Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

DIPLOMA THESIS ASSIGNMENT

Student:	Bc. Tomáš K r u p k a
Study programme:	Open Informatics
Specialisation:	Computer Vision and Image Processing
Title of Diploma Thesis:	Motion Capture using Sum of Gaussians Body Model

Guidelines:

Based on the implementation of algorithm [1] originally done as a software project propose an extension which allows to use the algorithm also in cases when lighting conditions change during the capture. Take inspiration from a recent technique presented in [2].
In addition try to optimize the algorithm in a way that real-time markerless motion capture of full human body will be possible. Verify the robustness of resulting implementation in more complex scenarios such as interaction of two persons or motion capture of animals.

Bibliography/Sources:

[1] Stoll et al.: Fast Articulated Motion Tracking using a Sums of Gaussians Body Model, Proceedings of IEEE International Conference on Computer Vision, pp. 951-958, 2011.
[2] Li et al.: Capturing Relightable Human Performances under General Uncontrolled Illumination, Computer Graphics Forum 32(2):275-284, 2013.

Diploma Thesis Supervisor: doc. Ing. Daniel Sýkora, Ph.D.

Valid until: the end of the summer semester of academic year 2015/2016

L.S.

doc. Dr. Ing. Jan Kybic	prof. Ing. Pavel Ripka, CSc.
Head of Department	**Dean**

Prague, January 21, 2015

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student:	Bc. Tomáš K r u p k a
Studijní program:	Otevřená informatika (magisterský)
Obor:	Počítačové vidění a digitální obraz
Název tématu:	Snímání pohybu pomocí Gaussovských směsí

Pokyny pro vypracování:

Vycházejte z existující implementace algoritmu [1] provedené v rámci softwarového projektu a navrhněte rozšíření, které by umožnilo použít algoritmus i v případě měnících se světelných podmínek. Inspirujte se technikami použitými v článku [2].
Dále se pokuste algoritmus optimalizovat natolik, že bude možné jej použít i pro snímání pohybů lidského těla v reálném čase. Výslednou implementaci otestujte na složitějších scénách obsahujících například interakci více herců nebo pohybující se zvířata.

Seznam odborné literatury:

[1] Stoll et al.: Fast Articulated Motion Tracking using a Sums of Gaussians Body Model, Proceedings of IEEE International Conference on Computer Vision, pp. 951-958, 2011.
[2] Li et al.: Capturing Relightable Human Performances under General Uncontrolled Illumination, Computer Graphics Forum 32(2):275-284, 2013.

Vedoucí diplomové práce: doc. Ing. Daniel Sýkora, Ph.D.

Platnost zadání: do konce letního semestru 2015/2016

L.S.

doc. Dr. Ing. Jan Kybic	prof. Ing. Pavel Ripka, CSc.
vedoucí katedry	**děkan**

V Praze dne 21. 1. 2015

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne ....

Podpis autora práce

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

Master's Thesis

Motion Capture using Sum of Gaussians Body Model

Bc. Tomáš Krupka

Supervisor: doc. Ing. Daniel Sýkora, Ph.D.

Study Programme: Open Informatics

Field of Study: Computer Vision and Image Processing

May 11, 2015

Aknowledgements

I would like to thank my family and friends for supporting me through the time of writing and my supervisor, Daniel Sýkora, for helping me tremendously throughout the whole thesis writing process.

Abstract

The goal of this work is to improve on the work done previously in a markerless motion capture project called Kostilam. The main goals of the thesis assignment were to optimize the algorithm as close to realtime performance as possible and to modify the approach to be able to adapt to light variations. We have managed to optimize the algorithm through the use of sparse vectors, custom memory management and parallelization. On the other hand the light adaptation was not a success and is perhaps a topic for some future work.

Keywords: *Markerless motion capture, Automatic differentiation, Sum of Gaussians body model, performance capture*

Abstrakt

Cílem této práce je pokračovat na práci započaté při semestrálním projektu na systému bezznačkového snímání pohybu jménem Kostilam. Hlavní části zadání tvoří za prvé optimalizace snímacího algoritmu tak, aby se co nejvíce přiblížil zpracování snímku za reálného času, a za druhé úprava algoritmu takovým způsobem, aby byl schopen se přizpůsobit měnícím se světelným podmínkám. Optimalizace snímacího algoritmu se nám zdařila pomocí implementace řídkého vektoru, vlastního alokátoru paměti a paralelizace. Na druhou stranu se světelnou adaptací jsme neměli úspěch a ta tak zbývá jako námět k příští práci.

Klíčová slova: *motion capture bez značek, automatická derivace, suma gausiánů jako model člověka, zachycení pohybu*

Contents

1 Introduction **1**
1.1 Related work 1
1.2 Thesis structure 2

2 Motion capture cost function **3**
2.1 Sum of Gaussians body model 3
2.1.1 Positional model 4
2.1.2 Angular model 4
2.1.3 Rotation parameters 5
2.1.4 Model summary 6
2.2 Image model 7
2.3 Similarity measure 8
2.3.1 Model projection 8
2.3.2 Blob similarity 9
2.3.3 Model similarity 10
2.4 Cost function 10
2.4.1 Note about tracking 11

3 Cost minimization **13**
3.1 Automatic differentiation 14
3.1.1 Generalization 15
3.1.2 Operations over dual objects 15
3.1.3 Multidimensional differentiation 16
3.1.4 Example 17
3.2 Gradient descent 18

4 Implementation overview **21**
4.1 Overview 21
4.2 Initial solution 23
4.2.1 Inverse kinematics 23
4.3 Fitting a body model 24
4.3.1 Multi-pose cost function 24
4.4 Color calibration 26
4.5 Creating an image model 27

5 Implementation details **29**
5.1 Automatic differentiation . . . 29
5.2 Dual objects . . . 30
5.2.1 Operations with Dual objects . . . 31
5.2.2 Generic math primitives . . . 33
5.3 Angular model representation . . . 34
5.4 Inverse kinematics . . . 35
5.4.1 Applying automatic differentiation . . . 37
5.4.2 Dual objects initialization . . . 38
5.5 Motion tracking . . . 38
5.5.1 Summary . . . 40
5.6 Optimization techniques . . . 41
5.6.1 Sparse vector . . . 42
5.6.2 Memory management . . . 43
5.6.2.1 Fixed-sized pool allocator . . . 44
5.6.3 Parallelization . . . 46
5.6.4 Summary . . . 46

6 Testing **47**
6.1 Various sequences . . . 47
6.1.1 Sequence blue_exercise . . . 47
6.1.2 Sequence red_exercise . . . 47
6.1.3 Sequence take1 . . . 47
6.1.4 Sequence take3 . . . 47

7 Conclusion **57**

Bibliography **59**

A Figures **63**

B Contents of CD **67**

List of Figures

2.1 Original human body model from Kostilam. 4
2.2 Angular model 5
2.3 Image decomposition 7
2.4 Projection of the model to 6 camera images 9

4.1 Kostilam overview 22
4.2 Finding the initial solution 23
4.3 Fitting a body model to an actor I 25
4.4 Fitting a body model to an actor II 26
4.5 Color calibration of the model 27
4.6 Image model with background subtraction 28

5.1 Evaluating a function using Dual objects $[a, \mathbf{b}]$ as the arguments. 32
5.2 Data types for the representation of the angular model 34
5.3 Structure of the vector of parameters x 35
5.4 Parameters influencing a blob position. 41
5.5 Size of `Dual` objects in calculations 42
5.6 `SparseVector` representation 43
5.7 Memory manager initialization 45
5.8 Memory manager allocation 45
5.9 Memory manager deallocation 45

6.1 Frame 2091 of the blue_exercise sequence, model view 48
6.2 Frame 1078 of the blue_exercise sequence, model view 48
6.3 Frame 2091 of the blue_exercise sequence 49
6.4 Frame 1078 of the blue_exercise sequence 49
6.5 Frame 672 of the red_exercise sequence, model view 50
6.6 Frame 1300 of the red_exercise sequence, model view 50
6.7 Frame 672 of the red_exercise sequence 51
6.8 Frame 1300 of the red_exercise sequence 51
6.9 Frame 390 of the take1 sequence, model view 52
6.10 Frame 913 of the take1 sequence, model view 52
6.11 Frame 390 of the take1 sequence 53
6.12 Frame 913 of the take1 sequence 53
6.13 Frame 674 of the take3 sequence, model view 54
6.14 Frame 824 of the take3 sequence, model view 54

6.15 Frame 674 of the take3 sequence 55
6.16 Frame 824 of the take3 sequence 55

A.1 Point clouds and skeletons 63
A.2 IIM studio 64
A.3 IIM studio, the dog experiment 64
A.4 take3 exported to Maya, courtesy of Radek Smetana 65
A.5 take11 exported to Maya, courtesy of Radek Smetana 66

Listings

5.1 C++ function using floats 30
5.2 A templated C++ function 30
5.3 Computing the gradient of $g(x, y)$ by using `Dual` objects 30
5.4 Definition of the `Dual` object 31
5.5 Multiplication operator with `Dual` objects as arguments 32
5.6 Examples of various operations with `Dual` objects as arguments 32
5.7 A helper object for computing the inverse kinematics cost function 35
5.8 Code calculating the inverse kinematics cost function 36
5.9 Code calculating the joints' reference frames 36
5.10 Evaluation using `floats`, `result` $= f(x)$ 37
5.11 Automatic differentiation, `result.a` $= f(x)$, `result.b` $= \nabla f(x)$ 37
5.12 A helper object for computing the motion tracking cost function 38
5.13 Multiplication operator with `Dual` objects as arguments 44
5.14 Custom memory manager class inspired by [31] 44
5.15 OpenMP enabled parallelism 46

Chapter 1

Introduction

Over the last few decades, *motion capture* has become a widely used term in many branches of work [8], especially with the ever larger presence of digital media technology in our lives. Motion capture is an act of recording an actor's performance, usually resulting in a 3D sequence which we can use to replay all the the motions exactly as the happened. Since we have full 3D information about the movements, we can watch the resulting animation from any angle. Motion capture typically refers to taking animations of humans, but the method can also be used to capture the movements of animals.

There are several reasons why someone would like to capture human or animal motion to a digital form. According to a survey [5], the fields where motion capture is used range from virtual reality - film production and computer games - continues with motion analysis for clinical studies or surveillance where it is used to recognize people by their gait - and as a last example we will give special user interfaces e.g. for sign language interpretation or gesture driven applications.

The traditional motion capture technologies are *marker-based*; they involve the actors having to change their clothes for specially crafted black suits which have distinctive white markers attached. Such markers are then tracked using a multiple camera setup giving us in the end a 3D reconstruction of the performed motion. Such approaches are already well developed today, the user can choose among several commercial solutions e.g. OptiTrack [24] or Qualysis [28].

Markerless motion capture on the other hand, is a relatively new trend. It tries to capture the movements just by analyzing usual camera images, no special suits should be necessary. Markerless motion capture is the central topic of this thesis, all the presented work will be done on our custom markerless motion capture software based on the ideas from Stoll [33]. To give some context, let us take a look at some related work.

1.1 Related work

Even though we said the trend of markerless motion capture is relatively new, it has actually been studied extensively in the past, e.g. the survey [5] from 2001 references over 40 other articles on the topic. A newer survey [27] from 2007 goes further and references over 100 other articles.

The most notable difference between the various approaches seems to lie in the way they use the body model. The survey [27] describes two ways; in the first case the algorithms use some *a priori* body model, typically consisting of a hierarchy of joints with varying *degrees of freedom.*

Those further divide into *top-down* or *bottom-up* categories. In bottom-up the algorithm tries to find individual parts of bodies in the camera image and construct from them a complete model, e.g. [22]. The top-down methods on the other hand try to find the position of the model in space, so that it projects to the actor in the image image, e.g. [14].

The second case are called *model-free* algorithms and that's because they don't assume any information about the model beforehand. Such methods either try to fit the model on the go using maximum *a posteriori* esimator, e.g. [16], or they work based on an existing database of poses such as [38].

All the presented works have been part of a long line of research dating back to the 80s and continuing through to the beginning of the century. Later, however, we have started to begin seeing approaches aimed solely on fast computation, as it became possible to achieve realtime speeds on new machinesm, e.g. Michoud [21], Stoll [33] or Wei [39].

Stoll [33] came with a novel approach for markerless motion capture involving usage of a simplified body model made up of 3D Gaussians. By the above definition, it is a top-down method, yet its formulation is relatively simple and promises realtime performance. Those were good reasons for this method to be used in our software as well.

1.2 Thesis structure

Following will be 4 key chapters of this thesis. In the chapter named "Motion capture cost function" we will present the core idea behind the motion capture algorithm as it was developed in Stoll [33], in the end we will present a cost function that will be used to determine the quality of a given body pose. In the next chapter, called "Cost minimization", we will show how we approach minimization of this cost function.

After that, in chapter named "Implementation overview", we will describe the complete workings of our motion capture software, where the above mentioned cost function is just one of the pieces in the puzzle. Next, in the chapter "Implementation details", we will talk about detailed workings of our motion capture algorithm plus we will show some optimization techniques we used to make the program run as fast as possible. Finally in the chapter "Testing," we will show what results we achieved with our implementation.

Chapter 2

Motion capture cost function

We chose to implement a markerless motion capture system based on the ideas introduced by Stoll in 2011 [33]. In this section we will give a theoretical overview of our problem, describe the choices we made in creating a model representing the tracked subject and finally we will make our way to defining a cost function that will be at the core of the tracking algorithm. Owing to the difficulties in the early beginnings the project has earned its name characteristically - *Kostilam*.

As mentioned in the introduction, this method involves creating a model for the tracked subject consisting of a simplified skeleton and 3D Gaussians attached to its joints. We will describe in this section how we built the model for a human body, however the approach can be modified for other subjects as well, namely what we have experimented with was a model for a dog body.

