KING'S College LONDON

University of London

# 6CCS3PRJ Final Year Individual Project: Capturing the golden moment in judo

Final Project Report

Author: Munkhtulga Battogtokh

Supervisor: Dr Rita Borgo

Student ID: 1603586

Programme: BSc (Hons) Computer Science with a Year in Industry

April 10, 2020

**Abstract**

This project aims to construct 3D models from images containing two judo fighters, by exploring alternative methodologies existing in (or building from) state-of-the-art 3D scanning technologies. Resulting 3D models relive the essence of the ”golden moment” captured in its input photo: the execution of the judo technique!

Since there are many 3D scenes corresponding to a given 2D image, a general 3D-reconstruction technique has trouble modelling a complex scene containing two human fighters in close intertangled contact. In this project, a more tailored approach is used through a tool that expects human bodies in its input images, and outputs only human body models. Furthermore, a novel methodology developed in this project optimises the pose of a 3D model against a target image. These combined with various pre- and post-processing successfully capture the golden moments of judo in a single 3D model!

**Originality Avowal**

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Munkhtulga Battogtokh

April 10, 2020

**Acknowledgements**

I thank my supervisor Dr Rita Borgo for guiding me throughout the project and supporting my passion by allowing me to chase my own idea. I also thank Vassilis Choutas, Nima Ghorbani and the rest of the SMPLify-X team for personally assisting me in using their fantastic work.

# Contents

# Introduction

This project aims to construct 3D models from image(s) containing two judo fighters, by exploring alternative methodologies existing in (or building from) state-of-the-art 3D scanning technologies.

A resulting 3D model must display the essence of the "golden moment" captured in its input photo: the execution of the judo technique! Therefore, it is sufficient to construct a model that approximates the fighters' physiques in relative positions characterised by the given judo technique in a standard 3D file format (Wavefront OBJ format), without any attention to facial features, clothing, hair etc.

Since the problem of reconstructing 3D models from 2D images is ill-posed (with infinitely many 3D scenes corresponding to a single 2D image [20]), modelling just a single person from photos is already difficult.

A general 3D-scanning technology like photogrammetry (implemented in available tools such as SCANN3D on Android, or Meshroom on Windows) fails to produce any intelligible result even with more than 10 images because photogrammetry algorithms have too broad output space yet too much sensitivity to input parameters. This sensitivity and unpredictable output space greatly reduces usability. By inference, given that photogrammetry has relatively low requirements among 3D-scanning technologies (requiring less specialist equipment), the usability of 3D scanning technologies is generally poor.

More specialised and focused 3D-scanning technologies construct only human body models in 3D from one or more photos. This project uses one such software called SMPLify-x. Although the output space being limited this way may sound like a disadvantage, this "limitation" is in fact a major **advantage** within the context of this project. The output space of SMPLify-x is already constrained to human bodies and saves the arduous effort.

However, even SMPLify-x cannot model two humans in close intertwining contact to a satisfactory quality level without additional methodology. In the following sections, we shall explore a way to break the input down to be as digestible as possible for SMPLify-x, explore a way to put the digested bits back together and discover novel approaches to further optimising an approximate solution.

# Background

## 2.1 Background and context

### 2.1.1 Context

The presentation of World Judo Championship 2019 by the International Judo Federation (IJF) awed spectators around the world with 360-view of the golden moments, in which new champions were crowned with beautiful techniques of the Gentle Way.

This was possible thanks to the multiple high-quality cameras watching the arena simultaneously. Consequently, I became motivated to create tangible 3D models of the forever-once moments capturing those truly unique executions of the World class judo techniques. No two executions of the same technique are *ever* the same—an execution is unique to each fighter, and each fight. Therefore, each execution is worth capturing for its uniqueness.

3D-scanning technologies (a class of 3D-reconstruction technologies, see 2.2, Review of relevant literature), photogrammetry in particular, were my first candidate technologies since they capture shape of an object from multiple views — as offered in the aforementioned 360-view by the IJF. Unfortunately, available photogrammetry software including SCANN3D on Android [34] or Meshroom on Windows [3] failed to produce any useful output from the stereo images.

As a human, I can create a cognitive "3D model" from a single image. Yet, despite their complexities and the fact that photogrammetry has relatively low requirements among 3D-scanning technologies (requiring less specialist hardware equipment), these photogrammetry software failed even with more than 10 images. The IJF invested in high quality and quantity of expensive equipment to create these images. It is disappointing that even then the requirements by a relatively usable 3D-scanning technology were not satisfied.

This gap between computers and humans has identified an opportunity to raise the usability of 3D-scanning technologies (at least within the scope of modelling judo scenes) by lowering the input requirement to ideally just a single image. Keeping in mind how humans can produce cognitive 3D models from single image, we explore a fresh domain-specific approach to 3D scanning in later sections.

**Judo terminology** Since the domain of this project is judo, it is important to know the essential judo terminology. This report uses the proper judo terms to describe judo scenes as opposed to using English translations. It will be sufficient to know the following two terms for the majority of the report:

- Tori - attacker or the person executing the technique
- Uke - victim or the person being subjected to the technique

### 2.1.2 Motivation

The 3D-scanning market is valued at:

> "USD 1.007 billion in 2018, and it is expected to reach USD 3.261 billion by the end of 2024"
>
> —Mordor Intelligence [16]

It is a market with promising growth (in comparison, the recently popular Blockchain industry is valued 1.878 billion in 2018 [30]). This project can contribute to the market's growth by:

1. Addressing the gap in usability of 3D-scanning technologies identified in the previous sub-section for a domain—judo, which allows further generalisation into other forms of martial and performance arts, or any multi-person scenario.

2. Advocating the combined use of diverse 3D-reconstruction approaches: domain-specific modelling, 3D-scanning, and inverse rendering (see 2.2, Review of relevant literature).

3. Directly introducing new communities to the technology and the technology to the communities: judokas, and potentially other martial artists, sports-people, dancers and so on.

### 2.1.3 Aim

The aim of this project is to do what current 3D-scanning technologies fails to do:

> Develop software/methodology that takes **input image(s)** (.jpeg, .png) of a judo scene with up to 2 people, and **outputs a 3D-model** (Wavefront .obj [24] or other) file capturing the fighter(s), by exploring alternative methodologies existing in (building from) state-of-the-art 3D reconstruction technologies.

## 2.2 Review of relevant literature

### 2.2.1 Minimal prior knowledge approach - 3D scanning

> "3D scanning is the process of capturing digital information about the shape of an object with equipment that uses a laser or light to measure the distance between the scanner and the object."
>
> —AbsoluteGeometries.com [15]

General 3D-scanning technologies like photogrammetry do not "expect to see" anything in particular. Therefore, they theoretically can model any 3D object. Advantages are general applicability, and theoretical ability to capture visual details like texture and the overall shape in a single step. Disadvantages are as explained in Introduction, too high requirements on input, relatively high learning curve spanning several computer vision principles just to optimise the output [4], and practical inability to produce desirable output due to high sensitivity to input quality/quantity.

**Software Evaluation** For example of photogrammetry software, meshroom [3] is an open-source software utilising traditional Computer Vision techniques in its pipeline: Feature Extraction, Image Matching, Structure from Motion and Depths Map Estimation (Stereo Vision), Meshing, Texturing and finally Localization [4]. These are steps at high level of abstraction, and each of them are complex and uncertain to produce desired results. When these steps are chained, the uncertainty accumulates like in a Chinese-whispers game, which may explain the complete failure despite "reasonable" input. However, for images satisfying the preferred conditions (appropriate lighting, high-resolution camera, critical distance between viewpoints, and camera parameters), meshroom produces results with reasonably accurate shape and good visual details (texture and color) for impressive range of objects: statues, toys, trees, etc. [2]

### 2.2.2 Maximal prior knowledge approach - modelling human body

Humans can efficiently interpret visual input thanks to extensive prior knowledge of real-life objects built through visual scanning and touch. Considering this and another insight that just the relative poses of two fighters are sufficient to capture the essence of a judo scene, a feasible solution should ideally possess the following two properties: prior knowledge, and abstract pose-modelling. Thankfully, *Expressive Body Capture: 3D Hands, Face, and Body from a Single Image* (Pavlakos et al. 2019) [28] has combined the two insights to create SMPLify-x, which can 3D-model a single person from a single image once at a time.

Advantages of this approach are minimal requirement on input (only 1 image), relative insensitivity to input quality, and relatively low learning curve to optimise output thanks to few well-encapsulated stages (see their Github [29], Dependencies).

Disadvantages of the approach are the general need to incorporate prior knowledge (not relevant in this case because it has already been done by Pavlakos et al. 2019), limitation of input/output space to only human bodies, limitation of application to only one person at a time and the inability to capture visual details like texture, hair, and clothes.

**Software Evaluation** Pavlakos et al., 2019 has developed the method SMPLify-x, under "Software Copyright License for non-commercial scientific research purposes" at [28]. It combines VPoser (human pose model prior), OpenPose (2D key points of joint locations), and SMPL-X (unified body model with shape parameters).

This relatively short pipeline and low requirement on input produce more reliable results for cheaper cost. However, simplify-x can only be applied to one person at a time and therefore, a methodology to "imagine" each fighter one-by-one by separating them to sub-images is additionally required for judo scenes.

### 2.2.3 Optimising approximate body models - Differential Rendering

An approximate body model refers to some 3D body model approximately capturing the pose and shape of its target person. For example, models constructed with SMPLify-x are approximate because not only are the poses of these models not fully accurate, but also there are other details SMPLify-x ignores: such as clothing, exact facial features, hair etc.

Differential Rendering, its application to Inverse Rendering (problem of deducing intrinsic scene parameters like depth, camera pose, and lighting from an image) in particular, has the potential to optimise approximate 3D models against the target person given in a target image. Loubet et al. (2019) demonstrates its effectiveness in adding intricate shape and texture details to approximate 3D models of objects like starfish and pear fruit [21][22].

Gradient-based optimisation is a recent approach to solving inverse rendering problems. It requires calculating partial-derivatives of pixels with respect to the scene parameters [21], which is why not just any renderer but a Differential Renderer is required to do optimisation. Gradients calculated for a desired scene parameter are then used to modify the values of the parameters themselves in a gradient-descent fashion. This is how for instance the geometry of a 3D object, which obviously determines the pixel values of its rendered image, can be modified until the object looks more like in its target image.

The major advantage of such gradient-based approach exists thanks to the calculus chain-rule. In simple words, the approach allows changing any variable that we can "trace" (or back-propagate) the computation of the rendered image pixels to. We can trace the computation of the rendered 3D object to the object's geometry for instance with differentiation. Then, if we can further trace the geometry of the object to "deeper" variables that determine the geometry itself, we can change those variables too.

For example, the rendered image of a 3D human body model can be traced to the geometry (the positions of the vertices), and the geometry itself can be traced to the pose of the body. It is indeed possible to trace down to the pose because a Differential Renderer can differentiate a rendered image of an object with respect to the object's geometry, and VPoser (a component of SMPLify-x) in turn can differentiate the geometry with respect to its pose; VPoser is "end-to-end differentiable" when constructing the human body from a pose [27].

The major disadvantage of gradient-based approach and Differential Rendering is 2D-ambiguity. Loubet et al. (2019) [21] uses Differential Rendering to optimise an approximate model rather than to construct a model from scratch. The initial approximate model serves as an intention to constrain the output space to only objects similar to it. This is because gradient-based optimisation seeks local minima, but not necessarily the global minima.

**Software Evaluation** Loubet et al. (2019) [21] demonstrates the effectiveness of their newly proposed technique for differentiating path-traced images with respect to scene parameters [22]. They compare their work against that of Li et al. (2018), *Differentiable Monte Carlo Ray Tracing through Edge Sampling* [19], and find that each have their own strengths. As explained in Loubet et al. (2019)[21], Li et al. (2018) performs better for simple scenes.

We use *redner* (MIT License) by Li et al. (2018) in this project simply due to public-availability (Loubet et al. 2019 had not published their code as of February 2020). As a Differential Renderer, *redner* can differentiate a rendered image with respect to geometry of the object it renders. This allows redner to have wide array of 3D-to-2D optimisation applications (e.g., 3D face fitting [20]). However, the optimisation requires an initial approximation and performs slower than SMPLify-x, which is accepted because this project prioritises the quality of result 3D-models highest.

# Design & Specification

## 3.1 Requirements

### 3.1.1 Scope

**Domain** Given the time constraint, it is wise to minimise tasks and leave outside of scope any functionalities that available tools already provide, e.g. converting .obj 3D format to .stl format, which can be done easily with an online tool.

Furthermore, while the spirit of creating 3D-models of pairs of people generalise to any combat sport like boxing, karate, Mixed Martial Arts, or any other performance-arts like dancing, we require any software resulting from this project to deal with **only** judo. Applications to domains outside judo are extra and not part of requirements.

**Detail of output** There are different extents to which the artifact 3D-model may be detailed. For example:

- Minimal: two fighters are in pose, have template face and physique, but no clothing, texture, or facial expression
- Medium: two fighters have some form of clothing not necessarily accurate to image
- High: two fighters are modelled with their outfits with motion effect accurate to image (e.g. the outfit waves and creases to create beautiful effects capturing the directions of movements).
- Ideal: two fighters are modelled with clothing with accurate effects, hair, their own facial features and facial expression when visible.

We require **only minimal level of detail**. Higher levels of detail are desired but **not required**.

**Software functionality** It is not of interest to automate integration tasks that are easily carried out manually. Such automation may not be trivial work and create more tasks than it saves.

For example, SMPLify-x (mentioned in sub-section 2.2) requires output of OpenPose: a keypoints-file. Manually running OpenPose first, and feeding its output as input to SMPLify-x is trivial for a human. However, automating this process is not trivial because the best way to validate whether the keypoints are on the right places is qualitative examination. For example, a keypoint can be a joint and the best way to tell if there is an actual joint on the source image

at that keypoint is to simply examine the keypoints-rendered version of the source photo that OpenPose conveniently provides.

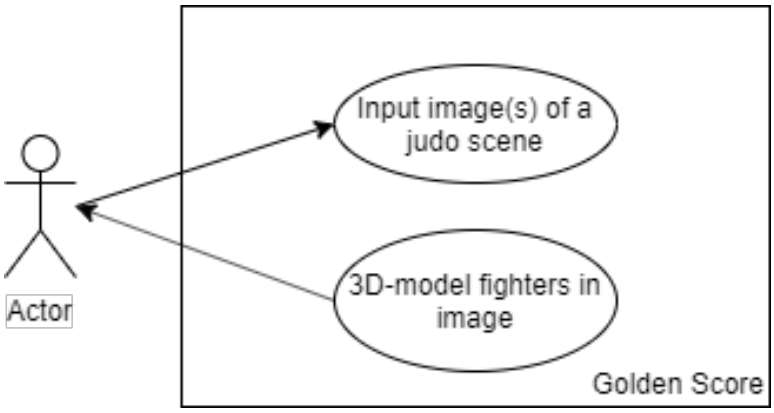
Therefore, *the software* is **not required** to complete the whole process in a plug-and-play style. A simple manual will accompany the software instructing how to convert the input image(s) to a 3D-model, where appropriate. This allows me to prioritise tasks that deliver more substantial value towards the goal.

### 3.1.2 Use case

Individual Project Guide to Deliverables [10] states "The final year project ... will require you to produce a substantial piece of written work, and a significant piece of software" [10].

*The software* of this project, which we will call Golden Score from here on, has use cases illustrated in figure 3.1.

Figure 3.1: User inputs image(s), Golden Score outputs 3D model




## 3.2 Specification

In this section, we define the specifications of "Golden Score", in terms of the input it must accept, the output it must produce, and the output quality - the degree to which the output should match the input.

### 3.2.1 The input – Domain Modelling

The input is one or more images. The fact that we are dealing with judo scenes allows us to narrow down the exact entities Golden Score must expect in its input. The entities are illustrated in the following class-diagram.

