Computer Science Faculty, Technion

LiveCapCover: Report

Writers:
Shai Guendelman
Izo Sakallah
Advisor: Oshri Halimi
GitHub: https://github.com/izosak/LiveCapCover

Introduction

Our work is mostly based on the LiveCap paper.

Problem Description

In this work, we consider the problem of Live Motion Capture**,** i.e. reconstruct the motion of an object given some sensory input, in real time.

In this work the object of interest is a person, and the sensor is a single color camera. We can split the problem into 2 parts:

1. Firstly we need to acquire a parameterized model of the person, with some of the parameters are constant throughout the session, those correspond to the shape of the person, and other parameters correspond to the movement, and change over time to fit the observed motion. The parameters that change over time will be called DOF's, shorthand for Degrees Of Freedom.
2. Secondly we need to find a way to automatically estimate the DOF's that best fit the input we receive from the sensors, here the monocular images.

Applications

This problem appears in numerous applications, for example:

1. Video games - instead of interacting with the game through some external controllers, players could interact using their own movement. This might yield much more immersive experience from their side. For example we can look at Microsoft's Kinect, which uses a depth sensor for motion capture.
2. 3d telepresence - now more than ever, people interact remotely through video conferencing and other forms of communications. The ability to recreate the same movement in 3d in some far away room with only a single camera required can greatly increase the richness of the interactions.
3. Correct motion capture, if done accurately enough, might be used by physiotherapists and personal trainers to identify good or bad movements. It might be even done automatically, thus improving the quality of physiotherapy and exercise when a person is performing it by themselves.

Related work

Parametric Human modeling - In the original paper the model of the person tracked has been reconstructed from images, and then manually rigged. Different approaches exist depending on the usage of the model. SMPL is one of them. In SMPL the decompose a large number of 3d scanned models to a small basis that spans their possible 3d shapes, and in those models there

are joints that their locations were learned from the 3d scans. They reduce the problem of creating the 3d model to finding a set of about 100 parameters.

Real Time 3d Pose Estimation - In the original paper, VNECT, is used for estimating the movement from the input images. VNECT uses a CNN regressor to estimate the 3d position of each joint in the target person, and the results from the regressor are then used to initialize a second optimization step that corrects the estimated pose to align to some skeleton model. XNECT is a paper that extends VNECT, to apply its results from a single human, to multiple humans. VIBE is also a method that combines pose and shape estimation from monocular images, based on SMPL body parameters, it estimates those and uses a Generative Adversarial Network (GAN) to train it's model, estimating both the shape and the pose at one go.

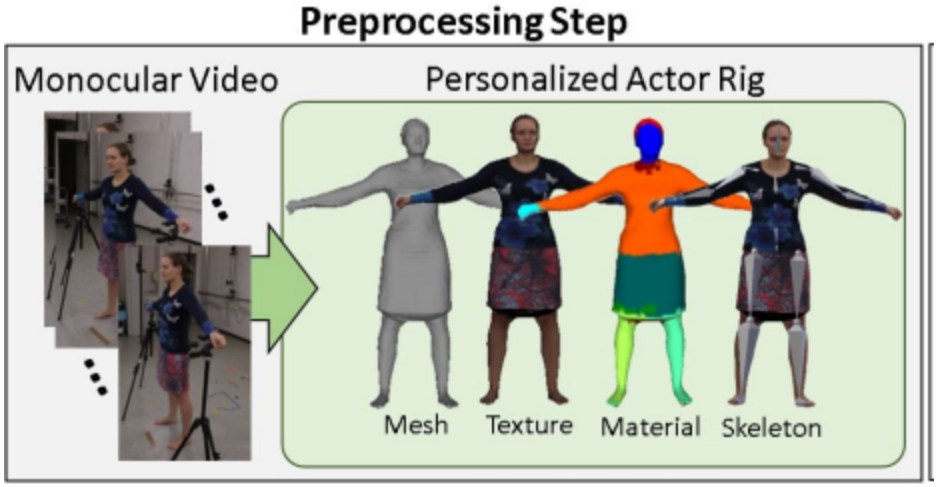
Approach

We will first start with explaining the main parts of the approach of the original work, then we will state the differences that we have in our implementation.