2.1 Sum of Gaussians body model

In developing our own human body model we have taken inspiration mainly in [33], also in [10], and in an atlas of biology to establish some reasonable proportions. In the simplest sense the body is modeled as a sum of unnormalized uncorrelated Gaussians. This approach is used both in the image domain where the Gaussians are two-dimensional and in the 3D space where the Gaussians have three dimensions. In the following definition $\mu \in \mathbb{R}^d$ stands for the mean, with d being either 2 or 3, σ^2 stand for the variance. The building blocks for the model are the following Gaussians:

$$\mathcal{B}(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x}-\mu\|_2^2}{2\sigma^2}\right) \tag{2.1}$$

A model $\mathcal{K}$ is defined as a sum of such Gaussians:

$$\mathcal{K}(\mathbf{x}) = \sum_{i=1}^{n} \mathcal{B}_i(\mathbf{x}) \tag{2.2}$$

As in [33] we store the color information for each Gaussian $\mathcal{B}_i$ in a separate variable $\mathbf{c}_i \in \mathbb{R}^3$, the Gaussians themselves only represent the volume of the model.

This definition doesn't reveal the underlying structure of the model very well. The placement of the individual Gaussians in space and their possible interconnections have to be defined in addition to this.

2.1.1 Positional model

In the original version of Kostilam the model was defined using a simple skeleton consisting of joints (visible in fig. 2.1) and links representing bones. The position for the Gaussians is defined to be a convex combination of two joints; to define the position three parameters are needed - 2 joints and a coefficient $\alpha \in \langle 0, 1 \rangle$.

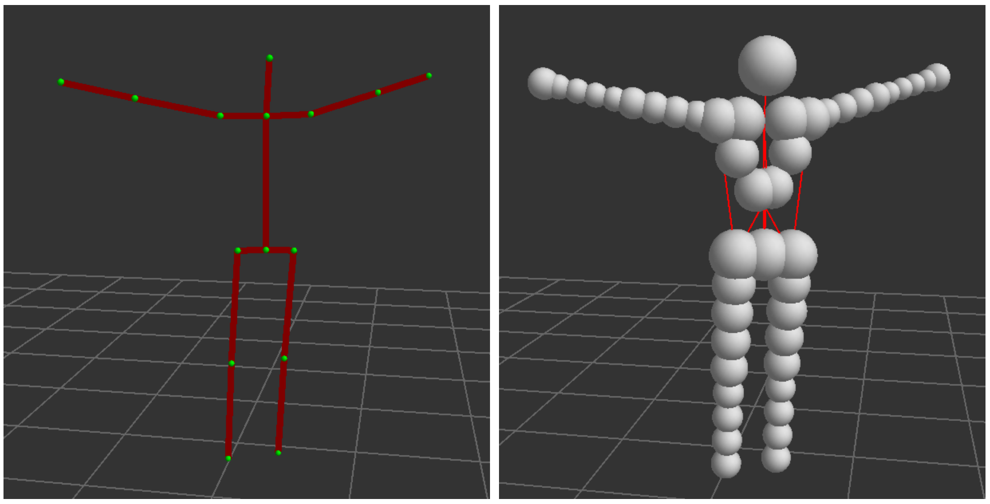


Figure 2.1: Original model from Kostilam. Skeleton structure with joints in green (left), Spheres representing the model's Gaussians, they are drawn with a radius of σ (right)

The parameters which are to be optimized for are the positions of the joints, given that there are 15 joints, this results in an optimization problem with 45 variables. Unfortunately, a model defined in this way poses some problems. Given the way the optimization process will be defined, it is quite hard to impose further restrictions on the positions of joints.

The most important one being that joints which are connected by a bone shouldn't change their length during the tracking. This was solved in the original implementation by using a regularization step based on ARAP similarity [36] with a default model in-between following optimizations.

Another problem is that it is difficult with this model to impose restrictions on some relationships between adjacent bones, which would be helpful to limit the occurrence of certain unnatural movements e.g. the shin or wrist bending in the opposite direction than they should be.

2.1.2 Angular model

A key part of this work was to introduce a model more alike to the one defined in [33] which would use rotations of the joints as the new parameters, leaving the bone lengths fixed.

That means the ARAP regularization step won't be necessary anymore. Such a model will also allow to make limits on the rotations themselves thus leading to a more rigid model, not susceptible to some of the problems of the original model such as unnatural looking movements.

We named the new model the *angular model* as it relies mainly on rotations as its parameters unlike the older model which used only translations. Each joint in angular model has two important parameters, a fixed offset which represents the length and direction of the bone leading to this joint, and a set of angles which are used to rotate the relevant bone. Each joint has a defined parent and together the joints form a hierarchy to model the human skeleton. This approach is common in animation software packages like Maya [3] or [2].

As is the case with the older model, Gaussians are attached to the joints of the angular model to represent its volume in space. This time each of the Gaussians has its location defined by a three dimensional offset to a given joint which allows us to place the Gaussians in arbitrary positions, not necessarily on a connection of joints. This is used e.g. to model the human torso and could potentially be used to model more complicated scenarios.

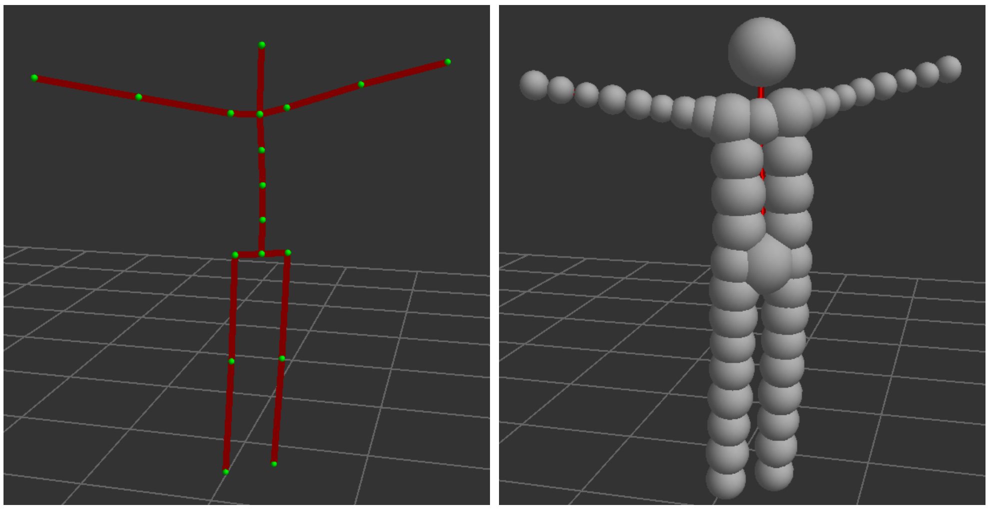


Figure 2.2: Angular model. Skeleton structure with joints in green (left), Spheres representing the model's Gaussians, they are drawn with a radius of σ (right)

The angular model differs with the previous model in that it has a spine modeled from several shorter bones (fig. 2.2) to allow for more natural bending animations. The pelvis joint was chosen to be the root of the skeleton's hierarchy, the position and rotation of the remaining joints is always relative to this root joint.

2.1.3 Rotation parameters

A more detailed description of how to calculate the precise position of the joints in space will follow. Before doing that however, let us first detail the choices we made for the representation of the rotations as this is an important step in many optimization problems. In general in optimization we want to keep the number of *degrees of freedom* to a minimum, in the case of 3D rotations, it should be 3 degrees of freedom. Introducing more degrees of freedom may lead to poor performance or even inability to find a good solution at all.

We have considered the *unit quaternion* and *axis-angle* representations for use in angular model. Both of these come with the restriction of unitarity, in the first case the quaternion has to be unitary to represent a rotation, in the second case the axis vector should be unitary. Unfortunately, that is a difficult constraint to satisfy in typical optimization solvers. Lot of work has been spent in literature to keep this constraint implicitly e.g. [30], however, the proposed solutions bring unnecessary overhead to the problem which is in conflict with our demands on fast performance.

Also it is not straightforward to come up with meaningful constraints on rotations in either of these representations, although some work has been done on this [4]. In the end we chose standard *Euler angles* for the rotation representation which allowed us to keep the number of degrees of freedom to a minimum, even going below 3 for specific joints. For example the knee and ankle joints have just 1 degree of freedom in our model.

We enforce limits on the values of the angle parameters to disallow unnatural movements, this also helps with keeping the problem of *gimbal lock* at bay. We have worried about it's possible occurrence, especially with shoulder joints, but in our later testing we didn't encounter it while tracking real-life videos.

2.1.4 Model summary

Here we will give a concise overview of the angular model parameters and then we will show how these parameters are used to compute the position of the model in space. This is helpful later when we introduce the final cost function further down this section 2.15 to see its full complexity.

- Joint $\mathcal{J}_j$:
  1. parent joint $p(j)$
  2. offset $\mathbf{s}_j \in \mathbb{R}^3$
  3. angles $\alpha_j \in \mathbb{R}^d, d \in \{0, 1, 2, 3\}$, the rotation matrix is given by $\mathbf{R}(\alpha_j)$
- Gaussian (for simplicity called *blob*) $\mathcal{B}_i$:
  1. joint to attach to $a(\mathcal{B}_i)$
  2. offset $\mathbf{w}_i \in \mathbb{R}^3$
  3. variance σ^2

First we will establish a transformation from something we call the *joint's reference frame* to the *world coordinates*, because the offsets w_i are taken relative to the joints rotation and translation. For a joint j the transformation will consist of a rotation by matrix $\mathbf{R}_j$ followed by a translation by vector $\mathbf{t}_j$:

$$\mathbf{R}_j = \mathbf{R}_{p(j)}\mathbf{R}(\alpha_j) \tag{2.3}$$

$$\mathbf{t}_j = \mathbf{t}_{p(j)} + \mathbf{R}_j\mathbf{w}_j \tag{2.4}$$

Given such a transformation, we can now compute the position μ_i of the Gaussian blobs attached to joint j, that is blobs for which $a(\mathcal{B}_i) = j$:

$$\mu_i = \mathbf{t}_j + \mathbf{R}_j \mathbf{w}_i \tag{2.5}$$

We have prepared a set of parameters to define a default human model to be used for tracking, on top that we have developed a way of fitting the model semi-automatically to an individual actor. This process involves using the final cost function (2.15) so we will describe it later in chapter 4.

2.2 Image model

For the purpose of defining a suitable similarity measure between the model and images from a camera, we introduce an approximation of the images in exactly the same form as the model itself - a sum of Gaussians (2.2). The difference this time is that the Gaussians are two-dimensional. This idea is taken from [33], where two methods of decomposing the image to Gaussian blobs are presented.

First method involves using a *quad-tree* structure to form an irregular grid of blobs based on color similarity. Some blobs will cover a larger area of similar color, while other may be small when there is lot of color variation in the image. The second method is simpler and it works by forming a uniform grid over the image and creating a blob in each cell with the average color of the cell's pixels. The cells can be as small as 1 pixel wide, larger cells decrease the overall computational complexity, as that results in fewer image blobs.

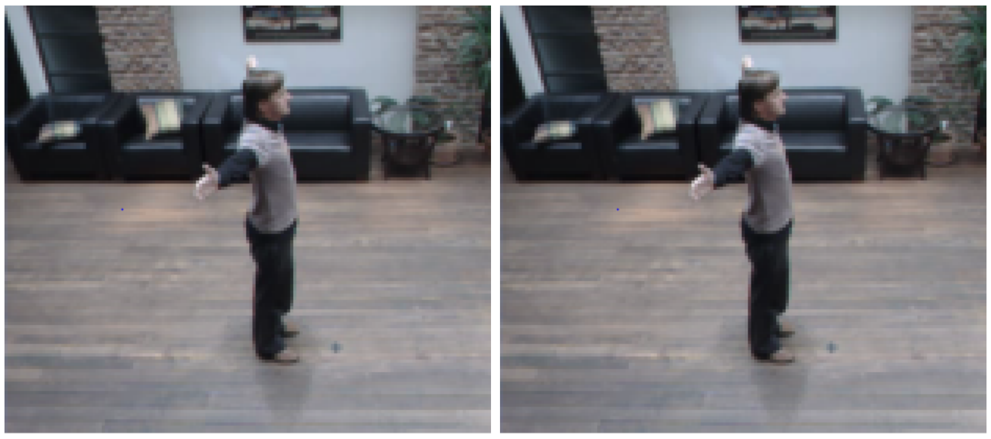


Figure 2.3: Image decomposition. Variable size blobs (left), Fixed size blobs (right)

We have tried both methods in Kostilam. Having fewer image blobs in the image model will speed up the computation of the similarity function, so the irregular structure looks promising at first. In practices, however, the irregular structure didn't reduce the number of blobs significantly for it to be worth the overhead it takes to create.

Creating the regular structure of the image model is as simple as going through all the image pixels once and averaging colors in square cells on the way. This proved to be a good compromise for Kostilam, also ability to control the cell size allows to easily create a finer or coarser approximation of the image. This choice may need to be reconsidered if we would use higher resolution cameras, currently we used cameras with $640x480$ resolution.

This approximation is not particularly precise in that we are actually replacing square cells of pixels with circular Gaussian blobs. This may even seem as quite questionable to say the least, on the other hand it is precisely this definition that allows us to later formulate a relatively simple analytical similarity function between the tracked model and the camera images.

2.3 Similarity measure

In the following paragraphs we will introduce one of the most important parts of the tracking algorithm - a similarity measure between the body model and the camera images. We have taken inspiration for it heavily in Stoll et al. [33], it is a core part of their approach as well. The measure works by first projecting the body model from the world coordinates to individual camera images and then calculating an overlap between the images' blobs and projected model blobs.

2.3.1 Model projection

We are using a projection matrix P_i of size 3×4 to model the geometry of each of the cameras C_i, each camera also has a focal length f_i which can be extracted from the projection matrix. We will define an operation Ψ which will project a 3D Gaussian $\tilde{\mathcal{B}}(\mathbf{x}) : \mathbb{R}^3 \to \mathbb{R}$ to a Gaussian in the image space $\mathcal{B}(\mathbf{x}) : \mathbb{R}^2 \to \mathbb{R}$. In other words, it will map the parameters of the 3D blob $(\tilde{\mu}, \tilde{\sigma})$ to the parameters of the 2D blob (μ, σ). First we project the center $\tilde{\mu}$ into the image to get a point $\mu^p = P_i\tilde{\mu}$ in *homogeneous coordinates*, then the operation Ψ goes as follows:

$$\mu = \begin{pmatrix} [\mu^p]_x / [\mu^p]_z \\ [\mu^p]_y / [\mu^p]_z \end{pmatrix} \qquad \sigma = \frac{\tilde{\sigma} f_i}{[\mu^p]_z} \tag{2.6}$$

The subscripts x, y and z are used to index the first, second and third coordinates respectively. The process of division by the third coordinate is a standard procedure used to bring the points from homogeneous coordinates to the image domain. The projection Ψ is actually only a simplification because spheres in general don't project circles, but to ellipses [18]. As the authors of [33] we have chosen to disregard this, again to keep the similarity measure simple enough. An example of the projection can be seen on fig. 2.4.