Figure 3.2: Class diagram modelling domain of Golden Score

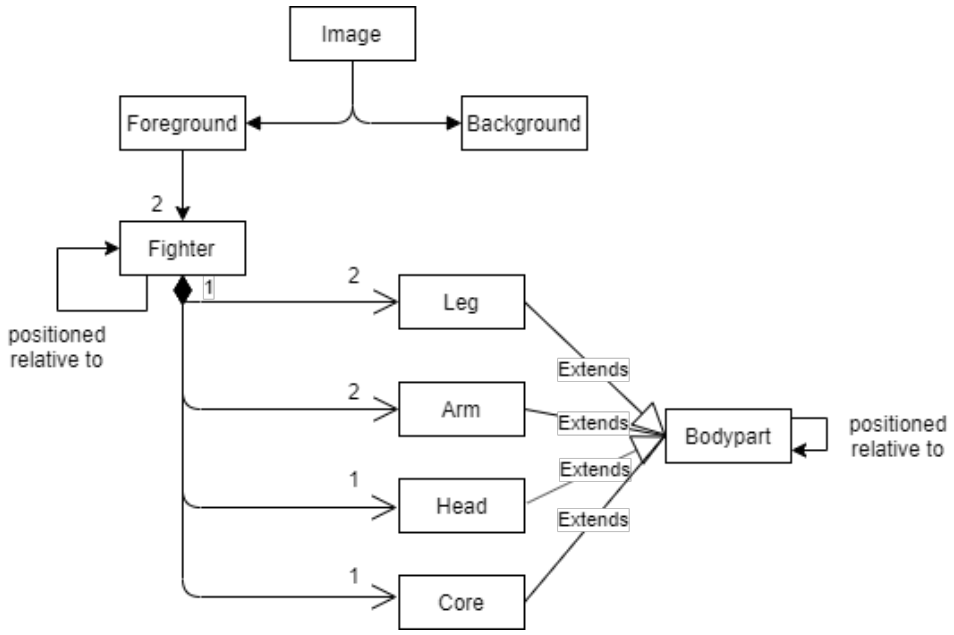



Figure 3.2 captures the below information more precisely:

- Image has 2 fighters in Foreground, and Background
- Fighter has body-parts: legs, arms, hands, head, and core
- Fighter has position relative to other fighter
- Body-part has position relative to other body parts

In cases where input is more than one image, each image must follow the above specifications. The multiple images must represent the same scene– the same technique from different angles.

### 3.2.2 The output – Expected functionality

Domain model in Figure 3.2 allows to formulate the specification in a concise manner:

- Background is ignored
- For each fighter, his/her body-parts are matching in input (source) and output.
- The relative position of the two fighters is matching in input and output.

See next sub-section for what is meant by ”matching” input and output,

### 3.2.3 Input vs. Output – analysing output

Determining the extent to which the source *matches* the output is primarily done through qualitative analysis. For example, for model of image in Figure 3 to *match* the source, model must fit the description:

> Head of blue is below that of white. Blue's both legs are airborne. White's both legs are at the lowest level in the scene.

This is satisfactory information to capture the essence of the scene - who is throwing and who is being thrown and rough idea of what the technique is. For example, the fact that white's both legs are on the ground and that his head is above blue's is enough to characterise the technique as tewaza (hand technique).

Figure 3.3: Example input image


A qualitative analysis may *only* supplement the degree of match analysis. The approach in this project is to:

1. Find source image's viewpoint on the 3D model
2. Remove any backgrounds
3. Reduce to 2D image
4. Compare foregrounds

**Optimising quantitatively** Coincidentally, the technique employed by *redner* to optimise 3D scenes against a target photo employs the above approach [19]. Pose-estimation, or manual parameter fine-tuning can find the camera position to render a given 3D model so that the viewpoint matches that of the target image. Reducing 3D model to 2D image is possible with the various rendering functionalities, e.g. path-tracing. Comparing foregrounds can be done in terms of calculating scalar loss as mean of pixel differences.

Figure 3.4: background removed

Figure 3.5: silhouette

This quantitative analysis is however **only supplementary** because comparing the overlap does not provide a reliable way to compare qualities of different outputs. For example, an output may have high rate of overlap, but significantly different pose and unnatural shape as illustrated in Figure 3.6, 3.7, and 3.8.

An example output in 3.7 has high rate of overlap with example source image in Figure 3.6, but not only does the output have an unnatural bending of the arm and feet, but also it fails to capture important characterisation of motion and the judo technique: left foot is highest up in the source—indicating a dramatic fall to the ground by a well-executed technique—but not in the output. In later chapters, results illustrate this flaw of the quantitative analysis approach and its subsequent implications.

Figure 3.6: source

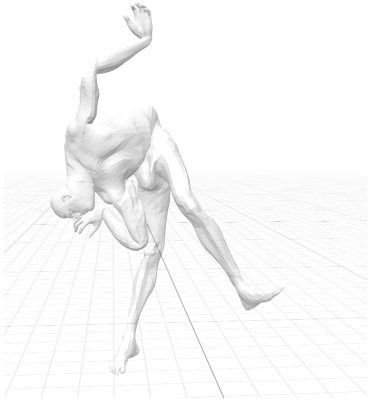


Figure 3.7: model

Figure 3.8: overlap

## 3.3 Design

In section **??**, the first two ”base approaches” are for solving the main challenge of constructing 3D models from 2D images and the last optimisation approach is for polishing the 3D model created from any of the base approaches. Due to time constraints, only one of the base approaches will be used, and we will detail how to incorporate the optimisation approach later in this chapter.

### 3.3.1 Evaluation of the alternative base approaches

In this sub-section, we evaluate the two alternative base approaches from section **??**:

1. ”minimal prior knowledge” approach with photogrammetry
2. ”maximal prior knowledge” approach with SMPLify-X

Then we make a design choice of whichever approach is best for the context of ”Golden Score” under the resource limitations of **time** and **input availability**.

Table below summarises the advantages and disadvantages of the approaches, in terms of the above resource constraints and the project aim.

| **Approach** | **Advantages** | **Disadvantages** |
|---|---|---|
| Minimal Prior Knowledge (Photogrammetry) | - More general applicability as 3D scanning method<br>- Theoretically, accurate visual details | - High requirements on inputs<br>- Higher learning curve (time requirement) |
| Maximal Prior Knowledge (SMPLify-X) | - Low requirements on input<br>- Lower learning curve<br>- Less general applicability as 3D scanning method* | - Low visual details (only high-level description of the scene)<br>- Less general applicability as 3D scanning method* |

### 3.3.2 Choice of base approach

The choice of base approach is Approach 2 of maximal prior knowledge simply because its advantages are tailored to the project's resource constraints of time, and requirement on input quality/quantity. One could argue that Approach 1 has no advantage for the scope of this project other than no theoretical need to ”separate” fighters into sub-images.

Approach 1 has the general advantage that it can be applied to broader range of inputs to produce broader range of outputs. While this is an advantage as a general 3D-scanning technology over Approach 2, which only models human bodies, it is in fact desirable that output-space of whichever methodology we choose be restricted to expected output-space set by the scope: human bodies.

Furthermore, while the theoretical advantage of Approach 1 that visual details like clothing, hair, and texture can be captured with photogrammetry without extra techniques, this advantage is indeed theoretical. In practice, Approach 1 has “no idea what it sees” and is too

sensitive to input. For stereo-vision that photogrammetry is based on, multiple high-resolution images under certain lighting are required, and it must be possible to pair them so that each visible point on the object is on at least 2 images. There are even requirements for hardware information like the physical distances of the cameras that took the photos, focal length, and distance to the objects etc.

Narrowing down the output-space as expected is an expensive task that Approach 2 has already completed. This is why the general limitation of Approach 2 that it only models human bodies is in fact an advantage for the context of this project.

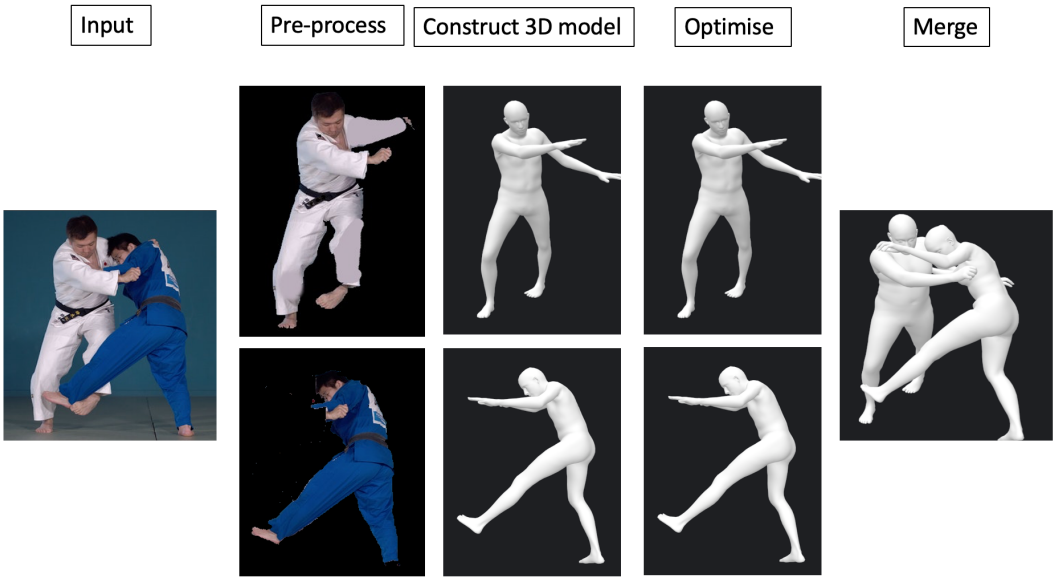
In conclusion, Golden Score **takes Approach 2** of "maximal prior knowledge" because Approach 1 has less practical advantage over it for the context of Golden Score under *time* and *input availability* constraints.

### 3.3.3 Final design

Finally, it's time to describe how to incorporate the chosen base approach and the optimisation approach into Golden Score's implementation. Following is a high-level description of how Golden Score is designed to achieve the properties in the specification:

1. Input image(s) following input specification (see 3.2.1).
2. Pre-process the image: separate the two fighters into separate sub-images.
3. Construct 3D model for each individual fighter from the sub-images (Use Approach 2)
4. Optimise each individual 3D model (Use optimisation approach).
5. Merge the individual 3D models into a single model.

Figure 3.9: Final-design workflow



# Methodology

## 4.1 General Implementation

In this section, we describe the concrete implementation of each step in the final design (see 3.3.3), except inputting image(s). For specification of the input image(s), please see 3.2.1.

### 4.1.1 Pre-process: Separate the two fighters into separate sub-images

#### Clear distractions

Since SMPLify-x produces 3D models for only one person at a time, and two judo fighters in close contact can make it difficult to tell which body part belongs to which person, the first step to making our input photos digestible for SMPLify-x is to reduce the image into two separate images each containing nothing but the visible parts of its respective fighter.

This can be done with many available tools (online or through traditional software). But since this project prefers minimal software that can potentially be automated, Grabcut [32] takes the task.

Grabcut is an algorithm designed for extracting the foreground from an image with minimal user interaction [26]. We use a slightly modified version of the python samples provided by OpenCV [1] under its Open Source Computer Vision Library (3-clause BSD License, see Chapter 7: Legal, Social, Ethical and Professional Issues).

#### Clear obstructions

A major challenge to constructing 3D models of humans in close contact scenarios is obstruction. This is the case where parts of either fighter's body are obstructed by parts of the other fighter. Trying to model both fighters in the image together may cause confusion as to which body part belongs to which individual. Therefore, clearing obstructions is an important step.

Initially, I considered the following research on clearing obstructions: A computational approach for obstruction-free photography (Xue et al. 2015), which removes obstructions and reflections from an image. However, the employed technique relies on multiple images that complement missing views in each other [36]. For example of such images, frames of a video where the camera is moving linearly captures the closer and further objects moving at different rates, and allows for distinguishing the closer and the further objects. More importantly, obstructed parts in one frame are in fact visible in the other ones.

Unfortunately, these complementary views are not available in judo images. The closest to such complementary views come from 360-views of throws during the 2019 Judo World Championships by IJF [11]. However, the motion of the camera there is circular: while an

obstructed body part in one frame is visible in another frame, the frames look at the object from significantly different angles. These frames do not complement the view of an object from a single viewpoint. Given the relative size of the obstructions, it may not be possible to clear the obstructions without "looking around" them at significant angles from the first place.

More basic yet effective approach is to simply remove pixels that are potential source of confusion, i.e. looking at each human body one by one. A subsequent problem is to then fill in for the missing pixels where the obstructions are removed. This too is solved by a simple manual edit where the "gaps" are filled in with the same colour as the edges of the discontinuities. See example application in Appendix **??**.

### 4.1.2 Construct an individual 3D model

Now, with two images each containing a single fighter, constructing the 3D models from them requires a 2-stage process.

1. Generate keypoints JSON file using OpenPose [7][8][33][35]. See instructions on their Github repository [6].

2. Generate 3D models in Wavefront OBJ format [24]. See general instructions on the Github repository of SMPLify-x [29].

### 4.1.3 Optimise an individual 3D model

Optimising the output by SMPLify-x (individual 3D body model) improves the overall quality of the finished 3D model (the two 3D models positioned together). In this project, two approaches were employed for such optimisations:

1. Improving the body pose priors

2. Differential Rendering

#### Improving body pose prior

VPoser is an implementation detail of SMPLify-x that provides prior distribution over valid human poses [28]. The default provided distribution is trained on the large AMASS datasets [23] of human motion capture.

As later will be discussed in Evaluation (6.2.3), poses of output 3D models are often too "neutral" when compared to the actual judo poses in the source photos. Often the poses simply cannot be modelled. This is likely because the pose prior distribution does not recognise the judo poses as "valid". This in turn is because the training datasets do not contain sufficient examples of judo poses.

Therefore, training VPoser on datasets that contain examples of judo poses would likely result in more accurate poses. One such dataset is PosePrior dataset of the AMASS, containing motion capture of acrobatic movements and dramatic kick motions [23]. Ideally, further training the already existing prior distribution on this dataset would hypothetically improve pose accuracy since the resulting distribution should recognise judo poses as more "normal". However, time and hardware constraints (see 6.2) only allowed for creating an alternative prior distribution that is trained on the PosePrior dataset starting from scratch: named vposer-judo.

**Differential Rendering to optimise close-approximation**

Rendering algorithms project 3D objects onto a 2D image. A differential rendering algorithm in particular allows differentiating the pixel values of its output image with respect to scene parameters – parameters of the rendering algorithm – such as camera position, lightning and geometry. Differential renderers have application in inverse rendering: the problem of reconstructing 3D models from 2D images, which encompasses the problem of this project.

*redner* by Li et al. (2018) is a differential renderer that offers many useful resources including python interface and tutorials. This can be used for inverse rendering (hence the name redner) through optimising loss calculated from comparing a rendering output image to a target image with gradient descent [19]. This is possible thanks to the fact that *redner* can calculate gradients for the rendering parameters: most relevantly the geometry of 3D object being rendered. Error calculated from comparing the target image to the render is back-propagated until gradients for the rendering parameters are calculated. Then, these gradients can for example be used in Stochastic Gradient Descent [31], or a similar optimisation mechanism to update the values of the rendering parameters.

In this project, Stochastic Gradient Descent [31] or Adam [18] updates either:

1. Rendering parameter directly: position of vertices in 3D mesh
2. Derivative of rendering parameter: body pose

Capitalising on the fact that VPoser is end-to-end differentiable [27] (which means that we can differentiate the 3D body model's vertices with respect to the body pose parameter), chain rule can be applied to optimise the rendering output with respect to the body pose.

### 4.1.4 Merge individual 3D models together

Define two functions:

1. Translate. Take a starting point in 3D space and three dimensional vector representing the translation. Return the resulting point in 3D space obtained from applying the translation vector to the input starting point.
2. Rotate. Take a starting point in 3D space, pivot point to rotate around, rotation angle and rotation axis. Return the resulting point in 3D space obtained from rotating the input point around the pivot point, for input rotation angle along the input rotation axis.

This is a minimal approach. Apply combination of the above functions interactively to all vertices of either fighter to translate or rotate them as whole. Although computation takes negligible time, note that multiple interactive iterations may be required in this trial-and-error process. Usually, only trivial transformations are sufficient to merge the models together on their relative positions so that they reproduce the scene in the target image qualitatively satisfactory.