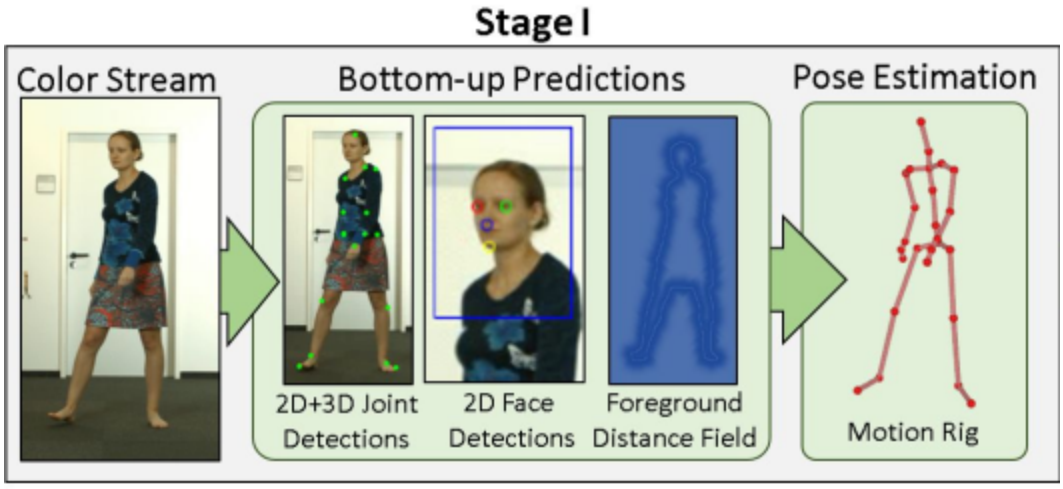
Original Work

The process can be split into 2 phases:

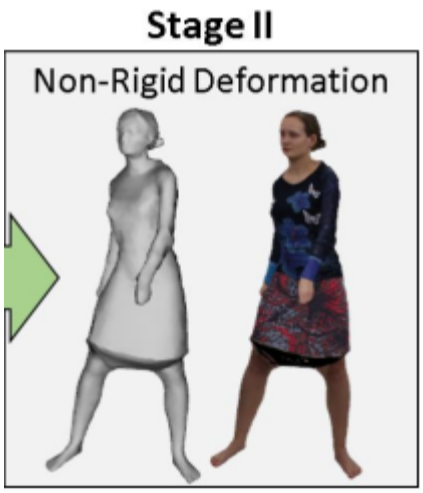
- Phase 1: offline preprocessing and model acquisition
    - 3d reconstruction from monocular images, for acquiring the model's geometry. In this part images of the person are taken from different angles, then an initial mesh is created. This mesh is downsampled to speed up calculations in the next phase. The tools used for this part are Agisoft Metashape, and MeshLab.
    - Model segmentation into different body parts - the images used for the reconstruction are segmented into their different body parts using the model described in Look into Person, we found a usable implementation here. Then the segmented images are used to assign each model vertex with a label corresponding to the body part it belongs to. Later on this label will affect the cost of moving this part.
    - Rigging a skeleton to the model - this is done using Blender. First the skeleton is aligned to the model, then an automatic algorithm sets the weights for each vertex based on the proximity to the joint.