To obtain the matrices P_i, we use the *Multi Camera Self Calibration toolbox* [35] provided to us by CMP [12]. We gather a set of multi-view correspondences by moving a point light source in the space so that it is visible in all the cameras, this gives a good input dataset for the calibration toolbox.

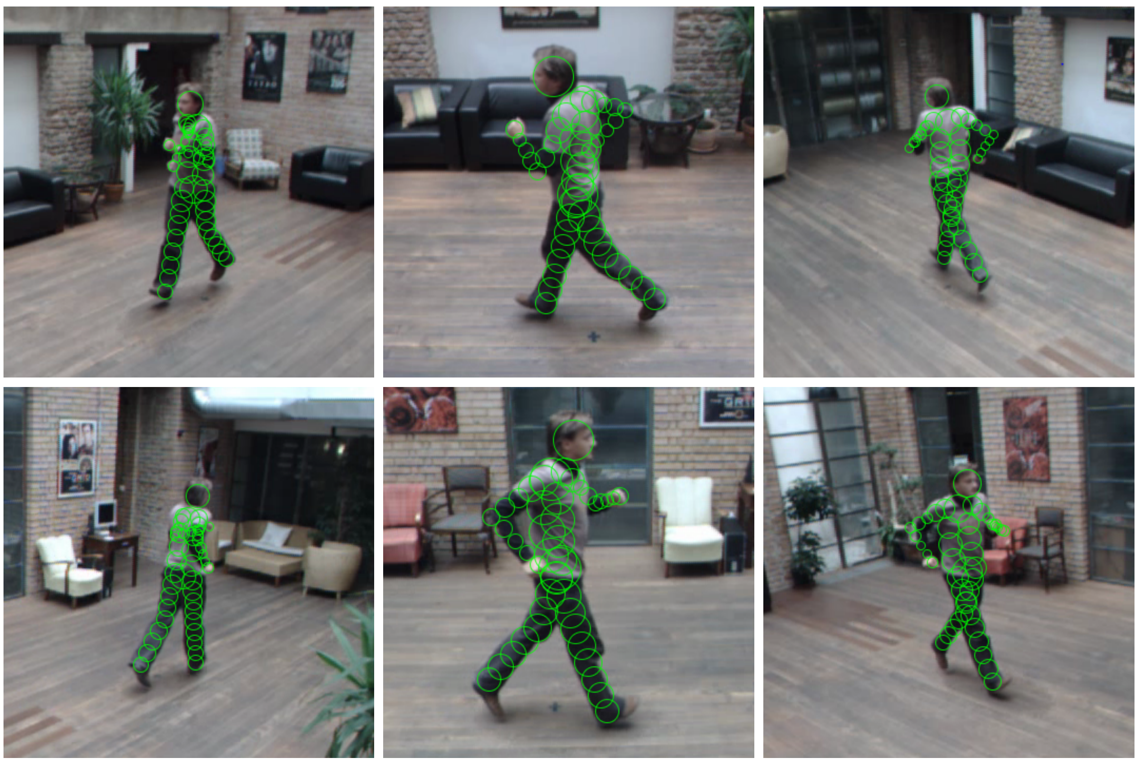


Figure 2.4: Projection of the body model to 6 different camera images using the (2.6) formula. Note that only crops from the whole images are shown here for better detail.

2.3.2 Blob similarity

The similarity E_{ij} of two blobs $\mathcal{B}_i$ and $\mathcal{B}_j$ consists of two parts; the color similarity $d(\mathbf{c}_i, \mathbf{c}_j)$ and the overlap of the blobs' volumes, which is calculated as an integral over the image domain Ω. Good for us, the integral can also be computed analytically:

$$\begin{aligned} E_{ij} &= d(\mathbf{c}_i, \mathbf{c}_j) \int_\Omega \mathcal{B}_i(x)\mathcal{B}_j(x)\mathrm{d}x && (2.7) \\ &= d(\mathbf{c}_i, \mathbf{c}_j)\, 2\pi \frac{\sigma_i^2 \sigma_j^2}{\sigma_i^2 + \sigma_j^2} \exp\left(-\frac{\|\mu_i - \mu_j\|_2^2}{\sigma_i^2 + \sigma_j^2}\right) && (2.8) \end{aligned}$$

For the color similarity $d(\mathbf{c}_i, \mathbf{c}_j)$ we used exactly the same function as the authors of [33]:

$$d(\mathbf{c}_i, \mathbf{c}_j) = \begin{cases} 0 & \text{if } \|\mathbf{c}_i - \mathbf{c}_j\|_2 \geq \epsilon_{sim} \\ \varphi_{3,1}\left(\frac{\|\mathbf{c}_i - \mathbf{c}_j\|_2}{\epsilon_{sim}}\right) & \text{if } \|\mathbf{c}_i - \mathbf{c}_j\|_2 < \epsilon_{sim} \end{cases} \tag{2.9}$$

Where $\varphi_{3,1}(x)$ is a smooth radial basis function. This way defined color similarity forces distant colors to always have similarity 0, but colors closer than ϵ_{sim} will enjoy a smooth similarity measure.

2.3.3 Model similarity

Now that we have all the necessary preliminaries ready, we can proceed to describing the similarity function between the image model and the projected body model. Lets say we have a body model $\mathcal{K}_m$ consisting of a sum of Gaussians (2.2), then we will denote its projection to a camera image as $\Psi(\mathcal{K}_m)$. The camera image consists of a sum of Gaussians as well, we will call it $\mathcal{K}_I$. The similarity E is then defined as a summation of overlaps (2.8) of all pairs of blobs, first one being from the image and the second one from the projected model:

$$E(\mathcal{K}_I, \Psi(\mathcal{K}_m)) = \sum_{i \in \mathcal{K}_I} \sum_{j \in \Psi(\mathcal{K}_m)} E_{ij} \tag{2.10}$$

$$= \int_\Omega \sum_{i \in \mathcal{K}_I} \sum_{j \in \Psi(\mathcal{K}_m)} d(\mathbf{c}_i, \mathbf{c}_j) \mathcal{B}_i(x) \mathcal{B}_j(x) \mathrm{d}x \tag{2.11}$$

In practice it is not necessary to compute the overlaps (2.8) for all possible pairs of blobs, it suffices for example to only consider the image blobs which are within some distance from the projected body model.

There is a caveat, however, in the way the similarity measure (2.10) is defined. It is caused by the fact that very often blobs end up being projected on top of each other, this can be seen in fig. 2.4. We want the similarity measure to be roughly proportional to the area of the total overlap of the projected model and the tracked subject in the image. Half of this comes from the color similarity, but the other half can be skewed considerably by overlapping blobs - if the colors would be the same, 2 blobs projected to the same position would have the same total overlap as 2 distant blobs.

We have solved this problem the same way as authors of [33] by defining the maximum value a single image blob $\mathcal{B}_i$ can contribute to the total value of the similarity measure. That is set to be the overlap of the given image blob with itself: E_{ii}. Then the similarity can be rewritten to reflect this in the following way:

$$E(\mathcal{K}_I, \Psi(\mathcal{K}_m)) = \sum_{i \in \mathcal{K}_I} \min \left(\sum_{j \in \Psi(\mathcal{K}_m)} E_{ij},\ E_{ii} \right) \tag{2.12}$$

We have found that without introducing this way of limiting image blobs' contributions to the total overlap, our software couldn't track the subjects reliably.

2.4 Cost function

Now we can step up to describe the objective function which is at the heart of Kostilam's tracking algorithm. We have a body model $\mathcal{K}_m$ which has a set of parameters Θ, in the case of the positional model (2.1.1) it is 45 translational parameters, in the case of the angular model (2.1.2) it is 3 translational and 29 rotational parameters; these together determine the position of the model's blobs. We also have a set of n_{cam} cameras $\mathcal{C}_l$ together with the sum of Gaussians (2.2) representation $\mathcal{K}_l$ of their images.

We project the model to all the cameras and calculate a weighted sum of the corresponding similarity measures (2.12):

$$E(\Theta) = \frac{1}{n_{cam}} \sum_{l=1}^{n_{cam}} \frac{1}{E(\mathcal{K}_l, \mathcal{K}_l)} E\Big(\mathcal{K}_l, \Psi(\mathcal{K}_m(\Theta))\Big) \tag{2.13}$$

The term $E(\mathcal{K}_l, \mathcal{K}_l)$ in the denominator is there to balance the contributions of camera views where in one the model may fill most of the image while in another the model can be significantly smaller.

Altogether we want to maximize the value of $E(\Theta)$ so it will appear in the final objective function with a negative sign. One last piece of the objective function we need to present is a term responsible for limiting the possible values the parameters Θ can take. This is done on an individual basis by introducing a set of soft limits l_l and l_h and a quadratic penalty term $E_{lim}(\Theta)$:

$$E_{lim}(\Theta) = \sum_{\Theta(i)} \begin{cases} 0 & \text{if } l_l^{(i)} \le \Theta(i) \le l_h^{(i)} \\ (l_l^{(i)} - \Theta(i))^2 & \text{if } \Theta(i) < l_l^{(i)} \\ (\Theta(i) - l_h^{(i)})^2 & \text{if } \Theta(i) > l_h^{(i)} \end{cases} \tag{2.14}$$

Putting the terms (2.13) and (2.14) together we get the final cost function $\mathcal{E}$, we used w as a weighting parameter:

$$\mathcal{E}(\Theta) = wE_{lim}(\Theta) - E(\Theta) \tag{2.15}$$

By minimizing $\mathcal{E}(\Theta)$ we want to achieve maximal overlap of the projected body model with the subject in corresponding camera images, while also keeping some rigidity of the model thanks to the penalty term E_{lim}. We haven't experimented with the value of the weighting parameter w considerably, having it equal to 1 worked well in limiting undesirable motions like shins or elbows bending in the opposite direction or the spine bending excessively.

2.4.1 Note about tracking

We will describe the approach we use in Kostilam to minimize the cost function (2.15) in the following chapter. Note that by nature of the problem, when we track the motion of a subject in a video sequence, we should always have a relatively good initial solution at the ready, the optimization process tries to determine the change of parameters Θ between consecutive frames. Of course, this assumes that the subsequent solution always gets found correctly, which doesn't need to be the case e.g. for too fast motions. On the other hand, Kostilam proved to be able to recover from getting "lost" in our experiments when the subject returned to its previous position.

Setting of the initial solution for the first frame is done manually in Kostilam and we will describe this process in chapter 4.

Chapter 3

Cost minimization

In this section, we will give a theoretical overview of the approach we took to minimize the above mentioned cost function 2.15. The demands for our problem of motion tracking are specific in that we are solving the cost minimization problem in each frame and in each of them we have a solid initial solution from the previous frame. The change in parameters between the frames corresponds to the subject's movement performed in the short interval between frames, in our case 1/60 of a second. This greatly reduces the complexity of the cost minimization making the problem a suitable fit for a gradient descent method.

Given the requirements for our solution to be able to perform in realtime, we needed a fast method for computing the gradient, however we also wanted our method to be 'programmer friendly' in that a small change to the cost function or the defined body model would not require a significant rewriting of the code, ideally for it to be fully automatic.

A common practice in computing derivatives is that for each function $f(x)$ in the code one writes a separate function $f'(x)$ which computes its derivative. This process is usually done manually and it is prone to small and hardly noticeable mistakes when one forgets to update the definition of $f'(x)$ or makes a mistake in it. Although the process of writing the definition for $f'(x)$ can be automated, leading to symbolic differentiation of code [17], we decided not to implement this as it is very hard to do effectively and also it makes some constraints on the code that can be used in the differentiated function.

Another possibility is to calculate a numerical approximation using the definition of derivative:

$$\lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{\delta} \tag{3.1}$$

The method of finite differences calculates approximates the true value of the derivative by substituting the δ with a small number. The major benefit of this method is that it needs only the definition of the function we want to differentiate and then an appropriate value of d. Unfortunately picking a good value for d proves to be problematic as too large values give big approximation errors and too low values introduce rounding errors.

A less known method and the one we finally chose to implement is called emphautomatic differentiation [26]. It solves both the problem of numerical errors the finite differences method has as it computes the derivative exactly and it is also fully automatic meaning we only need to supply code for the definition of the function. The method uses a concept of emphdual numbers which was originally introduced in [11] and then later used in [19], for our implementation our main source was a later article [26] whose authors also used the method in the field of computer graphics.

3.1 Automatic differentiation

Assume we want to take a derivative of the function $f(x) = 3x^2 + 5$, the idea behind automatic differentiation is to substitute δ in (3.1) by a small number d, then for our function $f(x)$ the formula evaluates to:

$$\frac{3(x+d)^2 + 5 - 3x^2 - 5}{d} = \frac{6dx + 3d^2}{d} = 6x + 3d \tag{3.2}$$

On the right side we can see the exact derivative $6x$ plus an error term $3d$ where d is a small number. Now the idea is to remove the error term completely which could be done for some value of d such that $d^2 = 0$ as can be seen from the intermediate step. Unfortunately the only real number for which this is possible is 0 and then the formula wouldn't make sense. The solution to this problem is seemingly counter-intuitive, it lies in defining a special constant d which has the desired property $d^2 = 0$, this approach is similar to adding the imaginary unit to real numbers to solve the equation $x^2 + 1 = 0$. The set of real numbers extended by the constant d is called the emphdual numbers [9]. Having the number d at our disposal we can now rewrite the equation (3.1) as:

$$f(x+d) = f(x) + df'(x) \tag{3.3}$$

Which allows us to compute the derivative simply by evaluating the given function at $x + d$, consider the previous example of $f(x) = 3x^2 + 5$:

$$f(x+d) = 3(x^2 + 2xd + d^2) + 5 = (3x^2 + 5) + 6xd + 3d^2 \tag{3.4}$$

Since d^2 is defined to be 0, by comparing (3.3) and (3.4) we can see that the value of $f'(x)$ is exactly the coefficient of d in the result, in this case $6x$.