## 4.2 Generating demonstration models

In previous section (4.1), general implementation of design steps are described. In this section, methodologies (consisting of those implemented steps) for creating 3 demonstration models are detailed. Since trial-and-error is required to generate each model and intermediate results constitute the methodology to reach the final models, these intermediate results are described within this section while the final results are later discussed in chapter Results (5). Note that the merging step is not detailed since description in sub-section 4.1.4 applies equivalently to all models and provides adequate information, while its results are discussed in Results (5) as aforementioned.

### 4.2.1 Modelling Osotogari (Battogtokh M., 2020)

Figure 4.1: Osotogari Tori - the attacker

Figure 4.2: Osotogari Uke - the victim

Osotogari means "major outer reap". Judo Info describes the technique as follows:

> "As you step close to the outside of your opponent's foot you push your opponent off balance to the rear corner while holding him/her close."
>
> —Neil Ohlenkamp, Judo Info [25]

The first model is generated from two source images (Figure 4.1 and 4.2), each offering a clearer view of each fighter. See the entire process description in terms of its intermediate results below:

**Pre-process.** Each source image (Figure 4.1 and 4.2) is reduced to the fighter it most clearly shows as described in 4.1.1 using Grabcut [32]. The intermediate results are displayed in Figures 4.3 and 4.4.

Figure 4.3: Osotogari Tori - isolated

Figure 4.4: Osotogari Uke - isolated

**Construct 3D model.** As in the two-step process described in 4.1.2:

1. Generate keypoints JSON files. The information in these files are rendered below:

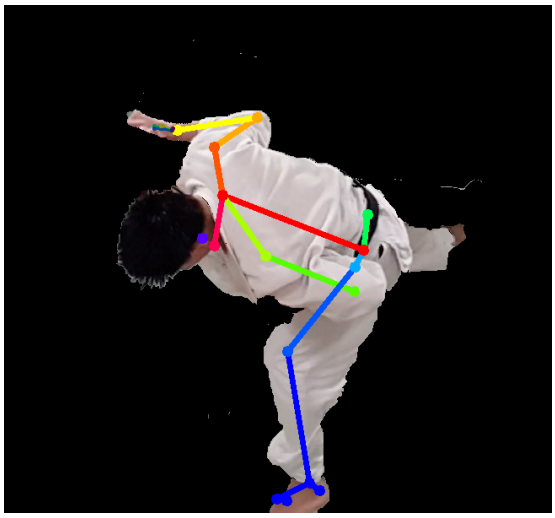


Figure 4.5: Osotogari Tori - keypoints rendered

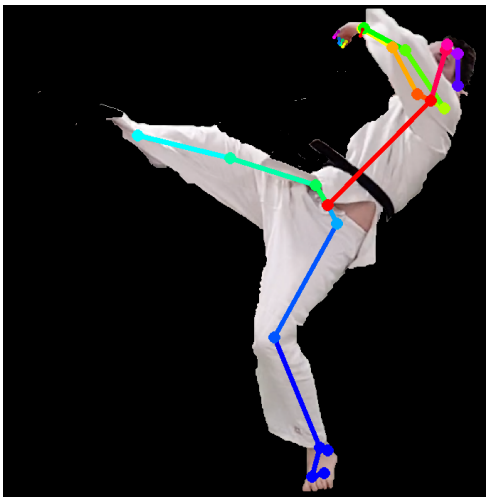


Figure 4.6: Osotogari Uke - keypoints rendered

2. Generate 3D models. See intermediate results below:

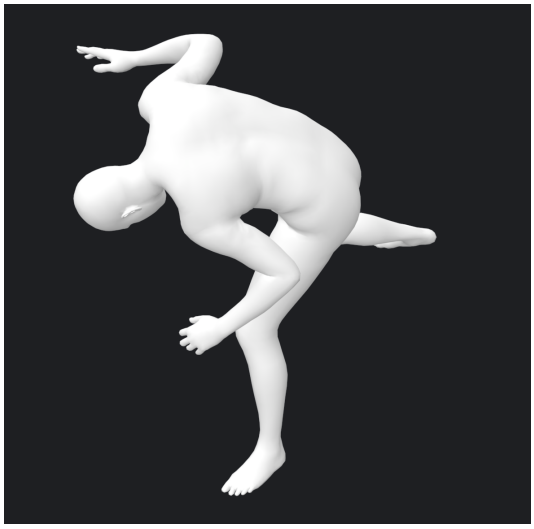


Figure 4.7: Tori - SMPLify-x output

Figure 4.8: Uke - SMPLify-x output

**Optimise.** Model of the tori (Figure 4.7) requires no optimisation, while that of the uke (Figure 4.8) definitely needs some. Not only is the pose of the uke's model unnatural, but also the shape of the body is too thin compared to actual physique of the uke (possibly due to view of the body being a side-view). There are two proposed methods to optimise a model in sub-section 4.1.3. Using Differential Rendering here holds small promise, as the starting point must be near-approximation. Therefore, the alternative body pose prior distribution – vposer-judo (see *Improving body pose prior*, 4.1.3)– was experimented. Figures 4.9 and 4.10 display respectively the experiment result of changing pose prior distribution to vposer-judo and an improvement created by tweaking body shape parameter in configuration file of SMPLify-x [29].

Figure 4.9: Body prior vposer-judo (all other variables controlled)

Figure 4.10: Body shape parameters adjusted

### 4.2.2 Modelling Tomoenage (Shohei Ono, 2019)

Figure 4.11: Tomoenage Tori - the attacker, Source: International Judo Federation [11]

Figure 4.12: Tomoenage Uke - the victim, Source: International Judo Federation [11]

Figures 4.11 and 4.12 display the beautiful execution of Tomoenage (Circular Throw) by Olympic and 3 times World Champion Ono Shohei (Japan) [13] during Judo World Championships 2019 in the final of -73kg men category.

> "The Tomoe-nage (Circular throw) Waza consists of submarining under the opponent and throwing him with a leg throw."
>
> —Judo Channel, Token Corporation [17]

Figures 4.11 and 4.12 are the two source images of the second demonstration model.

**Pre-process.** Since 4.2.1 demonstrated how this process works and intermediate results from this step are shown in the next step (Figures **??**), see Appendix (A.1) for original results.

**Construct 3D model.**

1. Generate keypoints JSON files. Figures 4.13 and 4.14 display the keypoints generated from rotated version of the source photos. The rotation is meant to improve OpenPose's [7] ability to recognise the keypoints based on the assumption that its data prior has lack of examples where humans are upside-down. Appendix A.1 shows OpenPose's output on the original images, which evidently provides no useful information about the fighters' poses.

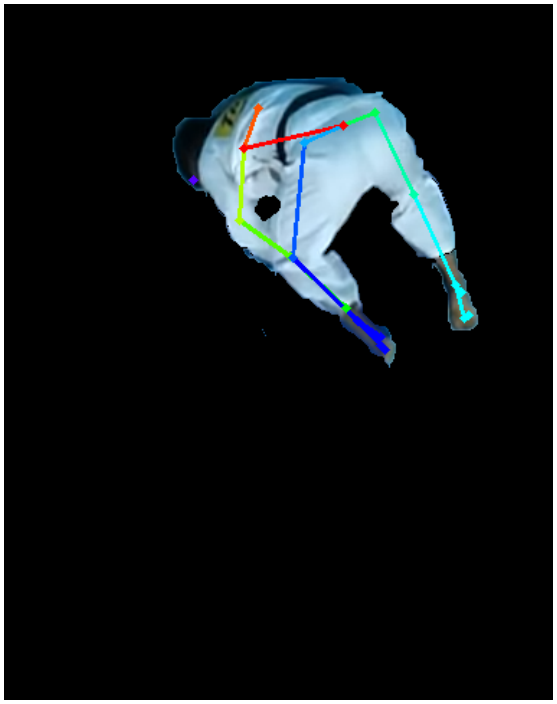


Figure 4.13: Tomoenage Tori - keypoints rendered

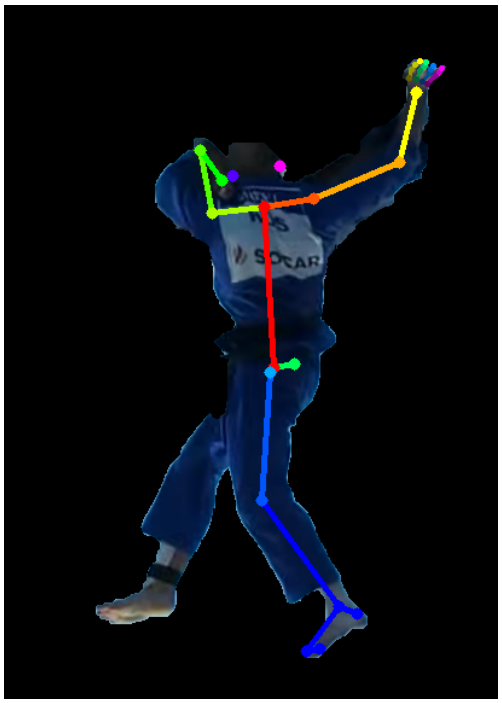


Figure 4.14: Tomoenage Uke - keypoints rendered

2. Generate 3D models. See intermediate results below:

Figure 4.15: SMPLify-x output

Figure 4.16: SMPLify-x output

**Optimise.** Model of the tori (Figure 4.15) requires no optimisation. Model of the uke is optimised with Differential Rendering (see 4.1.3).

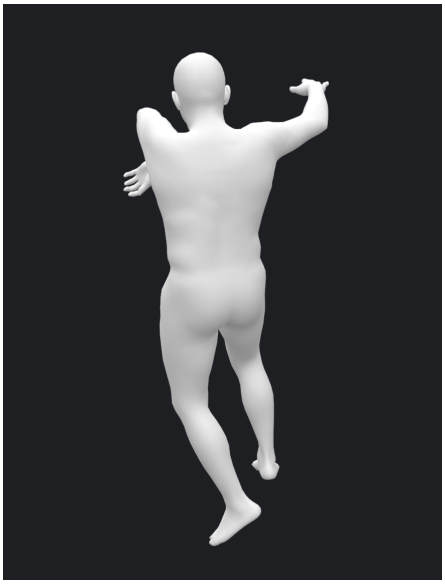


Figure 4.17: Uke - before optimisation

Figure 4.18: Uke - after optimisation

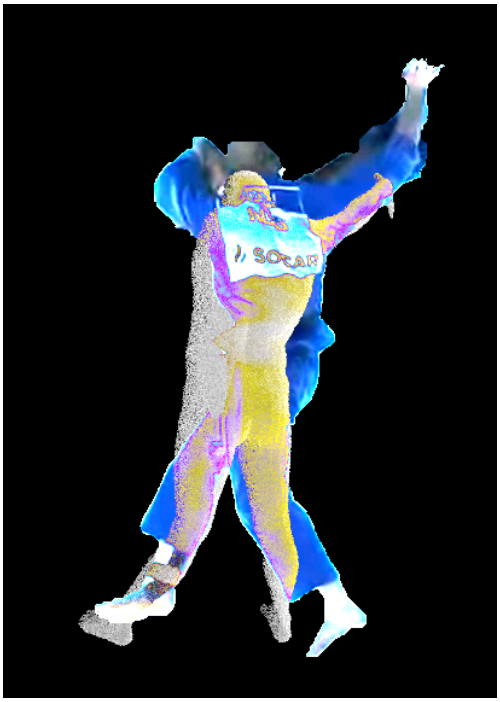


Figure 4.19: Uke - difference between final rendering output and target image

Figures 4.17 and 4.18 show the uke's model before- and after-optimisation. Figure 4.19 shows pixel difference between the target image and rendering output. See Results (5) for comments on the optimisation.

### 4.2.3 Modelling Osotogari (Fonseca Jorge, 2017)

Figure 4.20: Osotogari by Fonseca Jorge (2017). Source: Fighting Films [14]

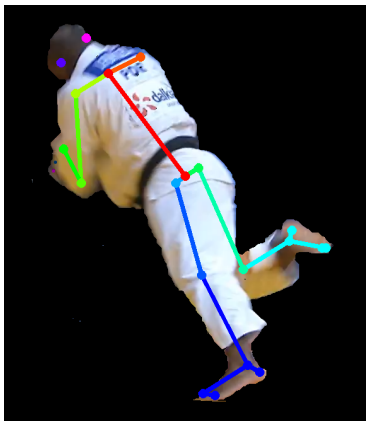
Figure 4.20 displays the incredible execution of the previously introduced technique Osotogari by 2019 World Champion Jorge Fonseca (Portugal) [12] during the 2017 Paris Grand Slam [14]. This image (figure 4.20) is the only source image for the third and last demonstration model.

**Pre-process.** See Figures 4.21a to 4.21c for preprocess results rendered with keypoints. Obstruction is accounted for by filling in missing pixels of the uke's body manually. More accurate keypoints are generated this way (contrast Figures 4.21b and 4.21c).

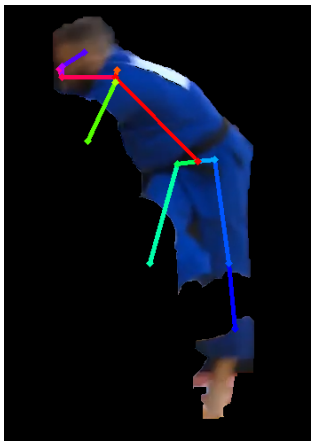
**Construct 3D model.**

1. Generate keypoints JSON files. See Figures 4.21a to 4.21c
2. Generate 3D models. See Figures 4.21d and 4.21g.

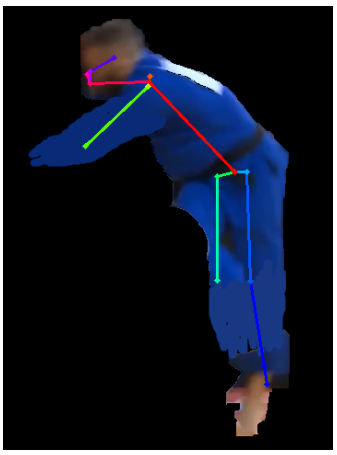
**Optimise.** Both models were optimised using Differential Rendering. See Figures 4.21d to 4.21i. See Results 5 for discussion.



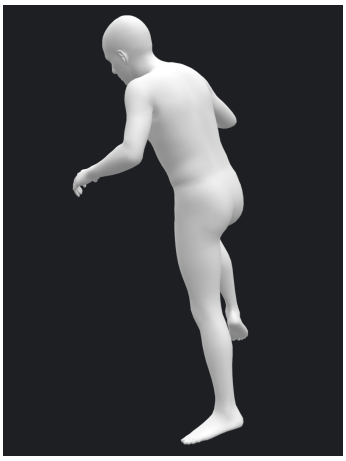
(a) Tori keypoints



(b) Uke keypoints



(c) Uke (filled) keypoints



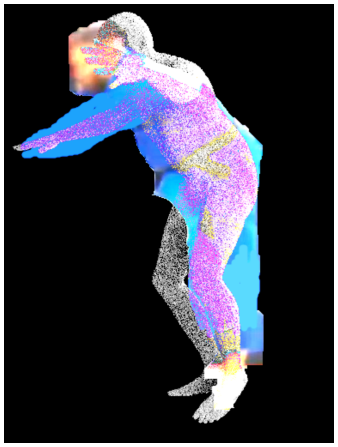
(d) Tori - before optimisation

(e) Tori - after optimisation

(f) Tori - difference between rendering output and target

(g) Uke - before optimisation

(h) Uke - after optimisation



(i) Uke - difference between final rendering output and target image

Figure 4.21: Intermediate results for Osotogari by Jorge Fonseca

# Results

**Osotogari (Battogtokh M., 2020)**

Figure 5.1: Osotogari (Battogtokh M.) final model

Quality of this model is excellent. Not only are the pose and body-shape of each fighter adequately accurate, but also the relative positions of the fighters characterise the judo technique well. The only significant but acceptable inaccuracy is position of uke's right arm, which should have appeared below tori's chest gripped by tori's left hand. For intermediate results, see 4.2.1.