- Phase 2: online motion capture. This phase is about estimating the transformation of the model from initial state to the state that best fits the monocular images and our prior knowledge(i.e., what we know about real human movement). It is done for each frame. Each stage is formulated as a Non Linear Least Squares (NLLS) problem, where we minimize the L2 norm of a vector function of the target parameters. The method used to approximate the solution to the NLLS problem is Gauss-Newton(GN). The cost functions used in each stage are:
    - Stage 1 - rigid pose estimation: the optimization parameters in this stage are the model's rigid 3d transformation (6 parameters) and the angles for each joint(~ 30 parameters). The cost is made out of 5 parts:
        - 3d - difference between the model joint's positions and the joint positions as predicted by the pose-estimation module for the current image.
        - 2d - difference between the model's projected joints, and the 2d prediction from the pose-estimation module.
        - Silhouette - we calculate the silhouette of the person in the image using background subtraction, then we calculate the image distance transform (IDT), then we sample a set of our model's contour vertices and take the IDT values on the contour vertices projected pixels.
        - Anatomic - for each joint angle we have a soft maximum and minimum values. If the joint passes those values we add the difference from the boundary to the cost.
        - Temporal - difference between the location in the previous frame of each joint, this is due to the small time interval between 2 frames and we know that the joint locations should not be too different.



- Stage 2 - non rigid vertex deformation: the optimization parameters in this stage is all of the models vertices, ~5000. The large number of parameters here is efficiently optimized using a sparse solver, due to the fact that each vertex affects only a small part of the cost. This stage's purpose is to capture deformations that are not directly caused by joint movement, like clothes wrinkling. The cost now is composed of:
  - **Photo -** project each vertex to the image plane, and compare the color of the image after applying gaussian blurring to it. If the difference between the color of the image and the color of the vertex passes some threshold then add this difference to the cost.
  - **Silhouette -** very similar to the silhouette loss in the first step, using the same set of contour vertices, but when a vertex crosses a region that is covered by a different body part with higher priority(this is discovered by using a body part mask) we just ignore it.
  - **Smooth -** compares the displacement of each vertex to its neighbors, as **vectors**, and compares them to the displacement after the pose estimation step. The semantic label of the vertex determines how high the cost for moving the vertex is, the same is true for the **Edge** cost.
  - **Edge -** compares only sizes of the vectors above, i.e. how much the distances from a point to its neighbors had changed from after the skinning stage.
  - **Velocity -** compares the translation of the vertex from its position in the previous frame.
  - **Acceleration -** compares the change in velocity that we observed in the last frame and the current frame.



Difference in our implementation:

- We did not recreate the entire paper, but mainly focused on the pose estimation part.
- We used python. The paper does not specify it, but we estimate that the authors used a combination of c++ and matlab for their implementation.
- We used VIBE instead of VNECT for 3d pose estimation. This resulted in some mismatch in the 3d joints extracted from the image and the model's 3d joints. Still it worked pretty well.
- We did not get to real-time results, due to our use of python and not running on a GPU.

Planning & Design

Project Milestones

When we started working on the project, we set some general milestones that we wanted to achieve:

1. Prepare, research
    a. Read the LiveCap article thoroughly, and list all the components needed in order to implement it.
    b. Experience and use the different libraries, packages and algorithms that are needed in order to implement LiveCap's different components.
2. Implement
    a. Implement LiveCap's different components and test each one of them as independently as possible.
    b. Put all components together to form LiveCap and test it.
3. Improve performance
    Identify bottlenecks in the project and try to reduce them.

We knew that we weren't necessarily going to exactly follow these milestones, but they surely helped us to focus and organize our work.

Programming language: Why did we choose Python over C++?
When we first started working on our project, we were wondering which programming language we wanted to use. We had two PLs in mind - C++, because it is the PL that we knew and studied best from our degree so far; and Python, because it is a PL with increasing popularity amongst the programming community which is known to be easier and faster to develop with.

What are our main requirements from the PL we use in this project?

- Code should run fast.
  Our purpose is to implement a real-time product, and that's why it's important for us to have code which runs fast.
- Code should be easy for development.
  As this is a project in the academy, it is more research inclined and is not meant to be released as a product in the industry. Therefore, we knew we would benefit more from an agile development process.