Let us show another example of using this strategy to compute derivatives, this time on a little more complex quadratic function $g(x) = (x + 3) * (x + 2)$. Let's say we want to take the derivative at $x = 2$, following the previous example, we should evaluate the function with the argument $2 + d$:

$$(2 + d + 3)(2 + d + 2) = (5 + d)(4 + d) = 20 + 9d + d^2 = 20 + 9d \tag{3.5}$$

The derivative at $x = 2$ is the coefficient of d in the result, in this case it is 9. We can also reach this this result using standard symbolic derivation:

$$g(x) = x^2 + 5x + 6 \tag{3.6}$$
$$g'(x) = 2x + 5 \tag{3.7}$$
$$g'(2) = 9 \tag{3.8}$$

3.1.1 Generalization

To generalize the above introduced principle on a wide variety of functions we have used the concept of *Dual object* as described in [26].

- Dual object: $a + bd$
    1. a - real part, scalar
    2. b - infinitesimal part, scalar

The idea is to use the dual objects as the default number type in the functions we want to take a derivative of. The parameters have to be dual objects, all the intermediate values and the result value as well. By doing that, taking a derivative is then a simple matter of reading the value of b of the function return value. In certain programming languages, most notably in C++, this approach can be relatively easily implemented using the templating functionality, which allows the programmer to write the functions completely oblivious to the number type being used, as long as the type is capable of certain operations e.g. addition or multiplication.

3.1.2 Operations over dual objects

Here we will show the operations which need to be implemented in order to compute a vast majority of functions. The process is similar to evaluating operations of complex numbers, which also implies that evaluating functions using dual objects is of comparable computational complexity as if we were using complex numbers.

The most simple operation is addition of two dual objects:

$$(a_0 + b_0 d) + (a_1 + b_1 d) = (a_0 + a_1) + (b_0 + b_1)d \tag{3.9}$$

Subtraction is along the same line, while product of two dual objects is as follows:

$$(a_0 + b_0 d)(a_1 + b_1 d) = (a_0 a_1) + (a_0 b_1 + b_0 a_1)d \tag{3.10}$$

Computing division is not that straightforward and requires a different form:

$$\frac{a_0 + b_0 d}{a_1 + b_1 d} = \frac{a_0 + b_0 d}{a_1} \frac{1}{1 + (b_1/a_1)d} \tag{3.11}$$

At this point we use the following expansion $(1 + x)^{-1} = 1 - x + x^2 - x^3 + (\ldots)$:

$$= \frac{a_0 + b_0 d}{a_1}(1 - (b_1/a_1)d + d^2(\ldots)) \tag{3.12}$$

$$= \frac{a_0}{a_1} + \frac{a_1 b_0 - a_0 b_1}{a_1^2} d \tag{3.13}$$

It makes sense that division by a dual object with the real part being 0 is not defined.

It is already possible to evaluate a big range of functions with only the operations (3.9), (3.10) and (3.13). The last step required to cover all the operations used in our cost function 2.15 is to be able to evaluate transcendental functions with the dual objects, most importantly for us the trigonometric functions. For this we use the following rule from [26] for differentiable functions:

$$f(a + bd) = f(a) + bf'(a)d \tag{3.14}$$

The trigonometric functions sine and cosine can then be evaluated:

$$\sin(a + bd) = \sin(a) + b\cos(a)d \tag{3.15}$$
$$\cos(a + bd) = \cos(a) - b\sin(a)d \tag{3.16}$$

3.1.3 Multidimensional differentiation

So far we have been describing exclusively computation with functions of a single variable. For computing the gradient of our cost function 2.15, we need all its partial derivatives with respect to the model parameters. Again using the idea from [26] we accomplish this by introducing a new variable d_i for each partial derivative we want to compute. All these variables d_i will have the same properties as the original variable d introduced in 3.1. On top of that it is defined that for all pairs i, j it holds that $d_i d_j = 0$. The dual object as defined in section 3.1.1 can then be extended to accommodate partial derivatives as follows:

- Extended dual object: $a + \sum_{i \in I} b_i d_i$, where $I = \{1, \ldots, n\}$
    1. a - real part, scalar
    2. $\mathbf{b}$ - infinitesimal part, vector of length n

Analogously to the singlevariate case, computing the partial derivatives is then a matter of evaluating the desired function using the extended dual object as the number type for parameters and then reading the corresponding coefficient of d_i from the return value.

For a better illustration, assume we have a function of two variables $f(x, y)$ and we want to compute partial derivatives with respect to both x and y. The analogy for the equation (3.3) for a two-dimensional function will look like this:

$$f(x + d_1, y + d_2) = f(x, y) + \frac{\partial f}{\partial x} d_1 + \frac{\partial f}{\partial y} d_2 \tag{3.17}$$

To compute the functions over extended dual objects the operations introduced in the previous section 3.1.2 will generalize relatively simply. In the following equations we will use vector $\mathbf{d} = (d_1, \ldots, d_n)$. The rules for addition, multiplication and division of extended dual objects are the following:

$$(a_0 + \mathbf{b_0}^\top \mathbf{d}) + (a_1 + \mathbf{b_1}^\top \mathbf{d}) = (a_0 + a_1) + (\mathbf{b_0} + \mathbf{b_1})^\top \mathbf{d} \tag{3.18}$$

$$(a_0 + \mathbf{b_0}^\top \mathbf{d})(a_1 + \mathbf{b_1}^\top \mathbf{d}) = (a_0 a_1) + (a_0 \mathbf{b_1} + \mathbf{b_0} a_1)^\top \mathbf{d} \tag{3.19}$$

$$\frac{a_0 + \mathbf{b_0}^\top \mathbf{d}}{a_1 + \mathbf{b_1}^\top \mathbf{d}} = \frac{a_0}{a_1} + \left(\frac{a_1 \mathbf{b_0} - a_0 \mathbf{b_1}}{a_1^2}\right)^\top \mathbf{d} \tag{3.20}$$

And last, the rule for evaluating differentiable functions:

$$f(a + \mathbf{b}^\top \mathbf{d}) = f(a) + f'(a)\mathbf{b}^\top \mathbf{d} \tag{3.21}$$

When these rules and all needed specializations of the last rule (3.21) are implemented, it becomes possible to evaluate all kinds of functions using the extended dual objects, therefore we gain access to the desired partial derivatives, which can be read off as the coefficients of d_i from the result.

3.1.4 Example

To get more familiar with the process of automatic differentiation, let us show the computation on a more complex example of a function of three variables:

$$f(x, y, z) = \sin\left(\frac{x + y^2}{z}\right) \tag{3.22}$$

Let's say we want to calculate the derivatives at point $(2\pi, 0, 3)$ with respect to variables x, y and z, then we will use the following extended dual objects as arguments:

- $\hat{x} = 2\pi + d_1 \quad (a = 2, \mathbf{b} = (1, 0, 0))$
- $\hat{y} = 0 + d_2 \quad (a = 0, \mathbf{b} = (0, 1, 0))$
- $\hat{z} = 3 + d_3 \quad (a = 3, \mathbf{b} = (0, 0, 1))$

Then the evaluation will be:

$$f(\hat{x}, \hat{y}, \hat{z}) = \sin\left(\frac{2\pi + d_1 + d_2^2}{3 + d_3}\right) = \sin\left(\frac{2\pi + d_1}{3 + d_3}\right) = \tag{3.23}$$

The term d_2^2 is zero by definition. Next we will use the rule for division (3.20):

$$= \sin\left(\frac{2\pi}{3} + \frac{3d_1 - 2\pi d_3}{3^2}\right) \tag{3.24}$$

And then the rule for differentiable functions (3.21):

$$= \sin\left(\frac{2\pi}{3}\right) + \cos\left(\frac{2\pi}{3}\right)\frac{3d_1 - 2\pi d_3}{9} \tag{3.25}$$

We repeat the above result here:

$$= \sin\left(\frac{2\pi}{3}\right) + \left(-\frac{1}{2}\right)\frac{3d_1 - 2\pi d_3}{9} \tag{3.26}$$

The derivatives at point $(2\pi, 0, 3)$ are the coefficients of d_1, d_2 and d_3 respectively:

$$\frac{\partial f}{\partial x}(2\pi) = -1/6 \qquad \frac{\partial f}{\partial y}(0) = 0 \qquad \frac{\partial f}{\partial z}(3) = \pi/9 \tag{3.27}$$

To verify the results, we can take standard symbolic derivatives of function $f(x, y, z)$:

$$\frac{\partial f}{\partial x} = \frac{1}{z}\cos\left(\frac{x+y^2}{z}\right) \qquad \frac{\partial f}{\partial y} = \frac{2y}{z}\cos\left(\frac{x+y^2}{z}\right) \tag{3.28}$$

$$\frac{\partial f}{\partial z} = -\frac{x+y^2}{z^2}\cos\left(\frac{x+y^2}{z}\right) \tag{3.29}$$

By substituting for the values $x = 2\pi$, $y = 0$ and $z = 3$ we can see that the method of automatic differentiation gives correct results for the given function.

$$\frac{\partial f}{\partial x} = \frac{1}{3}\cos\left(\frac{2\pi}{3}\right) = \frac{1}{3}\left(-\frac{1}{2}\right) = -\frac{1}{6} \qquad \frac{\partial f}{\partial y} = \frac{0}{3}(\cdots) = 0 \tag{3.30}$$

$$\frac{\partial f}{\partial z} = -\frac{2\pi}{9}\left(-\frac{1}{2}\right) = \frac{\pi}{9} \tag{3.31}$$

The given example can also be used to explain an interesting difference between the methods of finite differences and automatic differentiation. To get approximations for all partial derivatives of the function (3.22), we have to evaluate the trigonometric function at least $3 + 1$ times for one-sided difference or 6 times for symmetric difference. On the other hand with automatic differentiation it suffices to calculate the value of the trigonometric function just once, as can be seen from (3.25).

Hopefully, we have managed to show that automatic differentiation is a viable method for our purpose. This theoretical overview didn't consider the technical details of implementing the extended dual objects to some programming language. That will be the main focus of the chapter 5 as there are many ways one can choose for an efficient implementation, the main problem being an effective manipulation of vectors **b** (3.1.3) in memory as they will contain as many elements as there are parameters for our model.

3.2 Gradient descent

Being able to compute the derivatives effectively it is now necessary to choose a suitable minimization algorithm. In the paper [26] the authors use the *Levenberg-Marquardt* minimization together with computing the full *Hessian* matrix, they also describe a clever way of using the dual objects "recursively" to compute higher-order derivatives. We didn't choose this approach as we didn't feel confident with making it run effective enough with the problem of real-time motion tracking.

Another way is described in the article by Stoll [33] where the method of choice is a custom conditioned gradient descent method inspired by [29] where it is used to learn parameters for neural networks.

In the end we chose the *L-BFGS* algorithm (Limited-memory Broyden-Fletcher-Goldfarb-Shanno) which is a *quasi-Newton* method, meaning that it approximates the Hessian which is used to steer the algorithm efficiently through the solution space. According to [25] the most important part of the method - the update step for approximating the Hessian - has been independently developed by Broyden [6], Fletcher [13], Goldfarb [15] and Shanno [32], whose results have all been published in 1970. This method combines the low iteration count inherent to second-order minimization methods while still requiring only the gradient for the computation which made it a good fit for our problem.

To run this method in Kostilam we used an open-source implementation of the L-BFGS method written by Nocedal and Okazaki [23].

Chapter 4

Implementation overview

So far we have talked about the core concepts behind the Kostilam motion capture software.

In chapter 2 we have introduced a sum of Gaussian (2.2) model, its parametrization and a cost function (2.15) to determine which values of parameters most closely represent the actor's pose in multiple camera images.

Later in chapter 3 we have talked about the way we minimize this cost function - by using the L-BFGS optimization algorithm where we compute the cost's gradient by the method of automatic differentiation.

In this chapter we will set all the pieces together, giving a complete overview of the process of tracking an actor's performance using the Kostilam markerless motion capture software. The parts that didn't fit in the previous chapters will be laid out in detail in this chapter, those include among others: body model fitting, initial solution selection, color calibration or light source adaptation.

4.1 Overview

To give a complete and concise image of the whole motion capture process, each of its important parts will get a short summary in the following paragraphs. After that we will give a more detailed description of those parts that require it.

Fitting a body model. This is the process of adapting the sum of Gaussians body model to fit to an individual actor as close as possible. Important parameters are the bone lengths and shape. The fitting can be done manually in Kostilam, however, we have also developed a semi-automatic tool to help estimate the parameters, its description will follow in section 4.3

Initial solution. Finding the initial solution for the first frame means placing and shaping the body model in space in the same position as is the actor. This step is also done manually; Kostilam allows for the user to simply drag the model in space by its joints. For the case of the angular model we have developed an *inverse kinematics* approach to make the process more user friendly, this will be described in section 4.2

Color calibration. When the model is in good shape and place, we need to assign specific colors to its blobs. We do this by back-projecting the camera images to the 3D

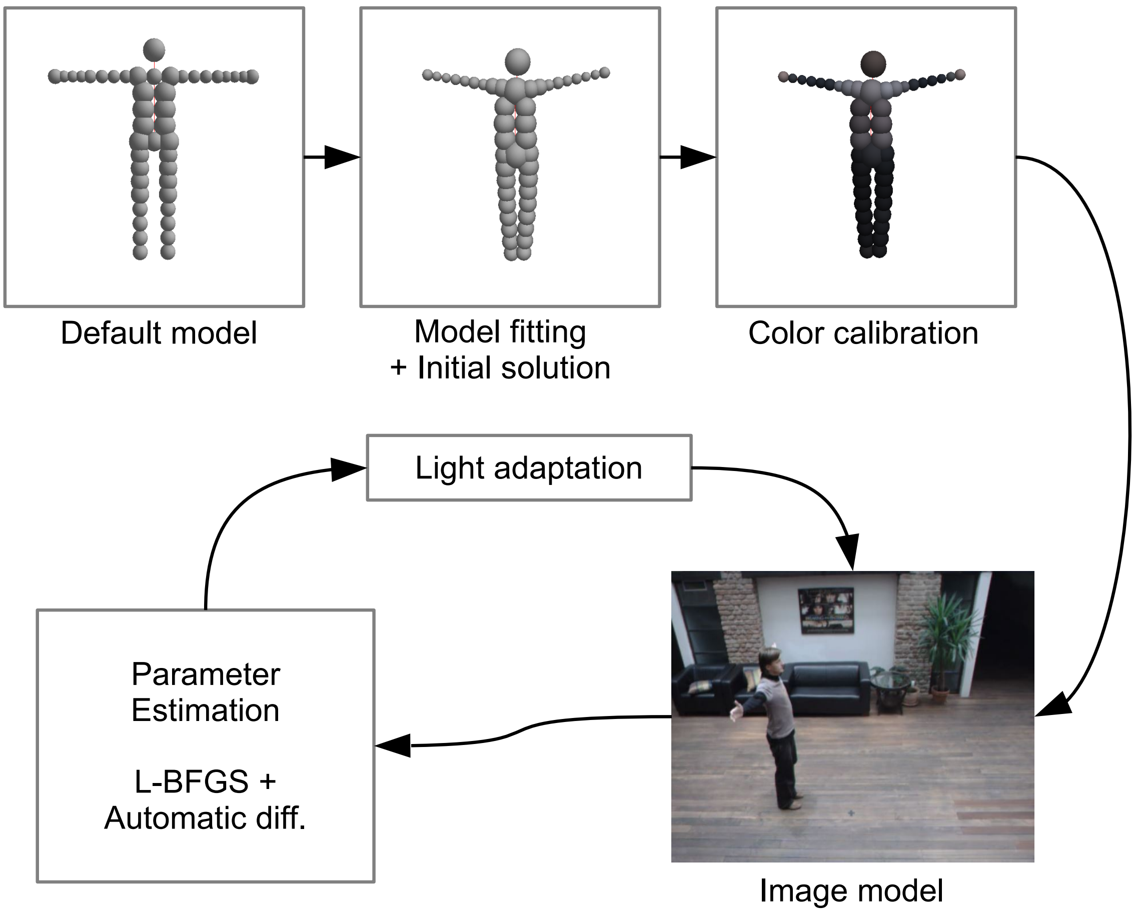


Figure 4.1: Showing all the parts of the motion capture algorithm.

space; essentially by casting a *camera ray* through each pixel and seeing if it intersects our model. We will give more details and illustrations of this in section 4.4.