**Tomoenage (Shohei Ono, 2019)**

Figure 5.2: Tomoenage (Shohei Ono, 2019) final model

Quality of this model is good. Relative positions of the two fighters clearly capture the technique being executed. The body poses of both fighters are satisfactory, but both significantly inaccurate. For intermediate results along the steps of the methodology, see 4.2.2.

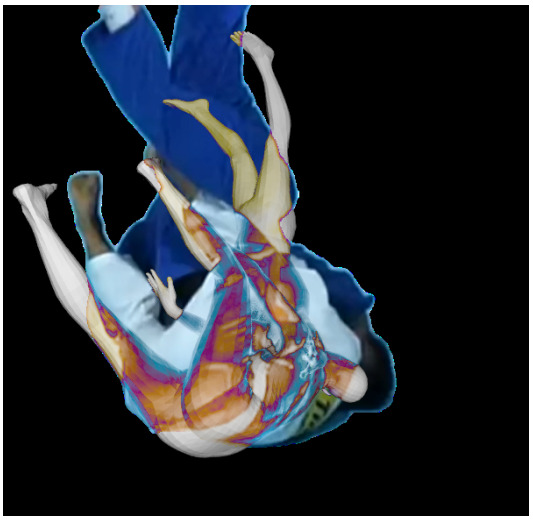
**Osotogari (Fonseca Jorge, 2017)**

Figure 5.3: Osotogari (Jorge Fonseca, 2017) final model

Quality of this model is good. The poses and the shapes of the individual body models are satisfactory and the relative positions of the fighters characterise the technique. However, it has flaws that can be attributed to lack of information; this model was created from a single photo and thus was limited to a single viewpoint only. The most significant flaw is the left arm of the uke being behind the tori. While this detail does not prevent recognising the technique as Osotogari, this characterises the Osotogari to be one executed from a shoulder-grip. The original is an Osotogari executed from double-sleeve grip. For intermediate results along the steps of the methodology, see 4.2.3.

**Supplementary analysis**

(a) Osotogari by Battogtokh M., square mean pixel loss: 0.076



(b) Tomoenage by Shohei Ono, square mean pixel loss: 0.043

(c) Osotogari by Fonseca Jorge, square mean pixel loss: 0.101

(d) Comparing input with irrelevant output for reference, square mean pixel loss: 0.079

Figure 5.4: Error = $|target - img|$ where $img$ is model rendered with *redner*

Figure 5.4 shows errors for each of the demonstration models. Figure 5.4d shows error calculated from comparing a input image with a model that was created from a different input image. The square mean loss calculated for this image serves as a reference. Its value ranges between 0 and 1 inclusive.

As first hypothesised in 3.2.3, the square mean loss as a metric of degree of match is unreliable. For example, Figure 5.4c has higher square mean loss than Figure 5.4d despite being qualitatively better. However, each value is indeed low in their range.

More helpfully, Figure 5.4 assists in qualitative analysis. For instance, it is possible to tell that Figure 5.4a has better overlap (therefore better degree of match) than in Figure 5.4d.

# Evaluation

In this chapter, I evaluate the results (see 5) and the methodologies with respect to background theory as well as general remarks from during the model-creation process (see 4.2). This will include reiterating the methodology with evaluation details of trials and setting up discussions for any future works.

## 6.1 Evaluating the demonstration models

**Osotogari (Battogtokh M., 2020)**

It owes its excellent quality to the fact that two separate images were used to provide a clear and as obstruction-free views as possible for both fighters. SMPLify-x was able to produce a near-perfect model of the tori (the attacker) without the need for any further optimisation or additional trials. However, both the posture and the shape were poor for the SMPLify-x model of the uke (the victim), which was successfully optimised.

Two additional optimisation steps (see Figures 4.9 and 4.10 in 4.2.1):

1. Firstly, the final satisfactory model used VPoser distribution trained on only the PosePrior [23] dataset (vposer-judo) (see 4.1.3), which contains motion capture data of movements like high-kicks and somersaults. This improved the posture of the model to be much more accurate to the source photo despite failing to do so for other poses that are less familiar to the PosePrior dataset.

2. However, because the source photo showed a side-view of the person in question, both the original VPoser distribution and the vposer-judo distribution created models that are too thin in physique. This was overcome with a slight modification to the body-shape parameters of SMPLify-x.

There are the following systematic errors and resulting flaws:

- Uke's right arm was not visible in the source photos (see 4.2.1). This caused SMPLify-x to "guess" its pose. In reality, this right arm must have been low down being pulled by tori's left hand. The systematic error is failing to capture image from an angle where this information is available.
- The two source images were taken at different points in time. Therefore, this model is based on the same throw but two different executions. Thus the final model could not possibly be simultaneously consistent with both source images.
- During the merging phase of the methodology, the relative position of the fighters were approximated manually to be only roughly accurate.

**Tomoenage (Shohei Ono, 2019)**

The original 3D mesh output by SMPLify-x for the uke had a too neutral pose (Figure 4.17), and barely resembled the source photo (Figure 4.12). Therefore, the Differential Rendering was used to optimise the pose. The improvements are:

- The left arm of the uke became stretched out and up like in source (Figures 4.18 and 4.19).
- The legs of the uke became more bent inward with feet outwards (Figures 4.18 and 4.19).

The pose of the uke is still significantly inaccurate because although the rear view of the uke shows uke's legs bent inward and feet outwards, the knees aren't bent in their natural directions like in the source photo. This is because the torso of the original mesh wasn't twisted sideways like in the source photo. Being twisted this way allows the lower-body to be in a side-view while the upper body is in a rear view. Differential Rendering optimisation was applied from a rear-view to ensure uke's left arm was visible in the view (so that its pose could be optimised). However, that forced the legs to take their side-view shape although being looked at from a rear-view, hence causing the slight unnatural bend. Several trials were carried out to balance the accuracy of the left arm against this unnatural bend in the knees.

**Osotogari (Fonseca Jorge, 2017)**

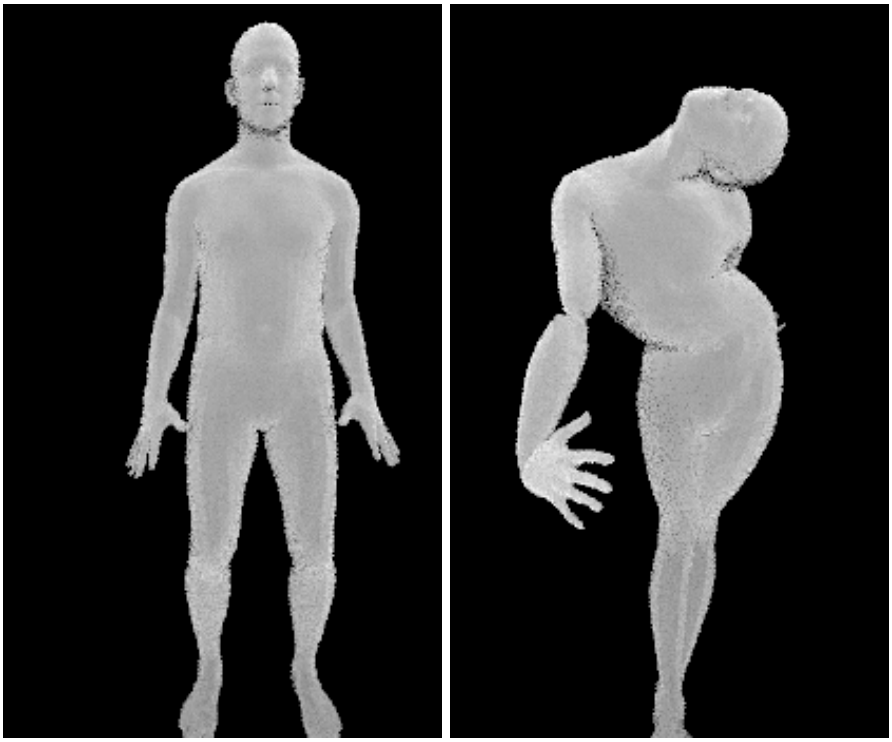
The SMPLify-x models had too neutral poses in the case of both fighters initially (Figures 4.21d and 4.21g). Both models were optimised using differential rendering:

**Tori.** The attacker's pose became less neutral and more forward-leaning like in the source photo (see Figures 4.21d, 4.21e and 4.21f). However, despite several trials, the attacker's right leg could not be bent backwards like in the source. This is understandably due to the fact that the gradient-based optimisation is not guaranteed to find the global optimal pose. In this case, slightly leaning forward reduces loss, but slightly bending the right leg backward *increases* loss. Therefore, while the resulting pose is an improvement, this is a local optimum only.

A classic approach to escaping a local minimum is trying out different initial configurations to the problem — importantly the pose and camera angle in this case. However, as Figure 6.1 shows, starting from a completely random ("unrelated") configuration rather than a close approximation can cause convergence at bad local minimum. The easiest solution is to start optimisation from a different camera angle — one where the two legs have a gap between them so the now visible right leg can change pose.

> Note that it is a property of optimising with Differential Rendering that mesh vertices not visible in the output do not contribute to the output by definition of the path-tracing algorithm (the rendering algorithm used). Therefore, there are no errors back-propagated to these mesh vertices (and subsequently the pose parameters dictating these vertices' positions).

Different approaches might be selectively randomising the pose of only the right leg if possible or performing interactive evolutionary algorithm with population comprising of slight variations of the starting pose. These need more time and/or computational resource (GPU).



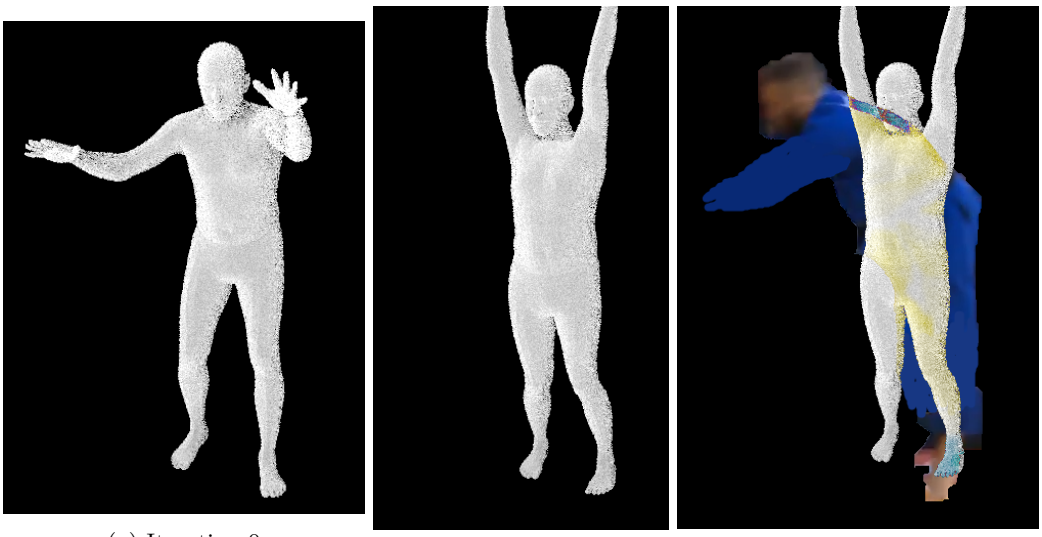
(a) Iteration 0 (b) Iteration 190

Figure 6.1: Optimisation from a neutral pose

**Uke.** First optimisation trial of the uke (Figure 6.2) wasn't used in the final model. While the legs came closer together to match the source photo, pose of the arms went astray. In this case, "lifting" the arms initially reduces the loss, but then this "lifting" is overdone until unacceptable. This phenomenon can be explained by two hypotheses:

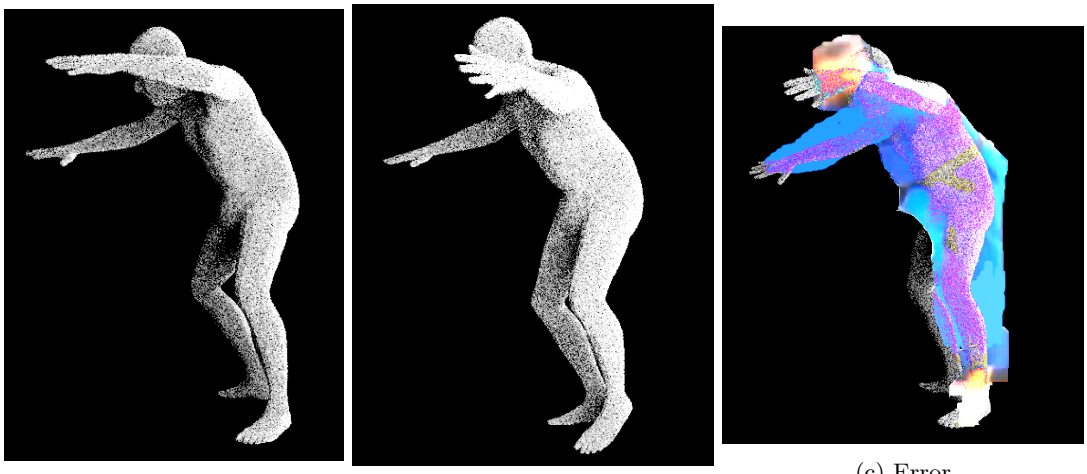
- The net change in loss is favorable. In this case, loss is more reduced from the legs coming together than it is increased due to the arms being raised. This is true at least initially until the net loss turns unfavorable and optimisation is stuck at local minimum.
- Calculations of the gradients for the pose parameters are inaccurately estimated. In redner (our Differential Renderer), Monte-Carlo sampling is used to represent the pixel loss [19]. However, this cannot be proper estimate for certain types of parameters, e.g. translation, according to Loubet et al. (2019) [21].

However, this local minimum (Figure 6.2) is easy to prevent by simply starting from a different configuration — same pose but different camera angle. Figure 6.3 shows optimisation starting from a camera angle more accurately matching that in the target photo. At iteration 50, the optimisation has reduced the error by pulling back the left arm of the uke and slightly bringing the legs together. While bringing the legs together is good, it would be better to lower the left arm rather than pulling it back. Nevertheless, the error reduces from Iteration 0 by both of these pose changes, and the final error (Figure 6.3c) is visibly better than error in Figure 6.2c (more overlap the better).



(a) Iteration 0 (b) Iteration 50 (c) Error

Figure 6.2: Optimisation starting from bad camera angle



(a) Iteration 0 (b) Iteration 50 (c) Error

Figure 6.3: Optimisation starting from good camera angle

## 6.2 Evaluating the methodology

**Resource constraints.**

It is important to acknowledge the resource constraints of this project when evaluating the methodology. The most significant resource constraint – one that affects every step of methodology and their results — is time.

For example, for training VPoser (see 4.1.3) to obtain alternative pose prior distribution without CUDA (GPU boost [27]), 1 epoch of training takes approximately 1 hour while the original distribution takes 10000 epochs. This made anything but minimal of such training infeasible.

Similar limitation due to lack of CUDA-compatible GPU applies to Differential Rendering based optimisation. 50 iterations of optimisation takes approximately 5 minutes. This forces optimisation to be minimal— only minor pose changes as opposed to for instance both camera and pose being optimised simultaneously. There are many other possible approaches to optimising with Differential Rendering that were not feasible due to this lack of computational resource (which ultimately leads to lack of time).

Generally, time as a resource was invested the most into more "interesting" steps like SMPLify-x, optimisation with Differential Rendering, and body pose prior training that contribute the most to the aim of this project: creating 3D models from 2D images by exploring state-of-the-art 3D scanning technologies (as opposed to less "interesting" tasks such as automating the process of pipe-lining output of one step as input into another, which often takes trivial human effort but significant development time to automate).

### 6.2.1 Pre-processing

**Clearing distractions.**

(a) Raw image

(b) Grabcut image

Figure 6.4: OpenPose output

Figure 6.4 illustrate the keypoints generated for a raw image versus Grabcut image focusing on the tori only. Apparently, Grabcut [32] effectively prevents keypoints being generated for irrelevant people — the spectators and even the uke in this case when we must focus on only one person due to limitation of SMPLify-x. Additionally, notice that the keypoints for the tori contains more information in 6.4b than the same in 6.4a. Namely, there are no keypoints detected for the right arm of the tori that is on uke's body in 6.4a, but this information is available in 6.4b. Therefore, clearing distractions with Grabcut is not only necessary in order to focus on one fighter at a time, but also empirically effective in achieving a more accurate model.