We organized our thoughts in a table:

C++ vs. Python	Pros	Cons
C++	● Code runs faster ● VNect is implemented in C++	● Harder to develop with
Python	● Easier to develop with ● Adding any ML algorithm / DL network would be easier for integration, in case we wanted	● Code runs slower ● Some packages might not be available / implemented well ● Packages don't always have good documentation

At first, while experimenting and trying to run and use different components, we began with C++, but then finally we decided we prefer an agile development process so we proceeded with Python.

Theoretical and Technical Background

Following our very first milestone, we started reading the LiveCap article. As mentioned in the introduction, we extracted the following main components needed to implement it:

- Facial recognition
- Background subtraction
- Rendering of a 3D human model
- Rendering an animation (Linear Blend Skinning)
- Joints detection in 3D and 2D

Then, following our next milestone, we started experimenting with all the different libraries and packages.

We went over all components that we knew we needed and we tried to implement and use each one independently, in order to make sure we knew how to use all our components so that we could finally put everything together.

Image processing

Here, we mainly used the OpenCV open source library. We tried using background subtraction methods available in the library but noticed that they weren't good enough for our use because they weren't precise enough. We also tried applying OpenCV's distance transform technique on some binary images, which turned out to work perfectly. We also tried using camera calibration and point projection.

Rendering

A crucial component in the project is the rendering of our 3D model and of an animation.

Linear Blend Skinning (LBS)

LBS is a very common algorithm used to deform a 3D Model. It allows animating the model. This algorithm is based on a skeletal structure of the model: in addition to the mesh the model has a skeleton (which is basically a tree of joints), and each vertex in the mesh has a list of joints - usually 3 or 4 - that affect this vertex. This list also includes the weights of these joints which represent how much each joint affects this particular vertex.

At first, we experimented with OpenGL to render a simple model and an animation of it, using PyOpenGL and the OpenGL Shading Language (GLSL). We managed to implement this by following an excellent Java tutorial series by ThinMatrix that shows how to:

1. read a 3D model mesh and skeleton data from a collada file,
2. implement an OpenGL render engine that uses GLSL to:
    a. render a 3D model,
    b. apply LBS algorithm.

We also followed a few other OpenGL tutorials that helped us with using OpenGL in Python (PyOpenGL), as ThinMatrix tutorials are in Java.

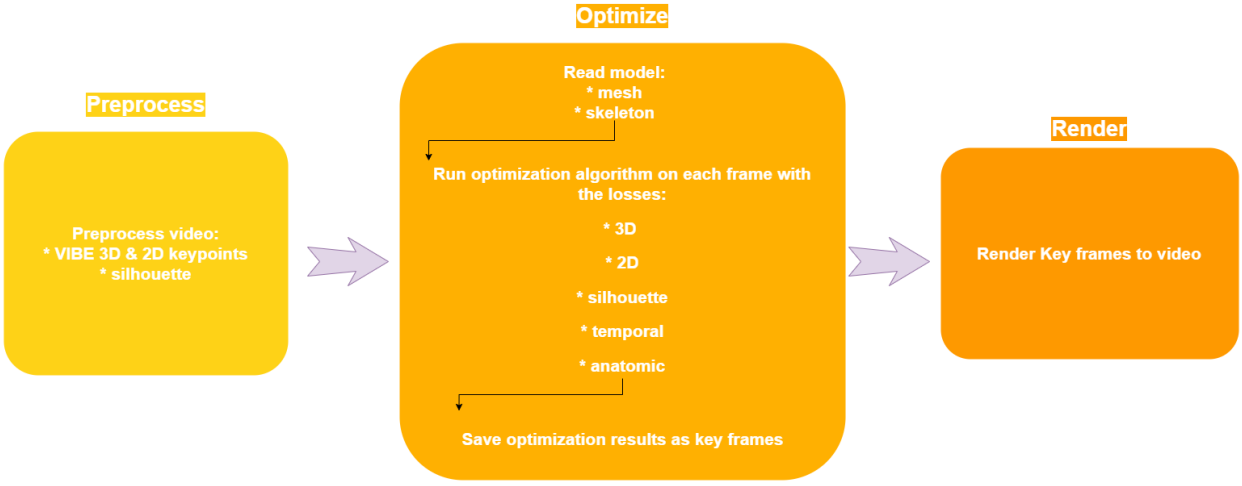
The ThinMatrix tutorials focused on a simple 3D model called “farm boy” and a simple animation of this model (both the model and its animation are saved in a single collada file). We managed to implement an OpenGL render engine that renders this farm boy model and its animation, and the rendering worked quite well. Unfortunately, we later found out that this engine didn’t match our needs, for two main reasons:

1. First, the OpenGL engine worked well only with models whose vertices have up to 4 joints affecting them. When we started working with a different human model that had 6-9 joints affecting each vertex we noticed that the animation of the model didn’t work well, because our engine cuts off extra joints and counts only the top 3 joints with the largest weights, and supporting more joints in GLSL felt counterintuitive.
2. As the LBS algorithm was implemented in GLSL, we didn’t manage to find out how to access the coordinates of the vertices during the animation, which is something we needed, for example, for the optimization of facial coordinates.

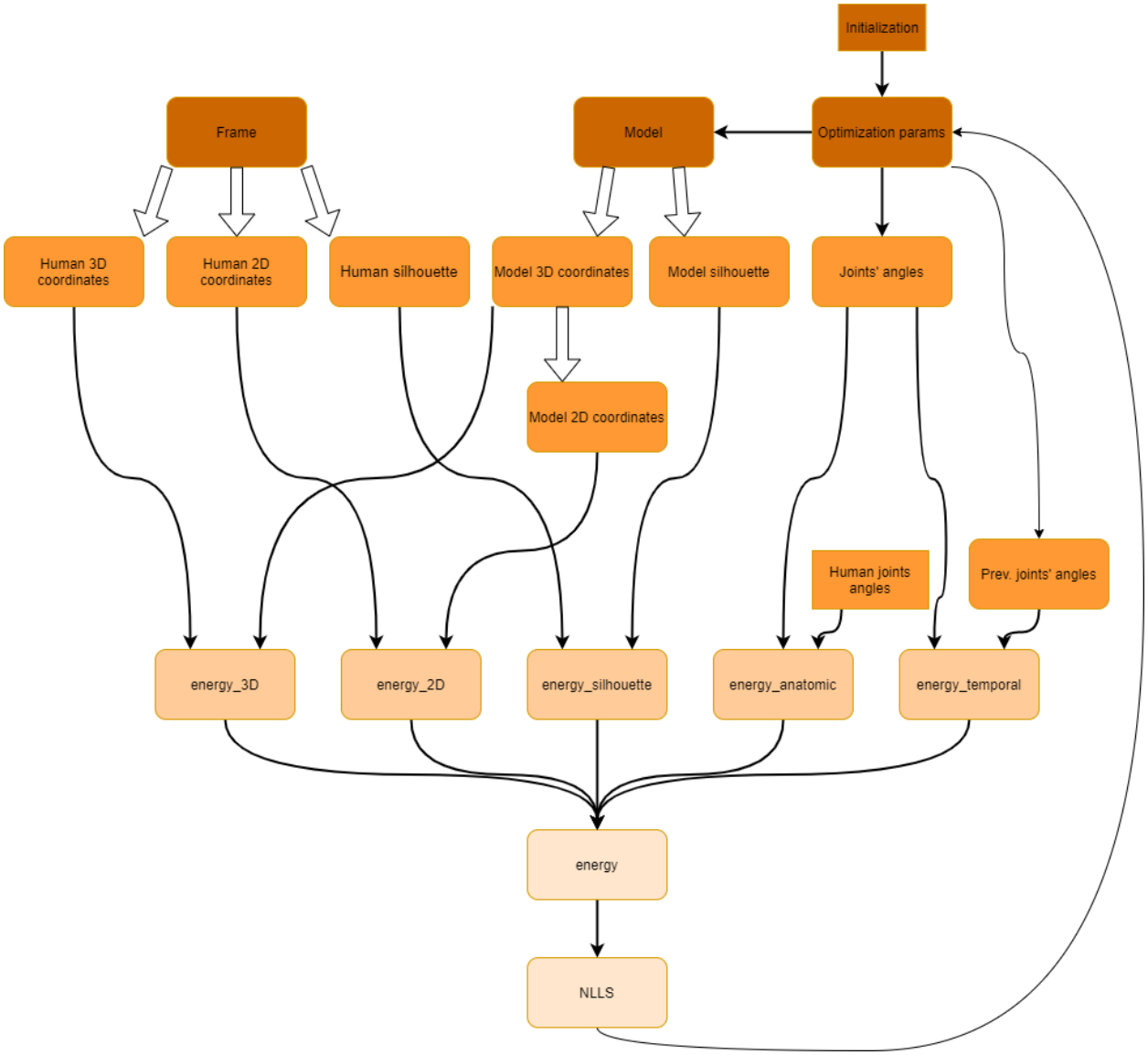
Eventually, due to the disadvantages of OpenGL and the fact that we implemented non-realtime, we decided to implement a new render engine using only Python and a 3D plotting package named PyVista. We used numpy.einsum to implement the LBS part (which was implemented in the GLSL code in OpenGL) and used PyVista for rendering.