Creating an image model. We need to create a sum of Gaussians image model for each of the frames of the tracked sequence. Moreover we want the image models to contain only the relevant blobs, ideally only the blobs which cover the actor in the given view. For this we use *culling* of blobs outside a bounding box given by the projected body model and a *background subtraction* based technique. More on this will be shown in section 4.5.

Estimating parameters. This is the most important part of the tracking software, it involves running the L-BFGS optimization algorithm to minimize the cost function $\mathcal{E}(\Theta)$ 2.15, which results in a new set of parameters that hopefully represent the actor's pose in the new frame. This step has been already covered extensively in the previous chapters.

Light source adaptation. We have encountered in our motion capture performances some problems related to color variation in time either with the actor changing color under different angles or with the whole scene changing brightness. We have devised a relatively simple countermeasure involving color recalibration when certain thresholds are met. We will detail the method in section **??**.

4.2 Initial solution

We have a 3D visualization of the model space built in Kostilam, to help find the initial pose in it, we have created a helper point cloud which represents the actors body (fig. 4.2). The point cloud is created by projecting a uniform grid of 3D points (inside some predefined bounding box) to all the camera images and then checking which of the points project to the actor in all of them. To determine if the point was projected to the actor, we use simple background subtraction which gives a rough estimate.

Now the task for the user is to move the predefined model to match the cloud of points, there are operations like translation, rotation and scaling available, usually working by mouse dragging. This can be done relatively easily for the original version of the model (2.1.1) where all the parameters are translational, on the other hand with the angular model((2.1.2)) moving certain joints of the model requires solving for all the translational and rotational parameters of the model.

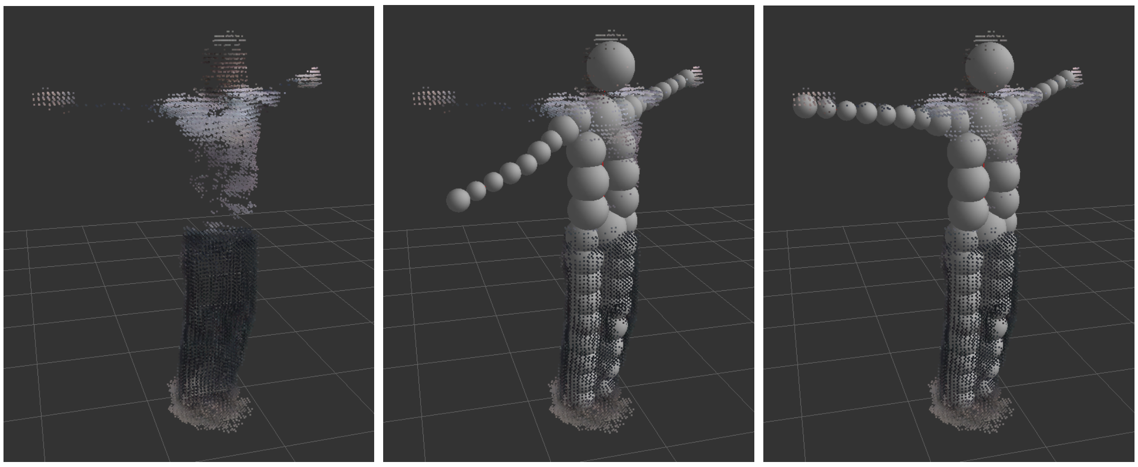


Figure 4.2: Setting the initial solution for the first frame. Helper point cloud (left) moving the model's arm via inverse kinematics (middle) initial solution and the point cloud (right)

4.2.1 Inverse kinematics

Inverse kinematics is a method of finding the parameters of a model given some requirements on position of its joints. It is quite broadly studied e.g. the summary [7]. Given that this wasn't the key part of our project, we can consider ourselves lucky that it was fairly easy to implement given the tools we already developed for minimizing the motion tracking cost function 2.15.

We introduced a set of *effectors* $e_i \in \mathbb{R}^3$, each of which we attached to some joint that it should control and then we allowed the effectors to be dragged by mouse. The set of parameters we want to solve for is Θ and $\mu_i(\Theta)$ is the position of the blobs in space (2.5). Then we can write this cost function:

$$E_{IK}(\Theta) = E_{lim}(\Theta) + \sum_i \|e_i - \mu_i(\Theta)\|_2^2 \tag{4.1}$$

Where $E_{lim}(\Theta)$ is the quadratic penalty term (2.14) responsible for limiting unnatural movements of the model. After that we used automatic differentiation to compute the gradient of this cost function and the L-BFGS optimization algorithm to solve for parameters Θ. This way we got a surprisingly strong inverse kinematics solution which allowed for the model to be moved around and shaped simply by dragging its joints.

4.3 Fitting a body model

In the previous section we showed what technique we use to position the model in space. Once that is done we proceed to adjust the model's fixed parameters to fit the individual actor, most importantly bone lengths and blob sizes. In our experiments we have found that especially for the angular model 2.1.2, our tracking algorithm is sensitive to discrepancies between the virtual model and the real actor. Without setting the correct dimensions for the skeleton, the algorithm was failing to track the actor reliably.

We have developed a semi-automatic tool for model fitting based on the ideas from [33]. Again we exploited the tools we had already at hand, extending the cost function 2.15 and using automatic differentiation together with the L-BFGS optimization algorithm.

The idea behind it is based on picking several poses from the sequence and trying to fit the model to all of them at once. Since we don't have the colors of the model yet at this point and since we only care for the shape of the model at this step, the input will consist of binary masks representing the position of the actor in respective frames (fig. 4.3). Unfortunately, those have to be segmented manually at this point.

4.3.1 Multi-pose cost function

Here we will introduce the cost function we used to fit the body model to an actor. We have n_{pos} poses, each consisting of n_{cam} camera images. For each pose p and camera c we create a sum of Gaussians representation $\mathcal{M}_c^{(p)}$ of the binary mask representing the actor in the image. For the next part we will take a second look at the similarity measure between a sum of Gaussians model and a projected body model (2.10) defined in chapter 2.

$$E(\mathcal{K}_I, \Psi(\mathcal{K}_m)) = \int_\Omega \sum_{i\in\mathcal{K}_I} \sum_{j\in\Psi(\mathcal{K}_m)} d(\mathbf{c}_i, \mathbf{c}_j)\mathcal{B}_i(x)\mathcal{B}_j(x)\mathrm{d}x \tag{4.2}$$

We will modify this similarity measure so that it doesn't rely on color information but only on the volume overlap, the idea is to reward blobs that project to the actor and to penalize blobs that project outside of it. To implement this we will use function $d(\mathbf{c}_i)$ defined to only have two values; $d(\text{black}) = 1$ and $d(\text{white}) = -w$, where the value of w represents the relative importance of having the model inside or outside a predefined mask, as seen in

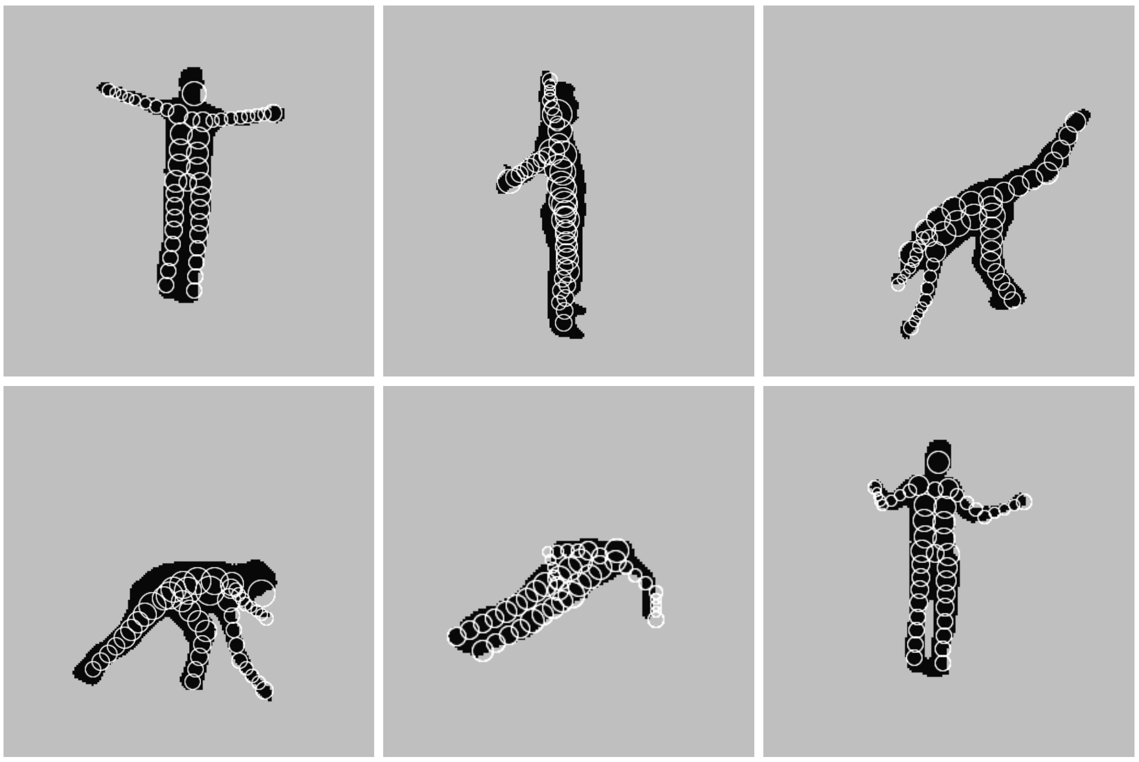


Figure 4.3: Several masks all used together to optimize the body model. The projected model on top of them is a rough initial solution picked by hand.

fig. 4.3. We have experimented with several values of w and we found the most satisfying results around $w = 0.1$. The modified similarity measure follows:

$$\tilde{E}(\mathcal{M}_c, \Psi(\mathcal{K}_m)) = \int_\Omega \sum_{i \in \mathcal{M}_c} \sum_{j \in \Psi(\mathcal{K}_m)} d(\mathbf{c}_i)\mathcal{B}_i(x)\mathcal{B}_j(x)\mathrm{d}x \tag{4.3}$$

Now consider all the parameters that occur in our setup. All of these will be later stacked together and optimized for using automatic differentiation and L-BFGS:

Shared parameters These parameters are shared amongst all the poses, they represent the physical dimensions of the model we are trying to fit. In the case of our human body model, we have 9 parameters for bone lengths, 1 parameter for chest width and 24 parameters for blob sizes. Notice that there are not as many parameters as there are bones or blobs in our model, that is because we are also sharing certain parameters between relevant body parts. It makes sense for example to make the lengths of bones to be equal on both sides of the body. In total we have 34 shared parameters, let us denote them Λ.

Pose related parameters These are the parameters that change in each pose - the 32 translational and rotational parameters that determine the position and posture of the body model. Let us denote the parameters of i-th pose Θ_i.

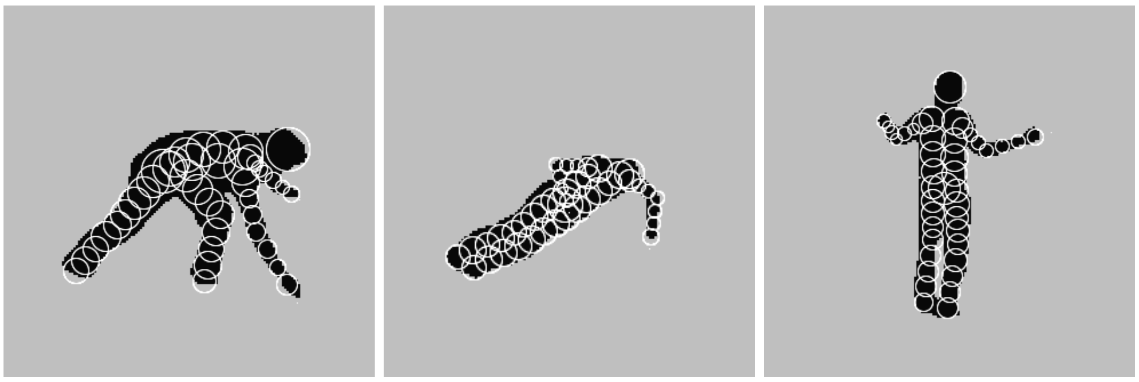


Figure 4.4: Here we can see the body model fitted to the image masks using the method outlined in this section, comparison can be made with the previous figure.