The limitation of this approach surfaces when being applied for the uke, who is obstructed by the tori. This is accounted for next.

**Clearing obstructions.**

Best results were obtained for the models that were built from two photos. This is in part because the two photos show each fighter separately with as little obstruction as possible.

When obstruction is unavoidable (for example in case of restricting the number of source photos to only one), the simple approach of separating the fighters and filling in the missing obstructed pixels (see 4.1.1) has effectively guided the generation of more desirable keypoints. Figures 4.21b and 4.21c contrast body keypoints detected when missing pixels are *not filled-in* against *filled-in*. In this case, there is a difference in the OpenPose output, which will lead to different SMPLify-x output (3D models).

On the other hand, filling in the missing pixels has no effect on model-construction when applied after OpenPose keypoints-detection (see 4.1.2), but only when applied before inputting the image to OpenPose. Figures 6.5 and 6.6 contrast test models created with the **same** keypoints, for the *not filled-in* and *filled-in* conditions. This means that the pixels were filled in after OpenPose had already detected the keypoints. In this case, the figures (6.5 and 6.6) show no difference in output models. These emphasize the importance of OpenPose keypoints to the output 3D models.

Notably, while the approach is simplistic, it should definitely guide optimisation with Differential Rendering since the entire optimisation is based on simple difference in pixel values of rendering output and the target image.

So far, this approach of filling in pixels is effective for guiding to more desired outputs and motivates any future work for automatically "imagining" what obstructed parts of an object should look like.

(a) Source image (b) Model

Figure 6.5: SMPLify-x produced test models

(a) Source image (b) Model

Figure 6.6: SMPLify-x produced test models - pixels filled

However, the simple approach cannot do anything when for example, the arm of the attacker is obstructed by the attacker's own back as in 4.2.3. That forces SMPLify-x to "guess" the position of that hand and possibly causes an important problem: during the merging process, inaccurate pose of the obstructed hand may prevent placing of the other fighter in the correct position without the body surfaces intersecting.

**Summary** The pre-processing methodology effectively reduces the input to be digestible by clearing distractions with Grabcut, allowing to focus on modelling one fighter at a time and thus improving the accuracy of the individual models. However, while Grabcut can reduce unnecessary information from the images, it cannot recover pixels missing due to obstructions. The simplistic approach of filling in missing pixels with a solid colour effectively guides the accuracy of a model when a fighter's body-parts aren't self-obstructed by his/her own other body-parts.

### 6.2.2 Constructing individual models

#### High dependency on OpenPose output

Quality of any final output is highly dependent on the initial 2D pose estimation by OpenPose, as previously implied in evaluation of pre-processing methodology (see 6.2.1). To demonstrate this point, let us consider the result of attempting to estimate body pose with Differential Rendering optimisation only — without OpenPose pose-estimation — in 6.1. As evident there, Differential Rendering optimisation must start from a starting configuration that is already an approximate of the final output.

In theory, with enough computational resources, the Differential Rendering based optimisation approach would suffice to create the final models, without any need for an external pose-estimator (a theoretical advantage over SMPLify-x). This would be possible if we apply the optimisation from sufficient number of systematically randomised starting configurations (pose and camera angle).

However, the practical disadvantage is that optimising even one of such starting configuration is computationally expensive (requiring minutes when on CPU-only). Furthermore, even if optimisation from one starting configuration is sped-up with more computational resource (GPU), it is not clear how many random starting positions must be trialed until a satisfactory result will be obtained.

This disadvantage is inherent in the general ill-posed nature of 3D reconstruction problem:

> "...given a single observation image, there are infinitely many possible 3D scenes that will render exactly like the observation".
>
> —3DMM Face Fitting, *redner* tutorials [20].

Optimising with respect to body pose as opposed to mesh vertices narrows the possibilities down to *human bodies only* from *any 3D shape*. But, it is not clear whether that is narrow enough for a satisfactory result to be obtained feasibly, because even now the possibilities are the union of natural and unnatural human body pose spaces — effectively arbitrary.

What is clear is however that OpenPose's pose-estimation narrows the human-pose space to just a single good estimation, allowing any further optimisation to start from a single "good" configuration. Therefore, given the resource constraints of this project, OpenPose pose-estimation is practically essential and quality of all results are highly dependent on its output.

Constructing the 3D models with SMPLify-x also depend on the process of training body pose prior distribution. Optimising the 3D model output by training alternative distributions (see 6.2.3) is expensive. Therefore, another limitation of this methodology is the resource constraint (see 6.2).

**Summary** A limitation of the methodology to construct the 3D models is its high dependency on output of the generating-keypoints step: the OpenPose output. Interestingly, this is true even when there is the theoretical possibility to avoid using OpenPose at all, as evidenced in 6.1. Another limitation is the resource constraint imposed by how expensive training an alternative pose distribution is.

### 6.2.3 Optimising

**Body pose prior model, training and resource constraints**

As result of dependency on prior data, certain 3D models have too neutral poses when compared to its source image (see results for Osotogari by Jorge Fonseca 6.1). Furthermore, there are no intelligible/acceptable results for many poses (see Figures 6.7 and 6.8).

(a) Source image (b) Model

Figure 6.7: Trial modelling the tori of a Koshiguruma execution

(a) Source image (b) Model

Figure 6.8: Trial modelling the tori of a Uranage execution

Rotating images based on the assumption that most prior data poses (of both OpenPose and VPoser) are standing poses (where the head is on top of the image) is an effective workaround for some limitations due to the prior data dependency (see 6.1).

Training an alternative VPoser pose-distribution is more long-term strategy but subject to resource constraint (see 6.2). This approach showed its effectiveness by producing a better pose for the uke in result for Osotogari (Battogtokh M., 2020) in 6.1. But since training on a single (suitable) dataset (see 4.1.3) for only 10 epochs took more than 5 hours, this approach is too expensive.

**Summary** Training an alternative body pose prior distribution is effective but expensive.

**Optimisation with Differential Rendering**

In Methodology (4.1.3), high-level description of the Differential Rendering optimisation is given. There I only explain the final version of this novel methodology. Here, we evaluate the methodology with respect to its initial version first, and then the results.

**Optimising with respect to mesh vertices directly, and 2D ambiguity** My initial idea was to directly optimise with respect to the mesh vertices directly. While this approach resulted in models that match the target image with incredible accuracy **when rendered to 2D** (see Figures 6.9b, 6.10b and 6.11b), these models —oblivious to any human-body shape or in fact any 3D object shape constraints— are unintelligible (Figures 6.9a, 6.10a and 6.11a). It is too difficult to control the output space of the 3D models.

Two interesting attempts to constrain the output space are:

1. Showing only a part of the 3D model to the renderer's camera (see 6.11). This allows to selectively optimise subset of the model's vertices, which is an advantage because the output space is constrained to a space where only a selected subset of the vertices are variable and the rest constant.

   Theoretically, this is possible because *path-tracing* is used to render the 3D models. Path-tracing renders a pixel in its output image by tracing a ray from the camera, through the pixel, onto some mesh face (simply 3 vertices forming a triangle) of the object, to a light source.

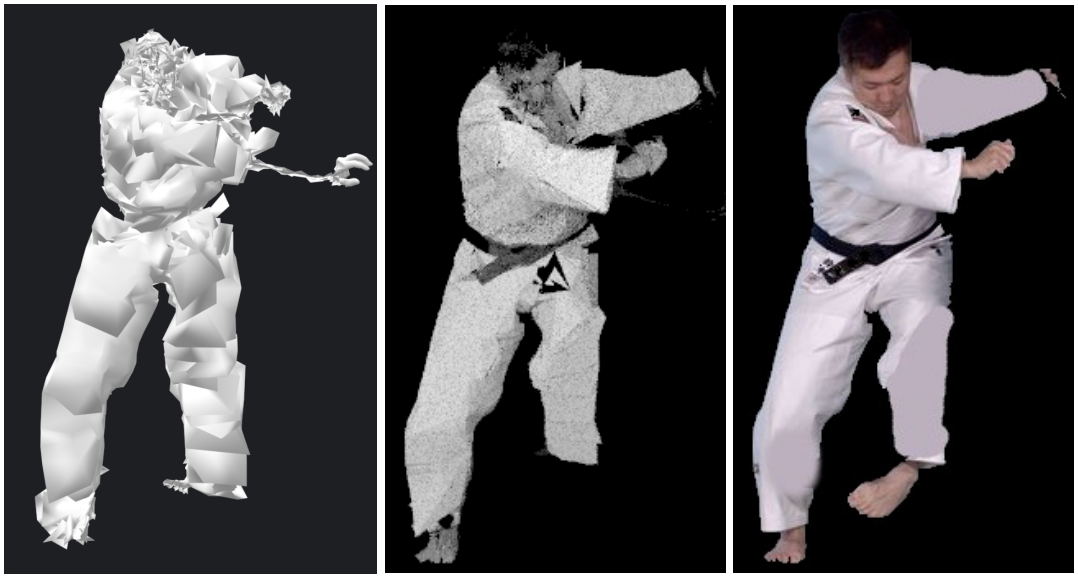
   Any mesh face $f$ that does not appear in the camera of the path-tracer does not contribute anything to the output, which means no error of the output can be attributed back to $f$. This is why in Figure 6.11, the upper body that does not appear in the rendering output is left unchanged while the leg area of the mesh is changed.

2. Defining a loss that accounts for changes in the mesh normals. A normal is an orthogonal vector to a plane in space – or in other words, the direction the plane faces.

   The idea of accounting for change in normals is to discourage a model's surface triangles from facing away from its original direction. This reduced the noise in the test model's surface, but not sufficiently in some parts (contrast Figures 6.9 and 6.10). Importantly, this is a single example of different possibilities for defining the loss function to constrain output space to a more desirable one.

**Summary** Optimising with respect to mesh vertices directly suffers from the difficulty to constrain the resulting model to a "reasonable" surface shape. But the approach has merit in the following two properties observed:

- Ability to selectively optimise.
- Ability to incorporate arbitrary constraints through the loss function.

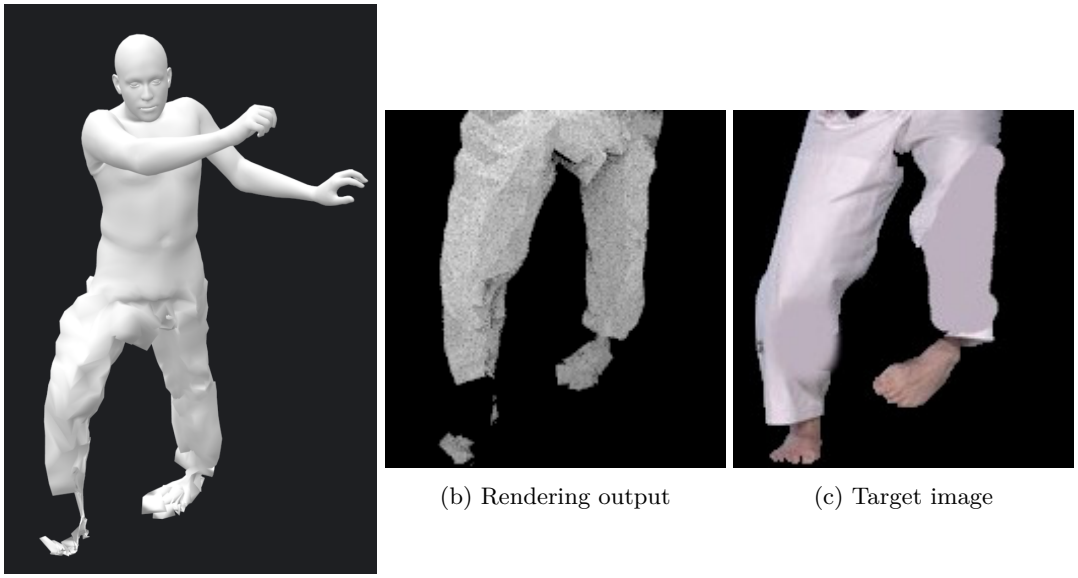


(a) 3D mesh (b) Rendering output (c) Target image

Figure 6.9: Error = sum square of pixel errors

(a) 3D mesh (b) Rendering output (c) Target image

Figure 6.10: Error = sum square of pixel errors + sum square of normal errors



(a) 3D mesh

(b) Rendering output

(c) Target image

Figure 6.11: Error = sum square of pixel errors (for legs only)

**Optimising with respect to pose, and local minima** In the previous section, we observed that the disadvantage of optimising with respect to mesh vertices directly is difficulty to impose human-body shape constraints. I looked at ways to group vertices semantically by body-parts such that the vertices that constitute the same body-part "move" together without manifesting chaotic surface, which is an approach similar to what existing 3D editing tools do [9]. There is a mapping between vertex index and body joints in SMPLify-x, but this is not enough information to tell which set of vertices correspond to which body-part.

To be more resourceful, in order to capitalise on the human-body shape and pose constraints of SMPLify-x, it is desirable to be able to optimise with respect to the body pose because VPoser automatically translates the pose to a reasonable human-body shape. Fortunately, VPoser is end-to-end differentiable, which means that it is possible to differentiate the 3D mesh vertices (their positions) with respect to the pose.

*redner* [19] on the other hand can automatically differentiate the pixels of its output image with respect to the rendered 3D mesh's vertices. By chain rule, it is therefore possible to differentiate the pixels of *redner*'s output with respect to VPoser's pose. Therefore, the error of rendering output can be back-propagated all the way to the pose. This allows calculating gradients and gradient-descent optimisation on the pose.

Optimising with respect to pose has following advantages:

- Constrains mesh shape to a human-body shape (major advantage over optimising with respect to mesh vertices directly).
- Optimises according to the instinctive quality-validation method proposed in 3.2.3.
- In theory, makes an external pose-estimation (by OpenPose) unnecessary.
- In theory (not tested), allows selectively optimising the pose of a particular body-part.

However, Differential Rendering based optimisations generally suffers from sensitivity to initial configuration (contrast 6.2 and 6.3). Therefore, this type of optimisation must be applied

starting from a near approximation, as so does Loubet et al. (2019) [21]. The sensitivity was explained in terms of local minima in 6.1. Often, the loss is observed to reduce, but the pose qualitatively gets worse. This is because the starting configuration leads the optimisation to a local minimum.

**Inaccurate gradient estimate due to Monte-Carlo sampling error** A major theoretical limitation of the Differential Rendering optimisation is based on the limitation of *redner* [19]. As explained by Loubet et al. (2019) [21], Monte-Carlo sampling does not accurately estimate gradients for certain type of parameters. Loubet et al. (2019) solves this problem through "reparameterizing discontinuous integrands".

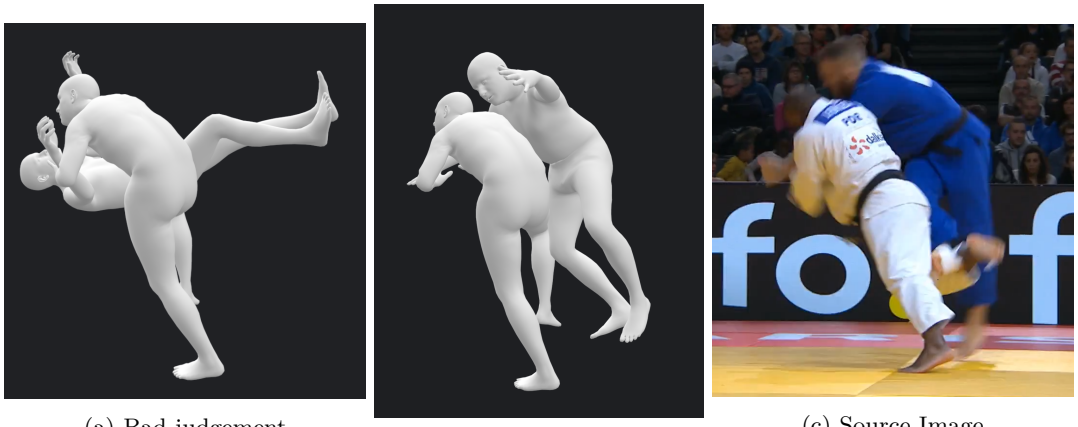
This is a theoretical limitation because the code for Loubet et al. (2019) was released too late within this project's time-frame, and once again the time constraint has not allowed me to empirically test whether this limitation causes *redner* to perform inferior to Loubet et al. (2019).

