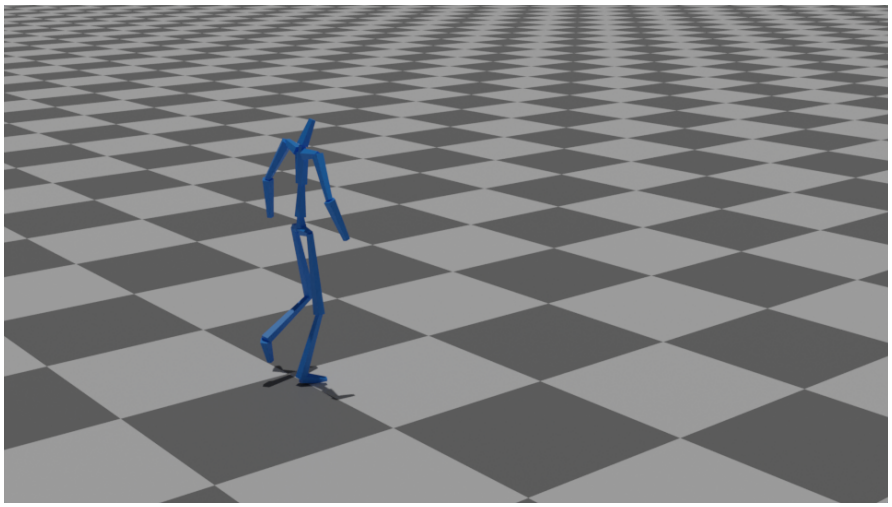

Figure 9: BVH import via deep-motion-editing