Putting all these parameters together gives us the full parameter vector Φ:

$$\Phi = (\Lambda, \Theta_1, \ldots, \Theta_{n_{pos}}) \tag{4.4}$$

Now defining the final cost function responsible for the model fitting is a matter of summing the modified similarity measure $\tilde{E}$ over all the n_{pos} poses and in each of them over all the n_{cam} cameras:

$$\tilde{\mathcal{E}}(\Phi) = \sum_{p=1}^{n_{pos}} \sum_{c=1}^{n_{cam}} \tilde{E}\Big(\mathcal{M}_c^{(p)}, \Psi(\mathcal{K}_m(\Lambda;\Theta_p))\Big) \tag{4.5}$$

Our standard procedure of minimizing the cost function $\tilde{\mathcal{E}}(\Phi)$ using automatic differentiation and L-BFGS worked surprisingly well, some results are shown in fig. 4.4. Sometimes however, the full joint optimization proved to be a tough match so for that case we also allowed the individual sets of parameters to be optimized selectively, e.g. only optimizing the bone lengths or blob sizes.

4.4 Color calibration

At this point we have the sum of Gaussians body model positioned to reflect the actor's pose in the first frame and also adjusted to resemble the actor's shape as close as possible. The next step is to assign a color c_i to each individual blob of the model.

Recall that we use the standard 3×4 projection matrix P_i to model the geometry of our cameras, furthermore for the color calibration step we use the following decomposition:

$$P = KR\Big[I \quad -\mathbf{c}\Big] \tag{4.6}$$

Where matrix $K \in \mathbb{R}^{3\times3}$ is the *camera calibration matrix*, $R \in \mathbb{R}^{3\times3}$ is a rotation from the world coordinate system to the camera coordinate system, $I \in \mathbb{R}^{3\times3}$ is an identity matrix and $c \in \mathbb{R}^3$ is the camera center.

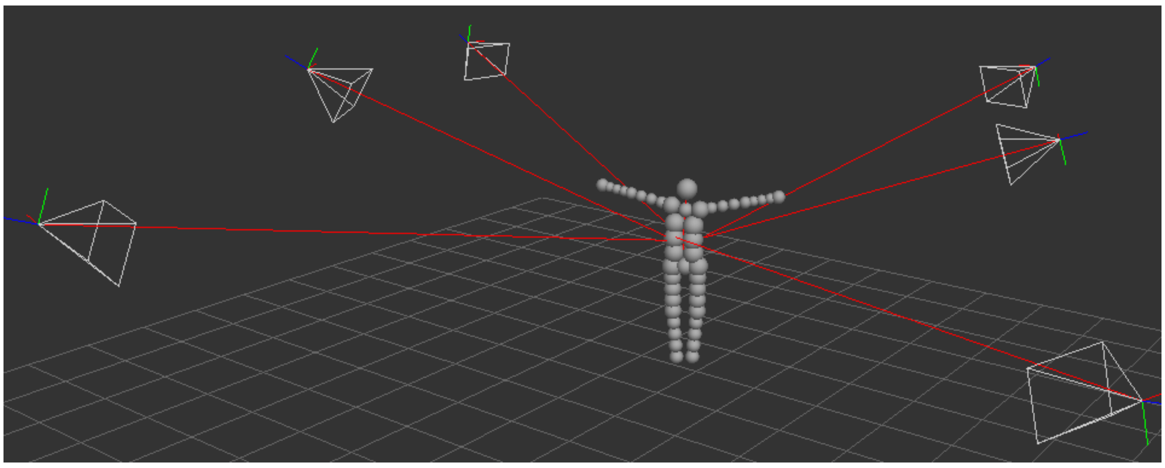


Figure 4.5: Color calibration of the model, casting rays through camera center and image pixel. Here we can see a single blob getting a contribution for its color from multiple cameras.

Typically the matrix P is used to project 3D points from the world coordinate system to 2D points in the image domain. For the color calibration part we reverse the process that is we cast a ray through the center of the camera and a pixel and then we check whether the ray hits some of the blobs in the model.

Consider that we have an image pixel with the coordinates $\mathbf{x} \in \mathbb{R}^2$, then to get the direction vector of the ray connecting it with the camera center we compute the following:

$$\mathbf{d} = (KR)^{-1} \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} = R^\top K^{-1} \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} \tag{4.7}$$

Then we intersect the ray $(\mathbf{c}, \mathbf{d})$ with all of the blobs of our body model and the blob that is the closest to the camera center gets assigned the color of the pixel to an accumulator. The colors are then averaged over all the contributions of various pixels. An illustration of this process is shown in fig. 4.5.

We usually carry out this process with the first frame of the video sequence, the color model obtained this way works well enough in sequences where the actor's color remains consistent throughout the whole sequence. When there are color variations in time, we try to compensate by relearning the color model more often. This will be described in detail in section **??**.

4.5 Creating an image model

The process of creating a sum of Gaussians representation of a camera image has already been described shortly in a previous chapter in 2.2. Note that we have to create the image model n_{cam} times for each of the frames of the tracked multi-view sequence so we can't afford anything too complex to be happening.

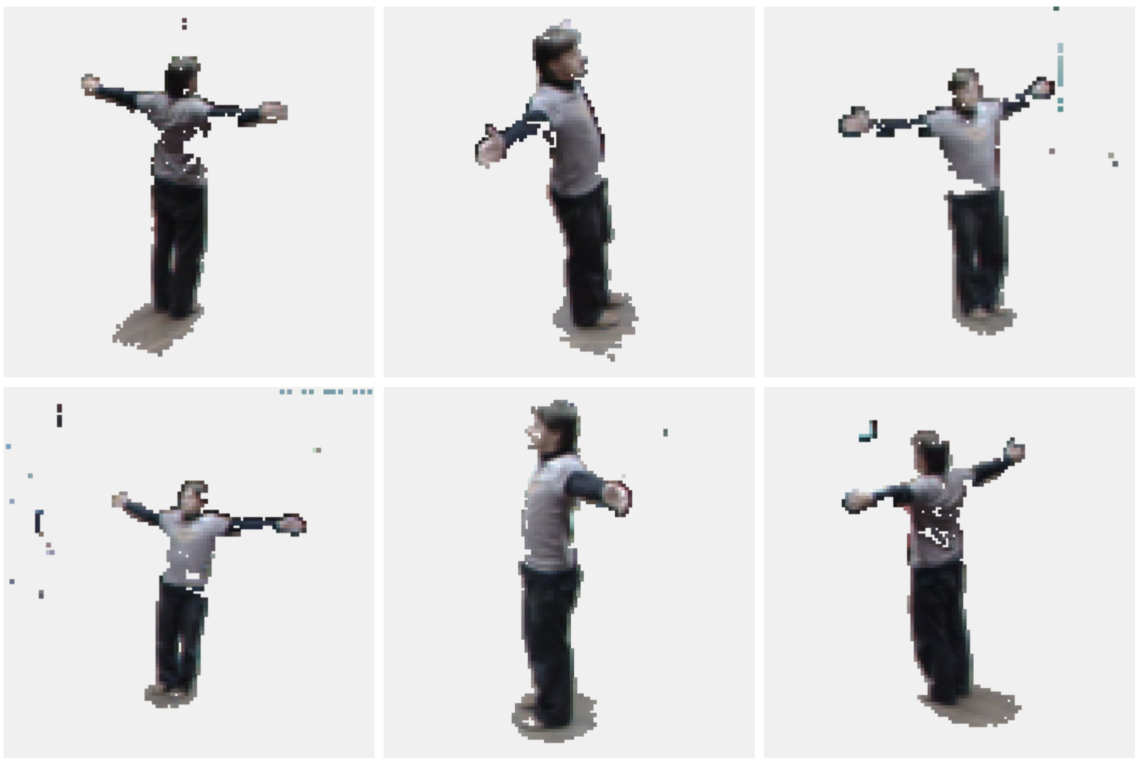


Figure 4.6: A sum of Gaussians image model created for six different views of the same pose, regular grid of cells was used. Note that only crops of the full image are shown here.

We have shown in 2.2 two ways of decomposing the camera image to a sum of Gaussians image model; a quad-tree based irregular grid approach and a uniform grid approach. The one we ended up using the most was the uniform grid, which is a simple method of dividing the image to adjacent square cells and then averaging the color in them. For each cell we assign a blob with σ equal to half the width of the cell.

Furthermore, to speed up the whole motion tracking algorithm, we decided to throw away all blobs that are unimportant for the algorithm to run - those are the blobs which don't represent the actor. We accomplish this by means of background subtraction, where the background is captured separately beforehand / afterwards.

Such a method may be detrimental to the accuracy of the tracking in that it may even discard some blobs which actually belong to the actor. But the motion tracking algorithm used proved to be robust enough to deal with most of such situations.

Another problem may arise when the lighting conditions of the scene change, this may happen for example when some of the cameras automatically adjusts exposure during the video capture, because of some strong light getting into view. Then original captured background doesn't match the scene anymore and almost all of the blobs in the image are left in the image model. This shouldn't lower the accuracy of the tracking, but its speed will definitely suffer from this.

Chapter 5

Implementation details

In the previous chapters we have described all the the pieces of the Kostilam motion capture software, starting with formulating a cost function in chapter 2, then presenting an optimization strategy in chapter 3 and finally we gave the whole picture of everything that is done in the tracking algorithm in chapter 4.

The most computationally expensive part of the proposed solution lies in computing the value of the previously formulated cost function (2.15):

$$\mathcal{E}(\Theta) = wE_{lim}(\Theta) - E(\Theta)$$

Together with also calculating its gradient. We have decided to accomplish this by using the method of automatic differentiation as presented in [26]. In this chapter we will describe the implementation of our automatic differentiation framework, the optimization approaches we have taken to make it run as fast as possible and we will sum up by showing an implementation of the cost function $\mathcal{E}(\Theta)$ in our code.

The programming language of choice for Kostilam was C++. As we will show, C++ allows for relatively straightforward implementation of automatic differentiation. Also it is considered to be a good language for writing fast and efficient programs plus there is a wide availability of drivers and libraries that can be used with it to load either recorded video sequences or live camera streams.

5.1 Automatic differentiation

As we have detailed in chapter 3, automatic differentiation computes derivatives by evaluating functions with arguments being of a special type - dual object. To compute partial derivatives of multi-dimensional functions, we have introduced the extended dual object:

- Extended dual object: $a + \sum_{i \in I} b_i d_i$, where $I = \{1, \ldots, n\}$

    1. a - real part, scalar
    2. $\mathbf{b}$ - infinitesimal part, vector of length n

Now consider that we have a function $f(x, y) : \mathbb{R}^2 \to \mathbb{R}$ written in C++, taking as its arguments standard variables of type `float`. By using a feature of the C++ language called *templates*, we can rewrite the function so that it can accept either `float` or dual object variables as its arguments. Let's call the new *templated* version $g(x, y)$:

```
float f(float x, float y)
{
  return 2.0f*x*y;
}
```

Listing 5.1: C++ function using floats

```
template<typename T>
T g(T x, T y)
{
  return T(2.0f)*x*y;
}
```

Listing 5.2: A templated C++ function

The templated functions in C++ have an interesting property in that just including them to the source files of the project doesn't actually generate any code in the final executable. Only when we use the function with some specified *template argument*, e.g. by calling `g<float>(.1f,.2f)`, the preprocessor notices this and creates an *instantiation* of the function with the template argument substituted to the function.

In this case calling the function g from listing 5.2 with the template argument being `float` will create exactly the same code as function f from listing 5.1. Now consider having an object `Dual` available, representing the extended dual object shown above. When we use it as the template argument for function g, the calculation of its gradient can be accomplished by the following code:

```
Dual x(3,{1,0});
Dual y(4,{0,1});
Dual result = g<Dual>(x,y);
return result.b;
```

Listing 5.3: Computing the gradient of $g(x, y)$ by using `Dual` objects

This piece of code will return the gradient of function $g(x, y)$ at point $(3, 4)$. The elements of the returned vector `result.b` are represent the coefficients of the dual numbers d_i introduced previously in 3.1.3. This piece of code should serve here to illustrate the method's elegance, hopefully giving some motivation for the following detailed explanations.

The strategy for using automatic differentiation in C++ should also be noticeable here - we need to write the functions we want to differentiate as templated functions, using the generic type as their arguments, intermediate values and the return value. Then after evaluating the function with a `Dual` object as the template argument, we can read off the gradient as the result's `b` vector.

5.2 Dual objects

For the above outlined strategy to work, we need to implement the `Dual` object type and *overload* all the operations that it is a part of to be able to handle computations with `Dual` objects. Such operations have already been shown in 3.1.3, to give an example here as well, let us show again how two `Dual` objects multiply:

$$(a_0 + \mathbf{b_0}^\top \mathbf{d})(a_1 + \mathbf{b_1}^\top \mathbf{d}) = (a_0 a_1) + (a_0 \mathbf{b_1} + \mathbf{b_0} a_1)^\top \mathbf{d} \tag{5.1}$$

In the following listing, we will show a part of the definition of the `Dual` object which is present in Kostilam. To make matters more complicated, we have also made the object use templates:

```
 1  template<template<typename> class V, typename T>
 2  struct Dual
 3  {
 4  public:
 5    T a;
 6    V<T> b;
 7
 8    Dual(const T& a) : a(a) {};
 9    ...
10  };
```

Listing 5.4: Definition of the `Dual` object

There are two template arguments here, T represents the precision level of math operations, we can choose either `floats` or `doubles`, while the V represents a container class for the vector $\mathbf{b}$, where we have been using a *dense* representation `Vector` or a *sparse* representation class `SparseVector`. For example to get a sparse vector representation using `floats` as the number type, we can create the `Dual` object like so: `Dual<SparseVector,float>`. We have given the choice for the container type to accommodate even large scale optimizations, where it often happens that the vector $\mathbf{b}$ is mostly empty and therefore suitable for a sparse representation.

It is important that we allow constructing the `Dual` object from a single scalar value as shown on line 8 in listing 5.4. This allows us to write code like in listing 5.2 on line 4 , where we automatically convert the type of some constant to the type given by the template argument. In such a conversion to a `Dual` object, the a member is assigned the constant's value, while the member vector $\mathbf{b}$ is left with only zeros.