**Resource constraint** Path-tracing, and back-propagation of error are computationally expensive. It takes about 5 minutes for 50 iterations of the Differential Rendering optimisation.

**Summary** Optimising with respect to pose not only possesses the same merits as optimising with respect to vertices (ability to selectively optimise, ability to incorporate arbitrary constraints through the loss function), but also has the additional advantages that the body-shape and pose are successfully constrained thanks to SMPLify-x. But Differential Rendering optimisation generally is sensitive to the initial configuration, which is a limitation. Other limitations are the resource constraint and possible inaccuracies in back-propagated gradients due to Monte-Carlo sampling error [21].

### 6.2.4 Merging

Merging the individual models together is usually a trivial human-task with help of the minimal *translate* and *rotate* functions (see 4.1.4), but a subject to risk of human systematic errors — failure to judge the optimal translation and rotation. Since the translation and rotation functions (see 4.1.4) must be manually used in combination, the merging process relies on the human operator's judgement. For example of a failed judgement see the Figure (6.12) below:



(a) Bad judgement (b) Good judgement (c) Source Image

Figure 6.12: Systematic error in judgement

In Figure 6.12a, it was judged that uke should be in a side-view. This caused uke to be in a horizontal position in contrast to the source in Figure 6.12c, where the uke is as upright as the tori. Because, the obstructed right hand of the tori did not allow any more upright position without the body surfaces intersecting in that particular side-view angle. However, upon a better judgement (in Figure 6.12b) that uke should be in half side- and half front-view, much more accurate relative positions are possible without any surface intersections.

**Summary** The merging step is implemented with a minimal but effective methodology that reduces the problem to usually trivial human-task. However, the implementation approach can be subject to human systematic errors.

# Legal, Social, Ethical and Professional Issues

I follow the Codes of Conduct for Computing Professionals issued by the BCS [5]:

- I make IT for everyone
- I show what I know, and learn what I don't
- I respect the organisation/institution and individual I work for/with
- I keep IT real. I keep IT Professional. I pass IT on.

I acknowledge the following essential works that I extensively use within this project as **not** mine, I claim no credit for their findings or artefacts and I respect and agree to the licenses under which I have used them:

- SMPLify-x [28] (for non-commercial scientific research purpose).
- *redner* [19] (MIT License).

The images used in this project are copyrighted to their authors. I especially thank International Judo Federation [11] and Fighting Films [14] for making the source images I have used in this project available.

# Conclusion and Future Work

## 8.1 Conclusion

The thematic challenge of this project has been the ill-posed nature of the 3D-reconstruction problem. Consequently, constraining the output space to a desired space has repeatedly been required. The major examples include:

- Constraining output model space of 3D-construction algorithm (initially any 3D scene as we started with photogrammetry) to desired space (human bodies only, with SMPLify-x).
- Constraining output pose space of SMPLify-x (initially determined by default VPoser pose prior distribution [27]) as narrowly as possible down to the poses in the source photos (using a alternative pose prior and Differential Rendering optimisation, see 4.1.3).
- Constraining output model space of Differential Rendering optimisation (initially any 3D object when optimising with respect to vertices) to desired space (human bodies only, by optimising with respect to VPoser pose, see 6.2.3).

I employed state-of-the-art computer-science/mathematical techniques (SMPLify-x [28], photogrammetry [4], Differential Rendering *redner* [19]) against these challenges, and even a novel optimisation approach (Differential Rendering optimisation with respect to pose) was born out of combining the diverse existing approaches applicable to our 3D-reconstruction problem. Given the diversity of approaches that solve the 3D-reconstruction problem and the resource constraints of this project, there are many possible techniques still yet to try.

That being said, let us revisit the aim (in 2.1.3) of this project before concluding with respect to it:

> Develop software/methodology that takes **input image(s)** (.jpeg, .png) of a judo scene with up to 2 people, and **outputs a 3D-model** (Wavefront .obj [24] or other) file capturing the fighter(s), by exploring alternative methodologies existing in (building from) state-of-the-art 3D reconstruction technologies.

To conclude, this project is successful with its aim, as:

1. The developed methodology makes constructing 3D-model of a judo scene possible with as little as a single image (see 5, Results).
2. The development of the methodology explores, implements, tests and evaluates diverse state-of-the-art 3D scanning technologies.

## 8.2 Future Work

The most novel and promising future work spurring from this project would be to further explore the possibilities of applying Differential Rendering to optimise the judo 3D-models. Loubet et al. (2019) [21] demonstrated impressive effectiveness of Differential Rendering for automatically optimising an approximate model with respect to various rendering parameters, which this project was unable to test due to resource constraints. However, this project has laid the foundation by:

- Offering an approximate model to start the optimisation from
- Offering an approach to constrain the optimisation output space to human bodies.

The promise of Differential Rendering is beyond the minimal quality models, which means that there is the potential to automatically optimise the models to have shape, texture, clothing etc. that matches the source images with incredible accuracy. This project has revealed a powerful property of Differential Rendering, which is that optimisation can selectively focus on sub-parts of the 3D-models once at a time on attention basis (see 6.2.3). The power of this is the potential to divide-and-conquer a complex 3D-object by selectively optimising it one simple bit at a time.

# Bibliography


[1] Alexander Alekhin, Muhammad Taha, and Steven Puttemans. grabcut.py. https://github.com/opencv/opencv/blob/master/samples/python/grabcut.py, 2020. commit 04fad57fc15c845f7e70105691102f2af9ff469d, OpenCV GitHub repository, accessed 15 January 2020.

[2] AliceVision. Meshroom: Open source 3d reconstruction software. https://www.youtube.com/watch?v=v_O6tYKQEBA&feature=emb_logo, January 2018. YouTube, accessed November 2019.

[3] AliceVision. Meshroom. https://alicevision.org/#meshroom, 2019. accessed November 2019.

[4] AliceVision. Photogrammetry pipeline. https://alicevision.org/#photogrammetry, 2019. accessed November 2019.

[5] The Chartered Institute for IT BCS. Bsc code of conduct, version 5. https://www.bcs.org/membership/become-a-member/bcs-code-of-conduct/, 2015. Last reviewed 2019, accessed 23 March 2020.

[6] Zhe Cao, Gines Hidalgo, Tomas Simon, Shih-En Wei, Hanbyul Joo, and Yaser Sheikh. Openpose: Real-time multi-person keypoint detection library for body, face, hands, and foot estimation. https://github.com/CMU-Perceptual-Computing-Lab/openpose, 2019. Github Repository, accessed November 2019.

[7] Zhe Cao, Gines Hidalgo, Tomas Simon, Shih-En Wei, and Yaser Sheikh. OpenPose: realtime multi-person 2D pose estimation using Part Affinity Fields. In *arXiv preprint arXiv:1812.08008*, 2018.

[8] Zhe Cao, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Realtime multi-person 2d pose estimation using part affinity fields. In *CVPR*, 2017.

[9] Remington Creative. Rig and animate anything in blender. https://www.youtube.com/watch?v=mhQY2_gVoVg, 2016. Youtube, accessed 8 March 2020.

[10] Natalia Criado and Steffen Zschaler. Individual project guide to deliverables. https://keats.kcl.ac.uk/pluginfile.php/4348794/mod_resource/content/4/BSC-BEng%20Individual%20Project%20Guide%20to%20Deliverables.pdf, 2017. Last revised 6 September 2019, accessed November 2019.

[11] International Judo Federation. World judo championships 2019: Day 3 - final block. https://www.youtube.com/watch?v=i0V1GbTZY4E&t=3694, August 2019. video interval 3694s-3700s, Youtube, accessed 27 August 2019.

[12] International Judo Federation. Jorge fonseca. https://www.ijf.org/judoka/9396/results?results_rank_group=all, 2020. International Judo Federation, accessed 29 March 2020.

[13] International Judo Federation. Ono shohei. https://www.ijf.org/judoka/4018/results?results_rank_group=all, 2020.

[14] Fighting Films. 101: Osoto gari - fonseca (por) v frey (ger) -100kg. https://www.youtube.com/watch?v=Nr5EcG-blY0, 2018. Youtube, 8 March 2020.

[15] Absolute Geometries. 3d scanning, laser scanning. http://www.absolutegeometries.com/3D-Scanning.html, 2009. accessed 10 April 2020.

[16] Mordor Intelligence. 3d scanning market - growth, trends, and forecast (2019 - 2024). https://www.mordorintelligence.com/industry-reports/gloval-3d-scanners-market-industry, 2018. accessed 6 December 2019.

[17] Token Corporation Judo Channel. Tomoe-nage (circular throw). https://www.judo-ch.jp/english/dictionary/technique/nage/masute/tomoe/, 2020. Judo Channel, Token Corporation, accessed 29 March 2020.

[18] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *3rd International Conference for Learning Representations, San Diego, 2015, arXiv:1412.6980v9*, 2017. Version 9.

[19] Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. Differentiable monte carlo ray tracing through edge sampling. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)*, 37(6):222:1–222:11, 2018.

[20] Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. 3dmm face fitting. https://nbviewer.jupyter.org/github/BachiLi/redner/blob/master/tutorials/3dmm.ipynb, December 2019. redner tutorials, accessed 1 March 2020.

[21] Guillaume Loubet, Nicolas Holzschuch, and Wenzel Jakob. Reparameterizing discontinuous integrands for differentiable rendering. *Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 38(6), December 2019.

[22] Guillaume Loubet, Nicolas Holzschuch, and Wenzel Jakob. Reparameterizing discontinuous integrands for differentiable rendering. https://rgl.s3.eu-central-1.amazonaws.com/media/papers/Loubet2019Reparameterizing.mp4, 2019. Video presentation, accessed 8 February 2019.

[23] Naureen Mahmood, Nima Ghorbani, Nikolaus F. Troje, Gerard Pons-Moll, and Michael J. Black. Amass: Archive of motion capture as surface shapes. In *The IEEE International Conference on Computer Vision (ICCV)*, Oct 2019.

[24] Blender Manual. Wavefront obj. https://docs.blender.org/manual/en/2.80/addons/io_scene_obj.html, 2020. Blender 2.80 Manual, accessed 27 March 2020.

[25] Neil Ohlenkamp. Osotogari. https://judoinfo.com/quiz0297_3/, 2007. Judo Info, accessed 29 March 2020.

[26] OpenCV. Interactive foreground extraction using grabcut algorithm. https://docs.opencv.org/3.4/d8/d83/tutorial_py_grabcut.html, 2020. accessed 15 January 2019.

[27] Georgios Pavlakos, Vasileios Choutas, Nima Ghorbani, Timo Bolkart, Ahmed A. A. Osman, Dimitrios Tzionas, and Michael J. Black. Vposer: Variational human pose prior. https://github.com/nghorbani/human_body_prior, August.

[28] Georgios Pavlakos, Vasileios Choutas, Nima Ghorbani, Timo Bolkart, Ahmed A. A. Osman, Dimitrios Tzionas, and Michael J. Black. Expressive body capture: 3d hands, face, and body from a single image. In *Proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[29] Georgios Pavlakos, Vasileios Choutas, Nima Ghorbani, Timo Bolkart, Ahmed A. A. Osman, Dimitrios Tzionas, and Michael J. Black. Expressive body capture: 3d hands, face, and body from a single image. https://github.com/vchoutas/smplify-x, June 2019. Github repository, accessed November 2019.

[30] Winter Green Research. Blockchain market shares, market strategies, and market forecasts, 2018 to 2024. page 42, 2018. https://www.ibm.com/downloads/cas/PPRR983X, accessed 6 December 2019.

[31] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, pages 400–407, 1951.

[32] Carsten Rother, Vladimir Kolmogorov, and Andrew Blake. "grabcut": interactive foreground extraction using iterated graph cuts. *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 309–314, 2004.

[33] Tomas Simon, Hanbyul Joo, Iain Matthews, and Yaser Sheikh. Hand keypoint detection in single images using multiview bootstrapping. In *CVPR*, 2017.

[34] SmartMobileVision. Scann3d. https://play.google.com/store/apps/details?id=com.smartmobilevision.scann3d\&hl=en, 2017. Google Play, accessed 27 August 2019.

[35] Shih-En Wei, Varun Ramakrishna, Takeo Kanade, and Yaser Sheikh. Convolutional pose machines. In *CVPR*, 2016.

[36] Tianfan Xue, Michael Rubinstein, Ce Liu, and William T Freeman. A computational approach for obstruction-free photography. *ACM Transactions on Graphics (TOG)*, 34(4):1–11, 2015.

# Appendix

## A.1 Pre-process results for Tomoenage by Shohei Ono

Figure A.1: Tori - result of pre-processing rendered with keypoints

Figure A.2: Uke - result of pre-processing rendered with keypoints (none recognised)

# User Guide

## B.1 Instructions

Note that the process may require you to do complicated time-consuming environment set up. All information about how to set up the environment for each dependency is available with their source code on Github (see Dependencies). Refer to their respective Github repositories and documentations I provide here if you wish to replicate the results.

1. Pick input image(s) (see 3.2.1)
2. Pre-process with Grabcut [26]. Follow the sample provided on OpenCV Github repository [1]. Alternatively, use slightly modified (made interaction simpler) version of this sample by downloading:

   ```
   https://github.com/MunkhtulgaB/golden-score/blob/master/Grabcut/
       grabcut.py
   ```

   Then place your input image in the same directory and run:

   ```
   python grabcut.py <input image>
   ```

   to start the interactive foreground detection. Reduce image to one fighter only; repeat for each fighter so you will have 2 images at the end of this step. For example, see 4.2.1. Tested environment: Windows 10.
3. (a) Generate keypoints using OpenPose for each of the 2 images you have. Examples in 4.2.1. See all details on:

   ```
   https://github.com/CMU-Perceptual-Computing-Lab/openpose
   ```

   Tested environment: Windows 10.

   (b) Construct the model using SMPLify-x by running:

   ```
   python3 smplifyx/main.py --config cfg_files/fit_smplx.yaml --
       data_folder <input directory> --output_folder <output
       directory> --model_folder <model folder> --gender male
   ```

   At the end of this step, in the output directory you will have `meshes` where Wavefront OBJ [24] 3D models of the fighters are created. You will also have .pkl files from which you can recover the model objects in the `results` subdirectory. See tutorial on SMPLify-x Github repository for any details.
   Tested environment: MacOS Mojave

4. (Optional) Optimise each individual model using:

   (a) An alternative pose prior distribution. Add the following directory under the directory you installed SMPLify-x and name it vposer_judo:

   `https://github.com/MunkhtulgaB/golden-score/tree/master/vposer_training/training/judo`

   To use this pose prior distribution instead of the default VPoser distribution (which is named vposer_v1_0 by default), change the following line in `<installation folder>/smplify-x/cfg-files/fit-smplx.yaml`:

   `vposer_ckpt: "../vposer_v1_0"`

   with

   `vposer_ckpt: "../vposer_v1_0"`

   (b) Differential Rendering optimisation. The source code to optimise pose of a fighter is provided in:

   `https://github.com/MunkhtulgaB/golden-score/blob/master/vposer/optimize_pose.py`

   Download and change `MODEL_NAME` constant to the name of the model `<input>` you wish to optimise. Then place the .pkl file from the `results` subdirectory mentioned in step 3b in:

   `./input/source/<input>/`

   Also place the target images you wish to optimise against in:

   `./input/targets/`

   Then simply run with:

   `python3 optimize_pose.py`

   At the end of this step, you will have modified versions of the two 3D models for each fighter.