Flow

Here are two diagrams that show the flow of our logic. First diagram depicts the general flow our our input:



This second diagram depicts the way we defined our minimization problem and implemented it:



Challenges and difficulties

While working on our project, we encountered natural challenges and difficulties.

First, the project required using many algorithms, libraries and code from different sources and projects that we weren't familiar with. We had to learn, catch up and complete all the necessary knowledge that we didn't have - from image processing, to model rendering and minimization problems.

We also had a few technical challenges. For instance, dealing with many different coordinate systems used to be confusing, such as in Blender and VIBE. To overcome this challenge we translated all input to the same coordinate system to make it easier for us to follow what we do. Implementing LBS, where we had different coordinate systems for the joint and its parent, also required attention to delicate ideas.

Background subtraction is also something that we spent quite some time on, trying to find the

best algorithm and conditions that should give a satisfying result.
In general, like in any other project, knowing when and how to advance in your work is critical and can be challenging. The ability to balance between digging into something and moving on with it was a great factor in our progress.

Experiments & Results

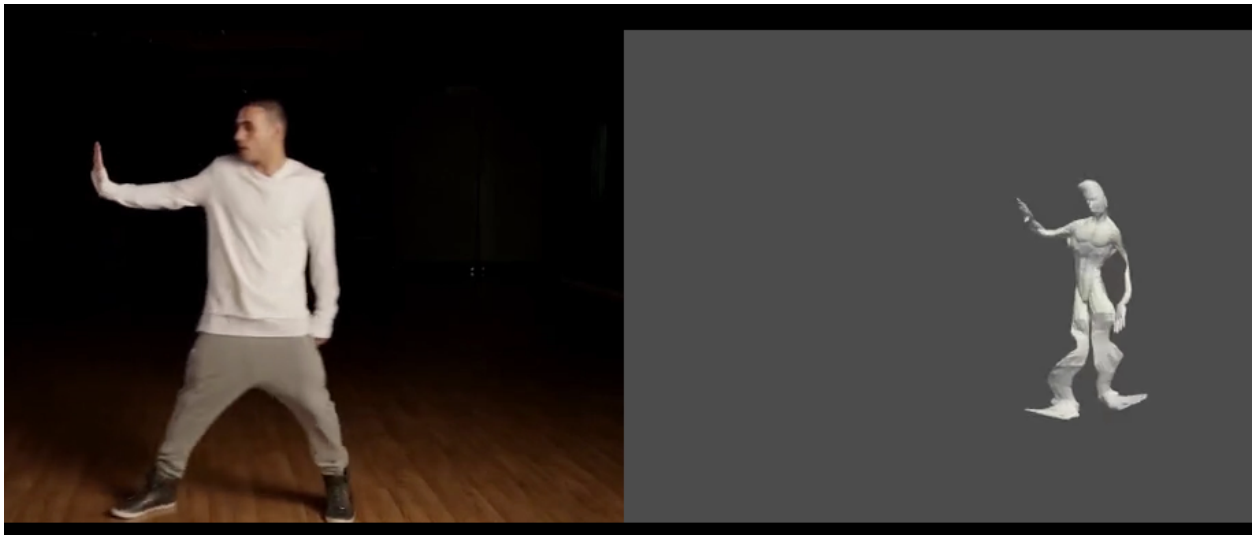
After getting all the system up and running we performed several experiments:

Experiment 1 - dance video + generic model

In this experiment we used a video that we have downloaded from youtube,
And a 3d model that was downloaded from the internet. We rigged the 3d model using blender, and used those for developing and testing the system. Even that something was a little off with the skinning weights of the model, and some other elements were missing, we still got some interesting results.

Results

- We can observe that the motion estimated by our program looks familiar to the input image, though that is far from perfect
- We can clearly see that the blending weights are not anatomically correct, due to the strange deformation of the model.
- Because the number of pose parameters, 3 for each joint instead of 1 as in the original paper, and due to the larger face and vertex count of our model of about 19,000 vertices, the time for each iteration took much longer then expected.
- The large number of pose parameters combined with the fact that we did not have available the joint limits values lead to some of the estimated positions did not seem natural at all.



Experiment 2 - original video and model

After trying to create our own models without success, we contacted one of the original writers of the paper that shared with us one of the videos and models used in their work.
We downloaded the rigged model named ‘mohamad’ with it’s skinning information, the first 2000 frames of the video, and the camera calibration parameters from the site, and ran the system on them.

Results

- The results look much more satisfying than the results with the previous experiment.
- The image appears a little jittery, this might be caused by large variation in the optimization parameters. Might be solved by increasing the temporal cost weight or by adding some other cost.
- The mismatch between the model’s joints and the prediction by the pose estimation module is apparent, for example the difference in chin location leads to the model to look down the entire video and because of the larger distance in the hips the pose estimated the legs are closer together then in the original video.
- We can see in some images that the model is lacking the ability to move in some way.

Experiment 3 - dropping parts from the cost and different weights

We wanted to check how changing the balance between the different parts of the cost will affect the results. So we experimented with changing their weights, and with dropping (setting their weight to 0) some of the parts. We did a total of 5 experiments + the baseline(experiment 3):

Experiment / Weight	λ_{3d}	λ_{2d}	$\lambda_{silhouette}$	$\lambda_{temporal}$	$\lambda_{anatomic}$
0. Baseline	1	1e-3	1e-3	0.1	0.5
1. No silhouette	1	1e-3	0	0.1	0.5
2. High anatomic + temporal	1	1e-3	1e-3	1	2
3. No Anatomic + temporal	1	1e-3	1e-3	0	0
4 No 3d	0	1e-3	1e-3	0.1	0.5
5. No 2d + silhouette	1	0	0	0.1	0.5

Reasons behind the experiments:

1. Dropping the silhouette energy - this part was a little bit problematic on implementing, and we wandered how much it affects our results?
2. Using higher weights for the anatomic and temporal energies - we saw that the results from the first part seems to be kind of jittery, and we wondered if increasing the temporal and anatomic costs might improve on those?

3. On the opposite side, we wondered what will happen if we drop the anatomic and

temporal costs?
4. (& 5) Now, we asked ourselves, is both the 2d and the 3d points necessary, or could we just use one of them? So we did 2 experiments - one without 2d loss and silhouette, and the second without the 3d loss.

Results

- We did not observe significant reduction in the results when removing the 2d and silhouette terms.
- Increasing the regulation (temporal and anatomical costs) did not significantly affect the results, but, completely removing them significantly reduced the quality of the results.
- Dropping the 3d loss leads to significant reduction in the accuracy. Complete movements like 360 rotation, were completely missed. But the results became less jittery, that led us to believe that it might be caused by the 3d points.
- Unnatural movement when dropping anatomic.