There is a number of boilerplate functions and operators present in our code for manipulating the `Dual` objects, like various *constructors*, *copy constructors* or *assignment operators*. Those will be left out of this explanation and we will focus more on the important operations that are at the core of the automatic differentiation process.

5.2.1 Operations with Dual objects

Luckily, C++ allows us to overload all the standard mathematical operators it uses to also support taking `Dual` objects as the parameters. When all the necessary operations are implemented, we can start to express templated functions like 5.2 in an elegant way, oblivious to the underlying data type.

For the case of the above shown multiplication (5.1), we can overload the standard C++ operator in the following way:

```
1 template< ... class V, typename T>
2 Dual<V,T> operator*(Dual<V,T>& x, Dual<V,T>& y)
3 {
4   return Dual<V,T>(x.a*y.a, x.a*y.b + x.b*y.a);
5 }
```

Listing 5.5: Multiplication operator with `Dual` objects as arguments

Note that we omit here some parts of the code for clarity, e.g. part of the template definition on line 1 or the `const` qualifiers on line 2.

At this point we have ready all the code that is needed to step through the example given in listing 5.3. We have the function $g(x, y) = 2xy$ and we are going to compute its gradient at point $(3, 4)$. All the values that will be calculated with will be of the `Dual` type. The order of operations will go as follows:

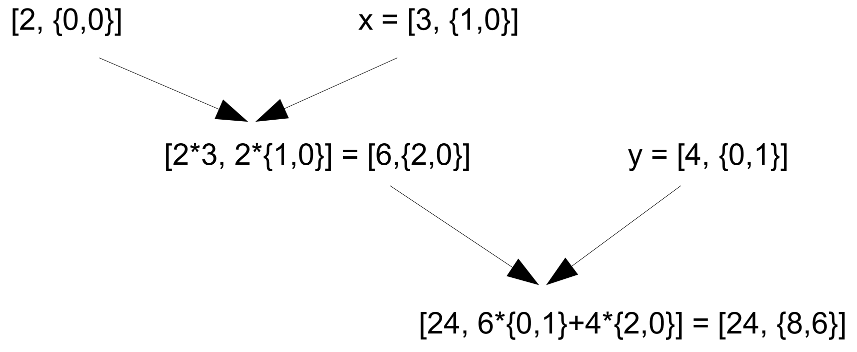


Figure 5.1: Evaluating a function using Dual objects $[a, \mathbf{b}]$ as the arguments.

The result shows that derivative with respect to x is equal to 8 and derivative with respect to y is equal to 6. This can be easily checked by symbolic differentiation. Note that the multiplication operation in C++ is *left-associative.*

In the following code listing we will show some more operators for `Dual` objects that are present in Kostilam, though not all of them as there is a lot of slightly modified variations that wouldn't be interesting here. However, even a small subset of operations as presented here is enough to already be used to evaluate a large group of functions.

```
template< ... class V, typename T>
inline Dual<V,T> operator+(Dual<V,T>& x, Dual<V,T>& y)
{
  return Dual<V,T>(x.a+y.a, x.b+y.b);
}

```

```
 7 template< ... class V, typename T>
 8 inline Dual<V,T> operator/(Dual<V,T>& x, Dual<V,T>& y)
 9 {
10   return Dual<V,T>(x.a/y.a, (x.b*y.a - x.a*y.b)/(y.a*y.a));
11 }
12
13 template< ... class V, typename T>
14 Dual<V,T> sqr(Dual<V,T>& x)
15 {
16   return Dual<V,T>(x.a*x.a, (T(2)*x.a)*x.b);
17 }
18
19 template< ... class V, typename T>
20 Dual<V,T> sin(Dual<V,T>& x)
21 {
22   return Dual<V,T>(std::sin(x.a), x.b*std::cos(x.a));
23 }
```

Listing 5.6: Examples of various operations with `Dual` objects as arguments

The last 2 rules shown in listing 5.6 are specializations of the rule for evaluating differentiable functions, which was described previously (3.21). The rule goes as follows:

$$f(a + \mathbf{b}^\top \mathbf{d}) = f(a) + f'(a)\mathbf{b}^\top \mathbf{d} \tag{5.2}$$

Note that with respect to code efficiency, it is preferable to use the above presented `sqr(x)` operation instead of writing simply `x*x`. The two approaches can be compared by the number of computations involving vectors `b` - while in the `sqr(x)` operation we perform just one multiplication of the vector `b`, when we use the multiplication operator we actually have to perform 2 multiplications and 1 addition as can be seen in the listing 5.5.

In the last example shown in code listing 5.6 we can see an interesting property of the automatic differentiation approach which we already touched upon in 3.1.4. On line 22 in the listing we can see, that evaluating the sin function with dual objects of arbitrary length only requires 2 evaluations of trigonometric functions.

5.2.2 Generic math primitives

The cost functions defined in chapters 2 and 4, either the motion tracking cost function (2.15), or the one responsible for inverse kinematics (4.1) all require the use of vectors and matrices for their calculations. For the above defined `Dual` objects to be usable in these cost functions as well, we need to define the required data types in a generic fashion.

In Kostilam, there are several objects that enable this functionality:

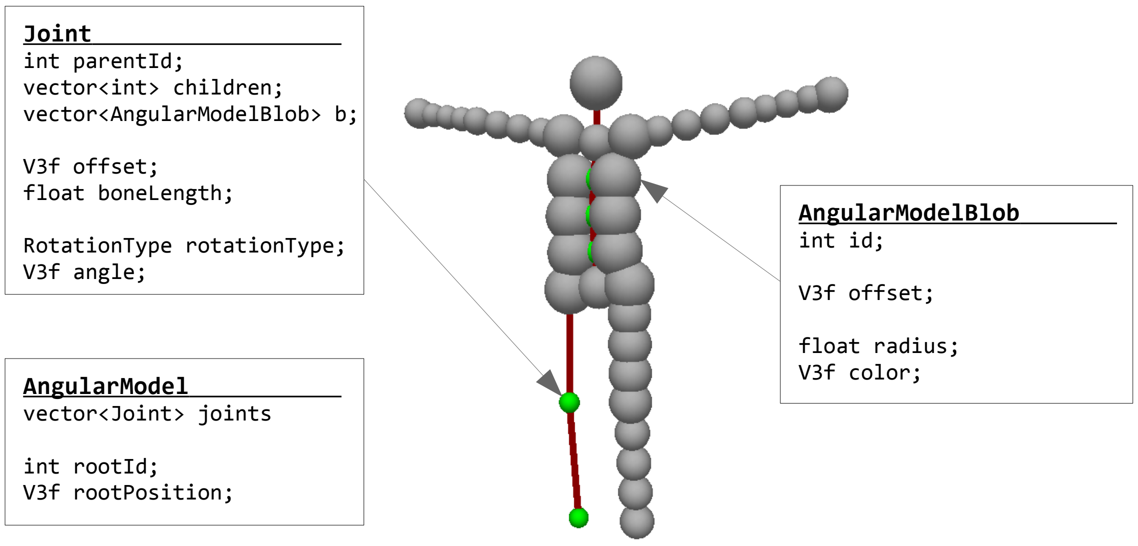
- `Vector` - dynamically allocated
- `Vec` - statically allocated

- `Mat` - statically allocated

All of these objects allow for specifying the underlying data type as a template argument. For example, one can create a vector of `floats` by calling `Vector<float>()`, or a vector of `Duals` by calling `Vector<Dual>()`. The statically allocated types also require to specify the dimensions as their template arguments, e.g. to get a 3×4 matrix of `floats` we should call `Mat<3,4,float>` in the code.

5.3 Angular model representation

Before we show an example of a cost function taken from Kostilam, let us present the data types we used to represent the sum of Gaussians body model 2.1.2. Note that we will only include the most important members of each data type here. Also in the following figure we use `V3f` as a shorthand for `Vec<3,float>`.



```
Joint
int parentId;
vector<int> children;
vector<AngularModelBlob> b;

V3f offset;
float boneLength;

RotationType rotationType;
V3f angle;
```

```
AngularModel
vector<Joint> joints

int rootId;
V3f rootPosition;
```

```
AngularModelBlob
int id;

V3f offset;

float radius;
V3f color;
```

Figure 5.2: Data types for the representation of the angular model

Members of the data types shown in the above figure are based on the model description we made in 2.1.4. Both the joints and blobs have an `offset` member to model the dimensions of the body, on top of that in joints we add an additional variable `boneLength` which is useful especially when we want to optimize the length of bones.

`AngularModel` keeps a list of its joints, each of which contains a list of its attached blobs. The joints are in a hierarchy defined by the value of the variable `parentId`, for example joint `knee_L` has as it's parent the joint `hip_L`. The `id` of a joint is set implicitly by its order in the `joints` list.

The `rotationType` determines the number of degrees of freedom of the given joint. It is used to choose an appropriate way to compute the joint's rotation matrix $\mathbf{R}(\alpha)$. Even though the number of degrees of freedom can range from 0 to 3, we always keep the `angle` variable of size 3 to simplify the implementation.

5.4 Inverse kinematics

Now that we have been exposed to the data types behind the body model, let us show our implementation of the inverse kinematics cost function (4.1) to illustrate the use of automatic differentiation in a more complex setting. We will introduce a helper *functor* object `AnglesCostIK` that will facilitate storage of several parameters:

```
template<typename T>
struct AnglesCostIK
{
  const AngularModel& model;
  const Effectors& effectors;

  std::vector< Vec<3,T> > params;

  std::vector< Mat<3,3,T> > R;
  std::vector< Vec<3,T> > t;

  void operator()(const Vector<T>& x, T& sum)
  ...
};
```

Listing 5.7: A helper object for computing the inverse kinematics cost function

Notice that the given functor is templated and the template argument T also appears in declaring the storage type of the intermediate values `params`, `R` and `t`. The vector `params` is purely for intermediate storage, while matrices `R` and `t` together represent a transformation from the joints reference frame to the world coordinates.

Calling the functor object with a set of parameters x will return the function value of the inverse kinematics cost function. The argument `x` represents the same set of parameters as the vector Θ previously introduced in 2.4. It contains the model's translational parameters and rotational parameters of all its joints stacked one after the other:

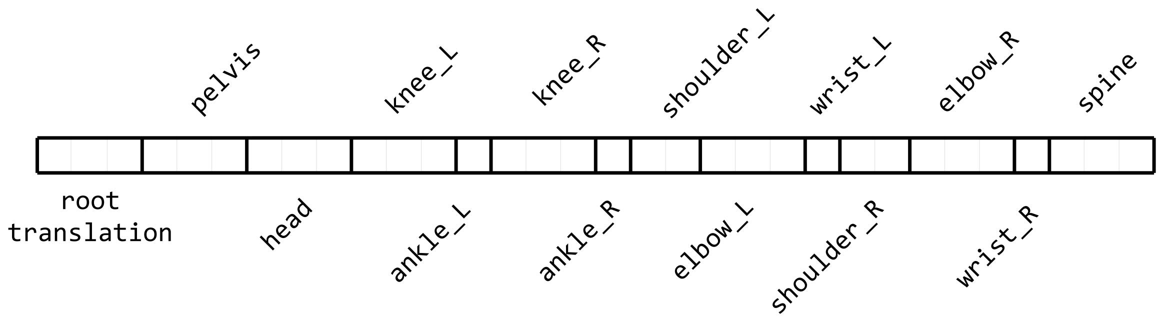


Figure 5.3: Structure of the vector of parameters x. In total, there are 32 parameters - 3 translational and 29 rotational.

Note that some joints don't appear in the above figure 5.3 as they have 0 degrees of freedom. We have made this choice for the hips joints to model the rigid pelvis bone.

Following is the implementation of the inverse kinematics cost function, for brevity, we have omitted the quadratic penalty term $E_{lim}(\Theta)$.

```
template<typename T>
void AnglesCostIK<T>::operator()(const Vector<T>& x, T& sum)
{
  params = parseParameters(x);

  Mat<3,3,T> eye = getRotation(EYE);
  calcJointFrames(model.rootId, eye, params[0]);

  for(int effector : effectors){
    Vec<3,T> dist = t[effector.target] - Vec<3,T>(effector.pos);
    sum += sqr(dist(0)) + sqr(dist(1)) + sqr(dist(2));
  }
}
```

Listing 5.8: Code calculating the inverse kinematics cost function

The function `parseParameters` splits the vector x according to the structure shown above in fig. 5.3. The resulting vector `params` contains 3-element vectors, so When there is a joint with less than 3 degrees of freedom, the remaining values are filled with zeros.

Calling the function `calcJointFrames` calculates the intermediate values of `R` and `t`, where `t` is the one that interests us as it gives us the position of the model's joints. The values are computed recursively by traversing through the skeleton hierarchy, during that, we are keeping the last calculated transformation in accumulators `R_acc` and `t_acc`:

```
template<typename T>
void AnglesCostIK<T>::calcJointFrames
        (int id, Mat<3,3,T> R_acc, Vec<3,T> t_acc)
{
  const Joint& joint = model.joints[id];
  R[id] = R_acc * getRotation(joint.rotationType, params[1+id]);
  t[id] = t_acc + R[id] *
              Vec<3,T>(joint.offset * joint.boneLength);

  for(int child : joint.children)
    calcJointFrames(child, R[id], t[id]);
}
```

Listing 5.9: Code calculating the joints' reference frames

For the functions to be usable with automatic differentiation, it is important that all the values used in the above computations are consistent in having the data type of the template argument `T`. For this reason you can see in the code at several places an explicit conversion to the data type `T`, e.g. the piece of code `Vec<3,T>(joint.offset * joint.boneLength)` converts the joint's `V3f` parameters to type `Vec<3,T>`.

Consider for a moment that the data types would be inconsistent, for example that we would use the `V3f` type for the intermediate values `t`. Then either the code wouldn't work at all when trying to assign to `t` a variable of type `Vec<3,T>`, or if we would have some implicit conversion in place, then in the case that `T` would be `Dual` we would necessarily drop the dual object's infinitesimal part **b**, losing all information about the derivatives.