5. Merge the individual 3D models together. Do this by first downloading:

   `https://github.com/MunkhtulgaB/golden-score/blob/master/gripping/grip.py`

   Then place the two models in the same directory, and run with:

   `python3 grip.py --tori <tori's OBJ model file> --uke <uke's OBJ model file>`

   Upon success, it will print:

   `Gripped model written to: grip.obj`

   The final model is then now output as `grip.obj`

**Note** You must manually configure the necessary translation and rotation of the uke by changing the following lines:

```
translation = (-0.4, 0.05, 0.15)
pivot = (0,0,0)
rotation_angle = -math.pi/2
rotation_axis = 1

rotation_angle1 = math.pi/7
rotation_axis1 = 0

rotation_angle2 = 0
rotation_axis2 = 2
```

This will require some experimentation, and knowledge of transformations in 3D space. But this is intuitive, and often trivial if you experiment starting from setting the rotation angles to `math.pi/2` (90 degrees).

# Source Code

I verify that I am the sole author of the programs contained in this folder, except where explicitly stated to the contrary.

Munkhtulga Battogtokh (Sunday 5th April 2020)

## C.1 Contents

Firstly, note that I do not provide the source code for SMPLify-x [28] and *redner* [19] since I do not claim their work as mine. I do use them substantially under the agreement to their licenses (see 7, Professional Issues).

The User Guide (see Appendix B), and the source code provided here below are sufficient to reproduce the methodology (see 4, Methodology).

For demonstration 3D models, source photos and experimentation results, explore my Github repository Golden Score at:

`https://github.com/MunkhtulgaB/golden-score`

See below for the source code of golden-score.

**golden-score**

| | |
|---|---|
| `Grabcut/grabcut.py` | Detects foreground in image |
| `gripping/grip.py` | Merges individual 3D models together |
| `redner/optimize_vertices.py` | Optimises vertices of a 3D model using *redner* |
| `vposer/optimize_pose.py` | Optimises pose of a 3D model using *redner* and *VPoser* |

## C.2 Grabcut/grabcut.py

```
#!/usr/bin/env python
'''
This file is slightly modified version of the Grabcut python sample file:
https://github.com/opencv/opencv/blob/master/samples/python/grabcut.py

I agree to their 3-clause BSD License and reproduce it below:
================================================================================
By downloading, copying, installing or using the software you agree to this license.
If you do not agree to this license, do not download, install,
copy or use the software.


                          License Agreement
                For Open Source Computer Vision Library
                        (3-clause BSD License)

Copyright (C) 2000-2020, Intel Corporation, all rights reserved.
Copyright (C) 2009-2011, Willow Garage Inc., all rights reserved.
Copyright (C) 2009-2016, NVIDIA Corporation, all rights reserved.
Copyright (C) 2010-2013, Advanced Micro Devices, Inc., all rights reserved.
Copyright (C) 2015-2016, OpenCV Foundation, all rights reserved.
Copyright (C) 2015-2016, Itseez Inc., all rights reserved.
Copyright (C) 2019-2020, Xperience AI, all rights reserved.
Third party copyrights are property of their respective owners.

Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are met:

  * Redistributions of source code must retain the above copyright notice,
    this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above copyright notice,
    this list of conditions and the following disclaimer in the documentation
    and/or other materials provided with the distribution.

  * Neither the names of the copyright holders nor the names of the contributors
    may be used to endorse or promote products derived from this software
    without specific prior written permission.

This software is provided by the copyright holders and contributors "as is" and
any express or implied warranties, including, but not limited to, the implied
warranties of merchantability and fitness for a particular purpose are disclaimed.
```

```
In no event shall copyright holders or contributors be liable for any direct,
indirect, incidental, special, exemplary, or consequential damages
(including, but not limited to, procurement of substitute goods or services;
loss of use, data, or profits; or business interruption) however caused
and on any theory of liability, whether in contract, strict liability,
or tort (including negligence or otherwise) arising in any way out of
the use of this software, even if advised of the possibility of such damage.
================================================================================
Interactive Image Segmentation using GrabCut algorithm.

This sample shows interactive image segmentation using grabcut algorithm.

USAGE:
    python grabcut.py <filename>

README FIRST:
    Two windows will show up, one for input and one for output.

    At first, in input window, draw a rectangle around the object using
mouse right button. Then press 'n' to segment the object (once or a few times)
For any finer touch-ups, you can press any of the keys below and draw lines on
the areas you want. Then again press 'n' for updating the output.

Key '0' - To select areas of sure background
Key '1' - To select areas of sure foreground
Key '2' - To select areas of probable background
Key '3' - To select areas of probable foreground

Key 'n' - To update the segmentation
Key 'r' - To reset the setup
Key 's' - To save the results
================================================================================
'''

# Python 2/3 compatibility
from __future__ import print_function

import numpy as np
import cv2 as cv

import sys

class App():
    BLUE = [255,0,0]        # rectangle color
```

```
RED = [0,0,255]         # PR BG
GREEN = [0,255,0]       # PR FG
BLACK = [0,0,0]         # sure BG
WHITE = [255,255,255]   # sure FG

DRAW_BG = {'color' : BLACK, 'val' : 0}
DRAW_FG = {'color' : WHITE, 'val' : 1}
DRAW_PR_FG = {'color' : GREEN, 'val' : 3}
DRAW_PR_BG = {'color' : RED, 'val' : 2}

# setting up flags
rect = (0,0,1,1)
drawing = False         # flag for drawing curves
rectangle = False       # flag for drawing rect
rect_over = False       # flag to check if rect drawn
rect_or_mask = 100      # flag for selecting rect or mask mode
value = DRAW_FG         # drawing initialized to FG
thickness = 3           # brush thickness

def onmouse(self, event, x, y, flags, param):
    # Draw Rectangle
    if event == cv.EVENT_LBUTTONDOWN:
        print("LButton down")
        if self.rect_over == False:
            self.rectangle = True
            self.ix, self.iy = x,y
        else:
            self.drawing = True
            cv.circle(self.img, (x,y), self.thickness, self.value['color'], -1)
            cv.circle(self.mask, (x,y), self.thickness, self.value['val'], -1)

    elif event == cv.EVENT_MOUSEMOVE:
        if self.rectangle == True:
            self.img = self.img2.copy()
            cv.rectangle(self.img, (self.ix, self.iy), (x, y), self.BLUE, 2)
            self.rect = (min(self.ix, x), min(self.iy, y), abs(self.ix - x), abs(self.iy - y))
            self.rect_or_mask = 0
        elif self.drawing == True:
            cv.circle(self.img, (x, y), self.thickness, self.value['color'], -1)
            cv.circle(self.mask, (x, y), self.thickness, self.value['val'], -1)

    elif event == cv.EVENT_LBUTTONUP:
        print("LButton up")
        if self.rectangle == True:
```

```
            self.rectangle = False
            self.rect_over = True
            cv.rectangle(self.img, (self.ix, self.iy), (x, y), self.BLUE, 2)
            self.rect = (min(self.ix, x), min(self.iy, y), abs(self.ix - x), abs(self.iy - y))
            self.rect_or_mask = 0
            print(" Now press the key 'n' a few times until no further change \n")
        elif self.drawing == True:
            self.drawing = False
            cv.circle(self.img, (x, y), self.thickness, self.value['color'], -1)
            cv.circle(self.mask, (x, y), self.thickness, self.value['val'], -1)

def run(self):
    # Loading images
    if len(sys.argv) == 2:
        filename = sys.argv[1] # for drawing purposes
    else:
        print("No input image given, so loading default image, lena.jpg \n")
        print("Correct Usage: python grabcut.py <filename> \n")
        filename = 'lena.jpg'

    self.img = cv.imread(cv.samples.findFile(filename))
    self.img2 = self.img.copy()                               # a copy of original image
    self.mask = np.zeros(self.img.shape[:2], dtype = np.uint8) # mask initialized to PR_BG
    self.output = np.zeros(self.img.shape, np.uint8)           # output image to be shown

    # input and output windows
    cv.namedWindow('output')
    cv.namedWindow('input')
    cv.setMouseCallback('input', self.onmouse)
    cv.moveWindow('input', self.img.shape[1]+10,90)

    print(" Instructions: \n")
    print(" Draw a rectangle around the object using right mouse button \n")

    while(1):

        cv.imshow('output', self.output)
        cv.imshow('input', self.img)
        k = cv.waitKey(1)

        # key bindings
        if k == 27:         # esc to exit
            break
        elif k == ord('0'): # BG drawing
```

```
        print(" mark background regions with left mouse button \n")
        self.value = self.DRAW_BG
    elif k == ord('1'): # FG drawing
        print(" mark foreground regions with left mouse button \n")
        self.value = self.DRAW_FG
    elif k == ord('2'): # PR_BG drawing
        self.value = self.DRAW_PR_BG
    elif k == ord('3'): # PR_FG drawing
        self.value = self.DRAW_PR_FG
    elif k == ord('s'): # save image
        bar = np.zeros((self.img.shape[0], 5, 3), np.uint8)
        res = np.hstack((bar, self.output))
        cv.imwrite('grabcut_output.png', res)
        print(" Result saved as image \n")
    elif k == ord('r'): # reset everything
        print("resetting \n")
        self.rect = (0,0,1,1)
        self.drawing = False
        self.rectangle = False
        self.rect_or_mask = 100
        self.rect_over = False
        self.value = self.DRAW_FG
        self.img = self.img2.copy()
        self.mask = np.zeros(self.img.shape[:2], dtype = np.uint8) # mask initialized to P
        self.output = np.zeros(self.img.shape, np.uint8)           # output image to be sh
    elif k == ord('n'): # segment the image
        print(""" For finer touchups, mark foreground and background after pressing keys 0
        and again press 'n' \n""")
        try:
            if (self.rect_or_mask == 0):         # grabcut with rect
                bgdmodel = np.zeros((1, 65), np.float64)
                fgdmodel = np.zeros((1, 65), np.float64)
                cv.grabCut(self.img2, self.mask, self.rect, bgdmodel, fgdmodel, 1, cv.GC_I
                self.rect_or_mask = 1
            elif self.rect_or_mask == 1:         # grabcut with mask
                bgdmodel = np.zeros((1, 65), np.float64)
                fgdmodel = np.zeros((1, 65), np.float64)
                cv.grabCut(self.img2, self.mask, self.rect, bgdmodel, fgdmodel, 1, cv.GC_I
        except:
            import traceback
            traceback.print_exc()

    mask2 = np.where((self.mask==1) + (self.mask==3), 255, 0).astype('uint8')
    self.output = cv.bitwise_and(self.img2, self.img2, mask=mask2)
```

```
        print('Done')


if __name__ == '__main__':
    print(__doc__)
    App().run()
    cv.destroyAllWindows()
```

## C.3 gripping/grip.py

```
import re
import sys
import argparse
import math

parser = argparse.ArgumentParser()
parser.add_argument(
'--tori',
'-t',
help=".obj file containing 3D model of tori",
type=str,
default="tori.obj"
)
parser.add_argument(
'--uke',
'-u',
help=".obj file containing 3D model of uke",
type=str,
default="uke.obj"
)
parser.add_argument(
'--output',
'-o',
help="name for file to write merged the model",
type=str,
default="grip.obj"
)

args = parser.parse_args()
print(args)

tori = open(args.tori, "r")
uke = open(args.uke, "r")

grip = open(args.output, "w")

# Read vertices and faces from tori file
tori_vertices = []
tori_faces = []
while True:
    nextLine = tori.readline()
    if not nextLine:
```

```
        break
    else:
     if nextLine.startswith('v'):
     tori_vertices.append(nextLine)
     elif nextLine.startswith('f'):
     tori_faces.append(nextLine)

# 1. Write the tori vertices
grip.writelines(tori_vertices)

def translate(orig, translation):
x, y, z = orig
dx, dy, dz = translation
return (x + dx, y + dy, z + dz)

def rotate(orig, pivot, angle, axis=0):
axes = [0, 1, 2]
axes.remove(axis)

a, b = (orig[ax] for ax in axes)
oa, ob = (pivot[ax] for ax in axes)

a1 = math.cos(angle) * (a-oa) - math.sin(angle) * (b-ob) + ob
b1 = math.sin(angle) * (a-oa) + math.cos(angle) * (b-ob) + ob

invariant = orig[axis]
to_return = [a1, b1]
to_return.insert(axis, invariant)
return to_return

translation = (-0.4, 0.05, 0.15)
pivot = (0,0,0)
rotation_angle = -math.pi/2
rotation_axis = 1

rotation_angle1 = math.pi/7
rotation_axis1 = 0

rotation_angle2 = 0
rotation_axis2 = 2

# Read vertices and faces from uke file
uke_vertices = []
uke_faces = []
```

```
while True:
    nextLine = uke.readline()
    if not nextLine:
        break
    else:
     if nextLine.startswith('v'):
     tokens = nextLine.rstrip().split(" ")[1:]
     x, y, z = [float(t) for t in tokens]



     tokens[0], tokens[1], tokens[2] = (str(c) for c in \
                translate(
                    rotate(
                        rotate(
                            rotate(
                                (x, y, z) ,
                                pivot,
                                rotation_angle,
                                rotation_axis
                            ),
                            pivot,
                            rotation_angle1,
                            rotation_axis1
                        ),
                        pivot,
                        rotation_angle2,
                        rotation_axis2
                    ),
                    translation
                )
           )
     new = " ".join(['v'] + tokens)

     uke_vertices.append(new + '\n')
     elif nextLine.startswith('f'):
     offset = len(tori_vertices)
     new = re.sub('(\d+)', lambda m: str(int(m.group(1)) + offset), nextLine)
     uke_faces.append(new)

# 2. Write uke vertices
# 3. Write tori faces
# 4. Write uke faces
grip.writelines(uke_vertices)
```

```
grip.writelines(tori_faces)
grip.writelines(uke_faces)

print('Gripped model written to:', args.output)
```