Conclusions

- The main difficulty in the solution is acquiring the model. If further work can be done on making this stage simpler and easier, it might make it easier to research this problem.
- The cost is composed of many different parts. Compressing the constraints might reduce the complexity of the method, and might make it much simpler. From the experiments we performed, it seems that the 3d pose estimation part is the most effective, thus it is worth investing most of the efforts there.
- Calibration between the model and the prediction model is very important. Might calibrate it with some automatic method?
- It seems that the skeleton is created based on common sense, and not based on biology. Might we create a biologically correct skeleton that will lead to more realistic results?

- The importance of making the work publicly available - a lot of small details are hidden from the paper. Without the official code, it gets much harder to track down their answers and to reproduce the same results.

Future Work

Symbiosis between the pose estimation, rigging, and skeleton optimization modules. Because calibration between the 3 systems is required, and to do it manually is relatively hard and boring work, might we find a way to tune it automatically, or better, create a relation such that one improves the other?

For example using the 3d pose estimation to fit a skeleton and weights to the 3d model, then running the given optimization, and feed the resulting joint positions to fine tune the model to this specific person?

Bibliography

Useful Links

Homogeneous coordinates:

- Math for Game Programmers: Understanding Homogeneous Coordinates
- Homogeneous Coordinates
- Projective geometry and homogeneous coordinates | WildTrig: Intro to Rational Trigonometry

OpenGL:

Youtube tutorials:

- OpenGL 3D Game Tutorials #1-10 (Java), OpenGL Skeletal Animation Tutorials (Java)
- OpenGL (Python)
- OpenGL (C++)

GitHub repositories:

- TheThinMatrix/OpenGL-Animation: A simple example of skeletal animation using OpenGL (and LWJGL).
- totex/Learn-OpenGL-in-python: All the source codes from my youtube tutorial series called "OpenGL in python".

Works Cited

Bogo, Federica, et al. “Keep It SMPL: Automatic Estimation of 3D Human Pose and Shape from a Single Image.” *Computer Vision – ECCV 2016 Lecture Notes in Computer Science*, 2016, pp.

561–578., doi:10.1007/978-3-319-46454-1_34.

“Discover Intelligent Photogrammetry with Metashape.” *Agisoft Metashape*, www.agisoft.com/.

Engineering-Course. “Engineering-Course/LIP_JPPNet.” *GitHub*, github.com/Engineering-Course/LIP_JPPNet.

Foundation, Blender. “Home of the Blender Project - Free and Open 3D Creation Software.” *Blender.org*, www.blender.org/.

Habermann, Marc, et al. “LiveCap.” *ACM Transactions on Graphics*, vol. 38, no. 2, 2019, pp. 1–17., doi:10.1145/3311970.

Kocabas, Muhammed, et al. “VIBE: Video Inference for Human Body Pose and Shape Estimation.” *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, doi:10.1109/cvpr42600.2020.00530.

Liang, Xiaodan, et al. “Look into Person: Joint Body Parsing & Pose Estimation Network and a New Benchmark.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 41, no. 4, 2019, pp. 871–885., doi:10.1109/tpami.2018.2820063.

Mehta, Dushyant, et al. “VNect.” *ACM Transactions on Graphics*, vol. 36, no. 4, 2017, pp. 1–14., doi:10.1145/3072959.3073596.

Mehta, Dushyant, et al. “XNect.” *ACM Transactions on Graphics*, vol. 39, no. 4, 2020, doi:10.1145/3386569.3392410.

MeshLab, www.meshlab.net/.

Webb, Jarrett, and James Ashley. “Beginning Kinect Programming with the Microsoft Kinect SDK.” 2012, doi:10.1007/978-1-4302-4105-8.

Zivkovic, Z. “Improved Adaptive Gaussian Mixture Model for Background Subtraction.”

Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004., 2004, doi:10.1109/icpr.2004.1333992.