## C.4 redner/optimize_vertices.py

```
"""
This file is based on the redner tutorial:
https://github.com/BachiLi/redner/blob/master/tutorials/01_optimize_single_triangle.py

I agree to their MIT License and reproduce it below:
================================================================================

MIT License

Copyright (c) 2018 Tzu-Mao Li

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
================================================================================
"""


# The Python interface of redner is defined in the pyredner package
import pyredner
import torch
import matplotlib.pyplot as plt
import numpy as np

# Optimize three vertices of a single triangle
# We first render a target image using redner,
# then perturb the three vertices and optimize to match the target.
```

```
target_img = pyredner.imread('kouchi_photo_tori_small.jpg')
target = torch.tensor(target_img).double()
img_dim = (target.shape[0], target.shape[1])

objects = pyredner.load_obj('kouchi_tori.obj', return_objects=True)
objects = [objects[0]]



# objects[0].light_id = -1
# objects[0].material_id = 0

# We need to tell redner first whether we are using GPU or not.
pyredner.set_use_gpu(torch.cuda.is_available())

# Now we want to setup a 3D scene, represented in PyTorch tensors,
# then feed it into redner to render an image.

# First, we set up a camera.
# redner assumes all the camera variables live in CPU memory,
# so you should allocate torch tensors in CPU
cam = pyredner.Camera(position = torch.tensor([0.1, 0.20, 1.4]),
                      look_at = torch.tensor([0.1, 0.25, 0.0]),
                      up = torch.tensor([0.0, 1.0, 0.0]),
                      fov = torch.tensor([45.0]), # in degree
                      clip_near = 1e-2, # needs to > 0
                      resolution = img_dim,
                      fisheye = False)

# Next, we setup the materials for the scene.
# All materials in the scene are stored in a single Python list.
# The index of a material in the list is its material id.
# Our simple scene only has a single grey material with reflectance 0.5.
# If you are using GPU, make sure to copy the reflectance to GPU memory.
mat_grey = pyredner.Material(\
    diffuse_reflectance = \
        torch.tensor([0.5, 0.5, 0.5], device = pyredner.get_device()))
# The material list of the scene
materials = [mat_grey]

# Next, we setup the geometry for the scene.
# 3D objects in redner are called "Shape".
# All shapes in the scene are stored in a single Python list,
# the index of a shape in the list is its shape id.
```

```
# Right now, a shape is always a triangle mesh, which has a list of
# triangle vertices and a list of triangle indices.
# The vertices are a Nx3 torch double tensor,
# and the indices are a Mx3 torch integer tensor.
# Optionally, for each vertex you can specify its UV coordinate for texture mapping,
# and a normal for Phong interpolation.
# Each shape also needs to be assigned a material using material id,
# which is the index of the material in the material array.
# If you are using GPU, make sure to store all tensors of the shape in GPU memory.


shape_triangle = pyredner.Shape(\
    vertices = objects[0].vertices,
    indices = objects[0].indices,
    uvs = None,
    normals = None,
    material_id = 0)

# Merely having a single triangle is not enough for physically-based rendering.
# We need to have a light source. Here we setup the shape of a quad area light source,
# similary to the previous triangle.
shape_light = pyredner.Shape(\
    vertices = torch.tensor([[-1.0, -1.0, 3.],
                             [ 1.0, -1.0, 3.],
                             [-1.0,  1.0, 3.],
                             [ 1.0,  1.0, 3.]], device = pyredner.get_device()),
    indices = torch.tensor([[0, 1, 2],[1, 3, 2]],
        dtype = torch.int32, device = pyredner.get_device()),
    uvs = None,
    normals = None,
    material_id = 0)

shape_light1 = pyredner.Shape(\
    vertices = torch.tensor([[-1.0, -1.0, 7.0],
                             [ 1.0, -1.0, 7.0],
                             [-1.0,  1.0, 7.0],
                             [ 1.0,  1.0, 7.0]], device = pyredner.get_device()),
    indices = torch.tensor([[0, 1, 2],[1, 3, 2]],
        dtype = torch.int32, device = pyredner.get_device()),
    uvs = None,
    normals = None,
    material_id = 0)

shape_light2 = pyredner.Shape(\
```

```
    vertices = torch.tensor([[-1.0, -1.0, 6.0],
                             [ 1.0, -1.0, 6.0],
                             [-1.0,  1.0, 6.0],
                             [ 1.0,  1.0, 6.0]], device = pyredner.get_device()),
    indices = torch.tensor([[0, 1, 2],[1, 3, 2]],
        dtype = torch.int32, device = pyredner.get_device()),
    uvs = None,
    normals = None,
    material_id = 0)
# The shape list of our scene contains two shapes:
shapes = [shape_triangle, shape_light, shape_light1, shape_light2]

# Now we assign some of the shapes in the scene as light sources.
# All area light sources in the scene are stored in a single Python list.
# Each area light is attached to a shape using shape id, additionally we need to
# assign the intensity of the light, which is a length 3 double tensor in CPU.
light = pyredner.AreaLight(shape_id = 1,
                           intensity = torch.tensor([5.0,5.0,5.0]),
                           two_sided=True,
                           directly_visible=False)
light1 = pyredner.AreaLight(shape_id = 2,
                           intensity = torch.tensor([0.5,0.5,0.5]),
                           two_sided=True,
                           directly_visible=False)
light2 = pyredner.AreaLight(shape_id = 3,
                           intensity = torch.tensor([0.5,0.5,0.5]),
                           two_sided=True,
                           directly_visible=False)

area_lights = [light, light1, light2]
# Finally we construct our scene using all the variables we setup previously.
scene = pyredner.Scene(cam, shapes, materials, area_lights)
# All PyTorch functions take a flat array of PyTorch tensors as input,
# therefore we need to serialize the scene into an array. The following
# function does this. We also specify how many Monte Carlo samples we want to
# use per pixel and the number of bounces for indirect illumination here
# (one bounce means only direct illumination).

scene_args = pyredner.RenderFunction.serialize_scene(\
    scene = scene,
    num_samples = 20,
    max_bounces = 1)

# Now we render the scene as our target image.
```

```
# To render the scene, we use our custom PyTorch function in pyredner/render_pytorch.py
# First setup the alias of the render function
render = pyredner.RenderFunction.apply
# # Next we call the render function to render.
# # The first argument is the seed for RNG in the renderer.
# img = render(0, *scene_args)
img = target

# This generates a PyTorch tensor with size [width, height, 3].
# The output image is in the GPU memory if you are using GPU.
# Now we save the generated image to disk.
pyredner.imwrite(img.cpu(), 'results/optimize_vertices_kouchi_tori_obj_exp0/target.exr')
pyredner.imwrite(img.cpu(), 'results/optimize_vertices_kouchi_tori_obj_exp0/target.png')
# Now we read back the target image we just saved, and copy to GPU if necessary
target = pyredner.imread('results/optimize_vertices_kouchi_tori_obj_exp0/target.exr').double()
if pyredner.get_use_gpu():
    target = target.cuda()

# Next we want to produce the initial guess. We do this by perturb the scene.
shape_triangle.vertices = torch.tensor(\
    objects[0].vertices,
    device = pyredner.get_device(),
    requires_grad = True) # Set requires_grad to True since we want to optimize this



# We need to serialize the scene again to get the new arguments.
scene_args = pyredner.RenderFunction.serialize_scene(\
    scene = scene,
    num_samples = 20,
    max_bounces = 1)
# Render the initial guess
img = render(1, *scene_args).double()
# Save the image
pyredner.imwrite(img.cpu(), 'results/optimize_vertices_kouchi_tori_obj_exp0/init.png')

# Compute the difference and save the images.
diff = torch.abs(target - img)
print(target)
print(img)
print(diff)

pyredner.imwrite(diff.cpu(), 'results/optimize_vertices_kouchi_tori_obj_exp0/init_diff.png')
input('wait')
```

```
# Now we want to refine the initial guess using gradient-based optimization.
# We use PyTorch's optimizer to do this.
shape_triangle.vertices.requires_grad = False
to_opt = torch.split(shape_triangle.vertices, [10, 20, 10445])
print(to_opt)
to_opt[0].requires_grad = True
optimizer = torch.optim.Adam([to_opt[0]], lr=1e-3)

# Run 200 Adam iterations.
for t in range(50):
    print('iteration:', t)
    optimizer.zero_grad()
    # Forward pass: render the image
    scene_args = pyredner.RenderFunction.serialize_scene(\
        scene = scene,
        num_samples = 4, # We use less samples in the Adam loop.
        max_bounces = 1)
    # Important to use a different seed every iteration, otherwise the result
    # would be biased.
    img = render(t+1, *scene_args).double()
    # Save the intermediate render.
    pyredner.imwrite(img.cpu(), 'results/optimize_vertices_kouchi_tori_obj_exp0/iter_{}.png'.forma
    # Compute the loss function. Here it is L2.

    diff = (img - target).pow(2)
    max_diff = diff.max()
    normalized = (diff / max_diff)
    loss = normalized.sum()
    # print('loss:', loss.item())

    # Backpropagate the gradients.
    loss.backward()
    # Print the gradients of the three vertices.
    # print('grad:', shape_triangle.vertices.grad)

    # Take a gradient descent step.
    optimizer.step()
    # Print the current three vertices.

# Render the final result.
scene_args = pyredner.RenderFunction.serialize_scene(\
    scene = scene,
    num_samples = 16,
```

```
    max_bounces = 1)
img = render(202, *scene_args).double()

result_obj = shapes[0]
result_obj.vertices = result_obj.vertices.detach()
pyredner.save_obj(result_obj, f'results/optimize_vertices_kouchi_tori_obj_exp0/result.obj')

# Save the images and differences.
pyredner.imwrite(img.cpu(), 'results/optimize_vertices_kouchi_tori_obj_exp0/final.exr')
pyredner.imwrite(img.cpu(), 'results/optimize_vertices_kouchi_tori_obj_exp0/final.png')
pyredner.imwrite(torch.abs(target - img).cpu(), 'results/optimize_vertices_kouchi_tori_obj_exp0/fi

# Convert the intermediate renderings to a video.
from subprocess import call
call(["ffmpeg", "-framerate", "24", "-i",
    "results/optimize_vertices_kouchi_tori_obj_exp0/iter_%d.png", "-vb", "20M",
    "results/optimize_vertices_kouchi_tori_obj_exp0/out.mp4"])
```

## C.5 vposer/optimize_pose.py

```
"""
This file is based on the redner tutorial:
https://github.com/BachiLi/redner/blob/master/tutorials/01_optimize_single_triangle.py
I agree to their MIT License and reproduce it below:
================================================================================
MIT License
Copyright (c) 2018 Tzu-Mao Li
Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:
The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
================================================================================
This file makes use of SMPLify-x:

@inproceedings{SMPL-X:2019,
  title = {Expressive Body Capture: 3D Hands, Face, and Body from a Single Image},
  author = {Pavlakos, Georgios and Choutas, Vasileios and Ghorbani, Nima and Bolkart, Timo and Osm
  booktitle = {Proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)},
  year = {2019}
}

By License agreement, I do not copy their Model & Software and the license. See more on:
https://github.com/vchoutas/smplify-x/blob/master/LICENSE

"""

import pyredner
import torch
import matplotlib.pyplot as plt
import numpy as np
import pickle
```

```
from human_body_prior.body_model.body_model_vposer import BodyModelWithPoser
from scipy.ndimage import gaussian_filter
from human_body_prior.tools.model_loader import load_vposer as poser_loader

NUM_ITERS = 50
MODEL_NAME = 'uke_osotogari_filled'

# Get the body model
expr_dir = '../SMPL-X/vposer_v1_0' #'TRAINED_MODEL_DIRECTORY'  in this directory the trained model
bm_path =  '../SMPL-X/models/smplx/SMPLX_MALE.npz'#'PATH_TO_SMPLX_model.npz'  obtain from https://
mano_exp_dir = expr_dir

bm = BodyModelWithPoser(bm_path=bm_path, batch_size=1, poser_type='vposer', smpl_exp_dir=expr_dir)

# Start from an approximation loaded from pickled SMPLify-x output
try:
body_model_path = 'results/' + MODEL_NAME + '/poZ_body.pkl'
print('Loading pose from:', body_model_path)
body_pickle = open(body_model_path, 'rb')
body = pickle.load(body_pickle)
pose_body = body.get('body_pose')
except:
print('No existing pose found starting from original')
body_model_path = 'input/source/' + MODEL_NAME + '/'
body_pickle = open(body_model_path + '000.pkl', 'rb')
body = pickle.load(body_pickle)

# hand left
bm.poser_handL_pt, bm.poser_handL_ps = poser_loader(mano_exp_dir)
bm.poser_handL_pt.to(bm.trans.device)

poZ_handL = bm.pose_hand.new(body.get('left_hand_pose'))
bm.register_parameter('poZ_handL', torch.nn.Parameter(poZ_handL, requires_grad=True))

    # hand right
bm.poser_handR_pt, bm.poser_handR_ps = poser_loader(mano_exp_dir)
bm.poser_handR_pt.to(bm.trans.device)

poZ_handR = bm.pose_hand.new(body.get('right_hand_pose'))
bm.register_parameter('poZ_handR', torch.nn.Parameter(poZ_handR, requires_grad=True))
bm.pose_hand.requires_grad = False

pose_body = torch.tensor(body.get('body_pose')[0])
```

```
random_modification = torch.empty(pose_body.shape).normal_(mean=1,std=0.01)
poZ_body = torch.tensor(pose_body * random_modification, requires_grad=True)
bm.register_parameter('poZ_body', torch.nn.Parameter(poZ_body, requires_grad=True))


# Prepare to render
target_img = pyredner.imread('input/targets/' + MODEL_NAME + '.png')
# target_img = gaussian_filter(target_img, sigma=10)
target = torch.tensor(target_img).double()
target *= 20
pyredner.imwrite(target.cpu(), 'results/' + MODEL_NAME + '/target.png')

img_dim = (target.shape[0], target.shape[1])

cam_pos = torch.tensor([2.0, -0.3, 1.3], requires_grad=True)
cam = pyredner.Camera(position = cam_pos,
                      look_at = torch.tensor([-0.3, -0.6, 0.0]),
                      up = torch.tensor([0.0, 1.0, 0.0]),
                      fov = torch.tensor([45.0]), # in degree
                      clip_near = 1e-2, # needs to > 0
                      resolution = img_dim,
                      fisheye = False)

mat_grey = pyredner.Material(\
    specular_reflectance = \
        torch.tensor([2.8, 2.8, 2.8], device = pyredner.get_device()))
# The material list of the scene
materials = [mat_grey]


shape_light = pyredner.Shape(\
    vertices = torch.tensor([[3.0, -1.0, -1.],
                             [3.0, -1.0, 2.],
                             [3.0,  1.0, -1.],
                             [3.0,  1.0, 2.]], device = pyredner.get_device()),
    indices = torch.tensor([[0, 1, 2],[1, 3, 2]],
        dtype = torch.int32, device = pyredner.get_device()),
    uvs = None,
    normals = None,
    material_id = 0)

shape_light1 = pyredner.Shape(\
    vertices = torch.tensor([[-2.0, -1.0, -4.],
```

```
                            [ 0.0, -1.0, -4.],
                            [-2.0,  1.0, -4.],
                            [ 0.0,  1.0, -4.]], device = pyredner.get_device()),
    indices = torch.tensor([[0, 1, 2],[1, 3, 2]],
        dtype = torch.int32, device = pyredner.get_device()),
    uvs = None,
    normals = None,
    material_id = 0)


light = pyredner.AreaLight(shape_id = 1,
                           intensity = torch.tensor([6.0,6.0,6.0]),
                           two_sided=True,
                           directly_visible=False)

light1 = pyredner.AreaLight(shape_id = 2,
                           intensity = torch.tensor([4.0,4.0,4.0]),
                           two_sided=True,
                           directly_visible=False)

vertices = bm.forward().v[0]
indices = bm.f
shape_triangle = pyredner.Shape(
vertices = vertices,
indices = bm.f,
material_id=0)


shapes = [shape_triangle, shape_light, shape_light1]
area_lights = [light, light1]
scene = pyredner.Scene(cam, shapes, materials, area_lights)

# Optimise
# optimizer = torch.optim.SGD([bm.poZ_body], lr=1, momentum=0.9)
optimizer = torch.optim.Adam([bm.poZ_body], lr=5e-2)
# optimizer = torch.optim.Adam([cam.position], lr=1e-2)
orig_poZ = bm.poZ_body.clone()

render = pyredner.RenderFunction.apply
for t in range(NUM_ITERS):
    print('iteration:', t)
    optimizer.zero_grad()
    # Forward pass: render the image
```

```
    shape_triangle.vertices = bm.forward().v[0]
    scene = pyredner.Scene(cam, shapes, materials, area_lights)
    scene_args = pyredner.RenderFunction.serialize_scene(\
        scene = scene,
        num_samples = 4 , # We use less samples in the Adam loop.
        max_bounces = 1)
    img = render(t+1, *scene_args).double()
    # print(img)

    # Save the intermediate render.
    # if t % 10 == 0:
    pyredner.imwrite(img.cpu(), 'results/' + MODEL_NAME + '/iter_{}.png'.format(t))
    # Compute the loss function. Here it is L2.
    loss = (img - target).pow(2).mean()
    # diff = orig_poZ.double() - bm.poZ_body.double()
    # loss += (diff.pow(2).mean())
    loss.backward()
    print('Loss:', loss.item())
    # Take a gradient descent step.
    optimizer.step()

    # Pickle the poZ_body
    pickle_out = open('results/' + MODEL_NAME + '/poZ_body.pkl',"wb")
    pickle.dump(dict(body_pose=bm.poZ_body), pickle_out)
    pickle_out.close()

    # Save the resulting 3D model in .obj format
    result_obj = shapes[0]
    result_obj.vertices = result_obj.vertices.detach()
    pyredner.save_obj(result_obj, f'results/' + MODEL_NAME + '/result.obj')

# Save the images and differences.
pyredner.imwrite(img.cpu(), 'results/' + MODEL_NAME + '/final.exr')
pyredner.imwrite(img.cpu(), 'results/' + MODEL_NAME + '/final.png')
pyredner.imwrite(torch.abs(target - img).cpu(), 'results/' + MODEL_NAME + '/final_diff.png')

# Convert the intermediate renderings to a video.
from subprocess import call
call(['ffmpeg', '-framerate', '24', '-i',
    'results/' + MODEL_NAME + '/iter_%d.png', '-vb', '20M',
    'results/' + MODEL_NAME + '/out.mp4'])
```