5.4.1 Applying automatic differentiation

Now we have a generic implementation of the inverse kinematics cost function realized in the code by means of the functor object `AnglesCostIK`. In the following listings, we will show the difference between simple function evaluation using `floats` and automatic differentiation of the function using objects of type `Dual`.

```
float result;
Vector<float> x(32);
for(int i = 0; i < 32; i++)
{
  x[i] = ___some_value;
}

AnglesCostIK<float> cost(...);
cost(x, result);
```

Listing 5.10: Evaluation using `floats`, `result` $= f(x)$

```
Dual result;
Vector<Dual> x(32);
for(int i = 0; i < 32; i++)
{
  x[i].a = ___some_value;
  x[i].b[i] = 1;
}
AnglesCostIK<Dual> cost(...);
cost(x, result);
```

Listing 5.11: Automatic differentiation, `result.a` $= f(x)$, `result.b` $= \nabla f(x)$

Some parts of the code are left out to make the point short, e.g. the `Dual` object would require further specifying a template argument to choose one of the implementations or the `AnglesCostIK` object would require some constructor parameters.

In both examples we prepare a vector of parameters `x` and fill it with some values - this is the point where we want to evaluate or differentiate the function, usually it comes from a previous state of the model. The difference is in the data types; in listing 5.10 we us `floats` so then all the calculations in the cost function will be done in standard `float` arithmetic and as a `result` we will get a single function value.

On the other hand when we specify the arguments as objects of type `Dual` we are going to perform the automatic differentiation process instead of a simple function evaluation. The `Dual` arguments are prepared in the same fashion as shown in the theoretical example 3.1.4, that is the real part `a` gets assigned the single scalar input and the infinitesimal part `b` is a vector of zeros with a 1 denoting the argument's position in the input.

Now all the calculations in the cost function will involve `Dual` objects as their operands, also giving the `result` as a `Dual` object. Its real part (a) is going to be the original function evaluation and the infinitesimal part (**b**) will give us the function's gradient at the specified point as has been explained in the chapter 3 in the theory part of this work.

5.4.2 Dual objects initialization

Returning to the way the `Dual` objects are initialized, another way of thinking of such a setup is when we think of the infinitesimal part `b` as the gradient of the function that resulted in the given `Dual` object.

At the beginning when we have only the input parameters in a list, such functions are of the form $f(\mathbf{x}) = x_i$. Consider the input parameters $\mathbf{x} = (x_1, x_2, \ldots, x_{32})$ and then a gradient of one such simple function:

$$f(\mathbf{x}) = x_2 \qquad \frac{\partial f}{\partial \mathbf{x}} = (0, 1, 0, \ldots, 0) \tag{5.3}$$

If we would leave the part `b` of the `Dual` objects only zeros, then we would get no derivatives in the end. Also setting the `b` vectors in any other way than presented above wouldn't in the end produce the desired gradient.

5.5 Motion tracking

In the previous section we have shown the inverse kinematics cost function in full detail, hopefully giving a good idea of how we use the automatic differentiation approach in Kostilam. Giving such a detailed description of our motion tracking cost function (2.15) here would be far too long and probably not much beneficial for the reader, so we will show only a simplified overview of the implementation.

Similarly to the previous case, we will define a functor object to help us with storing all the intermediate variables needed in the cost function:

```
template<typename T>
struct AnglesCostTracking
{
  const AngularModel& model;
  const Views& views;

  std::vector< Vec<3,T> > params;
  std::vector< Mat<3,3,T> > R;
  std::vector< Vec<3,T> > t;

  std::vector< T > perViewSum;
  std::vector< Vec<3, T> > projectedBlob;

  void operator()(const Vector<T>& x, T& sum)
  ...
};
```

Listing 5.12: A helper object for computing the motion tracking cost function

The members `params`, `R` and `t` have the same meaning and are computed identically as in the inverse kinematics cost function. On top of that we have `perViewSum` for storing contributions from different cameras and then `projectedBlob` for storage of the projected body model.

Following will be the code for the cost function itself, but note that it is not exactly the same as it appears in Kostilam, we have stripped it here to its bare essentials - some of the constructions shown here would have to be written on more lines of code in Kostilam and somewhere we would introduce more code to make the program run faster. We have tried to capture the general idea as well as possible.

```
template<typename T>
void AnglesCostTracking<T>::operator()(const Vector<T>& x, T& sum)
{
  params = parseParams(x);
  calcJointFrames(model.rootId, EYE, params[0]);

  sum += limitsPenalty(x);

  for (int k = 0; k < views.size(); k++)
  {
```

In the first part, we split the parameter vector `x` and compute the transformations $[\mathbf{R}\ \mathbf{t}]$ for each joint. New from the previous function we have the computation of the quadratic penalty term $E_{lim}(\Theta)$ (2.14) and we start a loop over the different camera views.

The following part is calculated for each camera view; in all of them we step through the model's blobs and project them using the given view's camera matrix $\mathbf{P}$, this process has been described already in section 2.3.1; first we project the blob centers to homogeneous coordinates and then divide by the third element to get the image coordinates. The new size of the projected blobs is calculated using the camera's focal length `f`.

```
    for (int i = 0; i < model.joints.size(); i++)
    {
      for (AngularModelBlob& blob : model.joints[i].blobs)
      {
        // blob position in world coordinates
        Vec<3, T> blobPos = t[i] + R[i] * Vec<3, T>(blob.offset);

        // blob projected to image
        Vec<3, T> pb = views[k].camera.P * Vec<4, T>(blobPos, 1);

        // conversion from homogeneous coordinates
        projectedBlob[blob.id].x = pb.x / pb.z;
        projectedBlob[blob.id].y = pb.y / pb.z;

        // calculating the new scale of the blob
        projectedBlob[blob.id].r = (views[k].camera.f * blob.r) / pb.z;
      }
    }
```

We will follow by calculating the similarity E_{ij} 2.8 of the blobs in the image with all the projected blobs of our model. This involves calculating the color similarity $d(\mathbf{c}_i, \mathbf{c}_j)$ and the integrated overlap of the two Gaussians representing the blobs:

```
29     for (ImageBlob& imageBlob : views[k].imageBlobs)
30     {
31       T imageBlobSum = 0;
32
33       for (Joint& joint : model.joints)
34       {
35         for (AngularModelBlob& blob : joint.blobs)
36         {
37           const T blobOverlap =
38             calcBlobOverlap(imageBlob, projectedBlob[blob.id]);
39           const float colorSimilarity = d(imageBlob.c, blob.c);
40
41           imageBlobSum += T(colorSimilarity) * blobOverlap;
42         }
43       }
44
45       float imageBlobSelfOverlap = E(imageBlob, imageBlob);
46       perViewSum[k] -= min(imageBlobSum, T(imageBlobSelfOverlap));
47     }
```

The contributions of all the projected blobs are summed up for each blob in the image in the variable `imageBlobSum`, then we compare the value with the maximum allowable contribution for a single image blob `imageBlobSelfOverlap`. The minimum of the two is then accumulated in the variable `perViewSum`. Notice on line 55 that we actually use subtraction here so by minimizing the final value, we will be maximizing the blob overlaps. The equations that were implemented at this step were described in section 2.3.3.

Now we have done all the necessary calculations in the individual camera images, we can close the loop starting at line 9 and sum the images' contributions to the final cost function:

```
48   } // end of the (for each view) loop
49
50   for (int k = 0; k < views.size(); k++)
51     sum += perViewSum[k] / views[k].sumBlobsOverlaps;
52 }
```

We are weighting the contributions of the images by the term $E(\mathcal{K}_l, \mathcal{K}_l)$ shown before in section 2.4, the term should balance images where the model appears large with images where the model is small.

5.5.1 Summary

We have shown a simplified implementation of the cost function $\mathcal{E}(\Theta)$ (2.15) responsible for finding the most probable pose of the model. Computing its gradient is done by means of automatic differentiation, the same code as in listing 5.11 can be used here as well, only we replace the previous cost function with the new `AnglesCostTracking`.

We then use the L-BFGS optimization algorithm with this cost function plugged in to finally track the model. We described the choice of this algorithm previously in section 3.2, basically it approximates a second-order optimization method but still it only requires us to supply only the function's gradient. We used an open-source library [23] to run the method in our code.

5.6 Optimization techniques

In order to run our tracking algorithm as fast as possible, we have focused mainly on efficient calculations with `Dual` objects because that is where most of the time is spent, in fact, more than 80% of the algorithm's running time is spent on operations of `Dual` objects or their life-cycle management.

The reason why those operations take such a significant portion of the used time is that they are actually operations with vectors (infinitesimal part **b**) and as such require heavy movement and allocation/deallocation of data. But it is also a fact that at the beginning of the computations, the vectors are mostly filled with zeros (5.4.2), only a single non-zero element is present. Those get added up, however, and this is a point where we can see a major difference between the original Kostilam model 2.1.1 which relied on translational parameters only and the new angular model 2.1.2.

Consider how position of a single model blob is influenced by different model parameters:

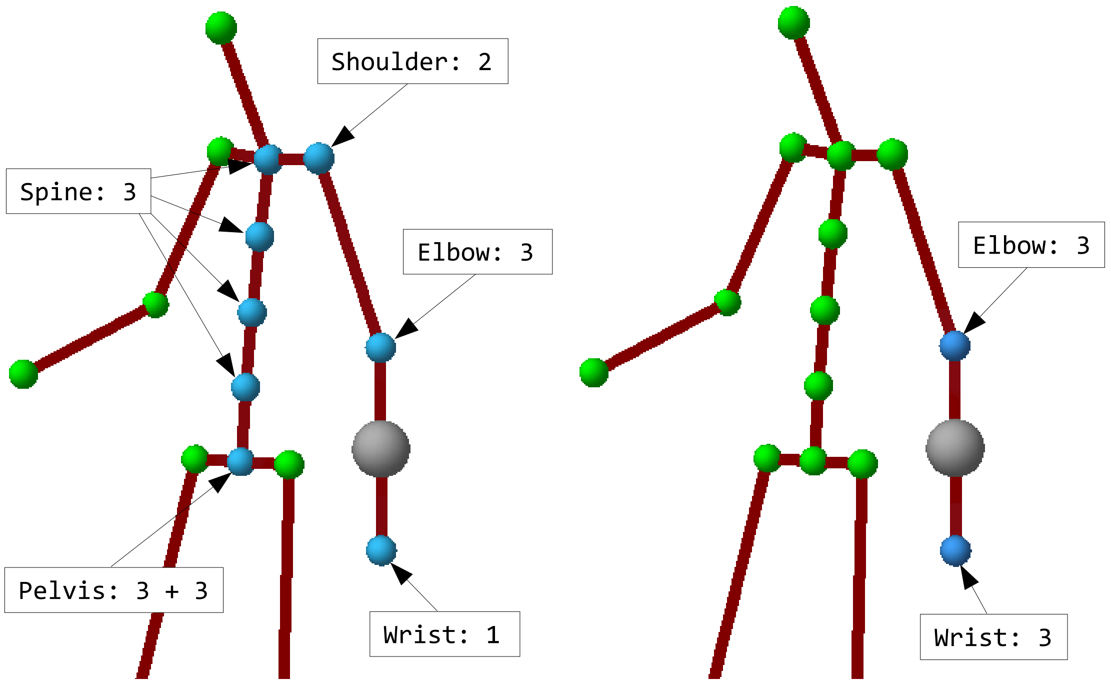


Figure 5.4: Parameters influencing a blob on the left hand, joints responsible for its movement are in blue. Angular model parameters (left), positional model parameters (right).

In the positional model the blob only moves when its adjacent joints move, on the other hand in the angular model any joint up to the root in the skeleton hierarchy can influence the blob's position. So while in the positional model the position of a blob can have up to 6 non-zero values in its `b` part, for the angular model this can rise up to 15 non-zero values.

Hopefully, this has shown the importance that effective `Dual` / vector operations have in our tracking algorithm. In the following part we will show our implementation approach.

5.6.1 Sparse vector

In our first implementation of the cost function (5.5), we have used as the underlying data structure for the `Dual` objects a standard dense vector representation - `Vector`. This implementation was extremely slow, it was taking tens of seconds to compute one frame.

We have observed the distribution of sizes of the `Dual` objects appearing in the computations over the first few frames, the size meaning how many non-zero values are actually present in the vectors:

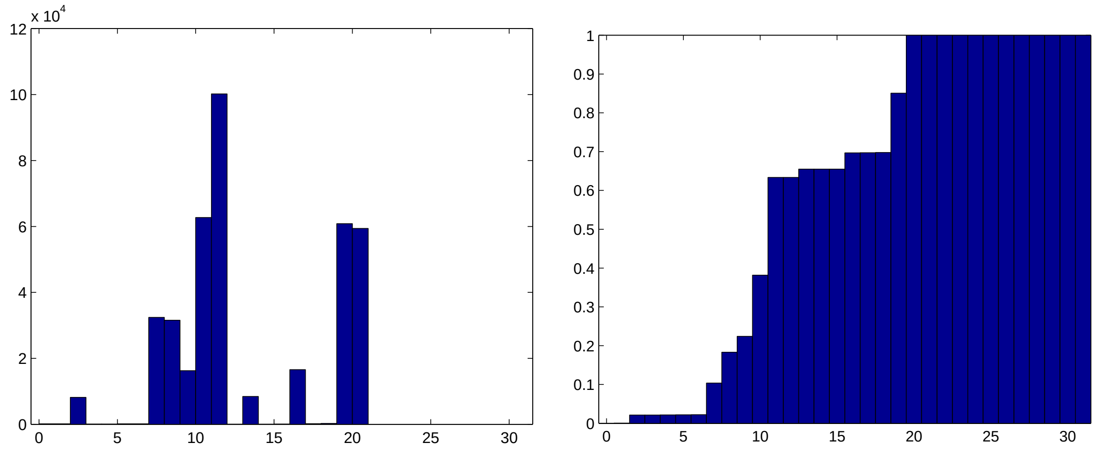


Figure 5.5: Size of `Dual` objects in calculations is given on the x-axis, no. of occurences (left), percentage of operations with `Dual` objects having less than x non-zero values (right).

Seeing how as much as 70% of all the operations with `Dual` objects only calculates with less than half-full vectors, we have decided to bring to the table a sparse representation of vector: `SparseVector`. Implementing sparse operations to power the automatic differentiation framework is not uncommon in literature, e.g. [26] or [37] use it with success. Our implementation contains the following four members:

- `size` - number of non-zero values in the vector
- `mask` - a bit mask having 1 at the position of the data
- `ind` - an array of indices, represents the position of the data
- `val` - an array of non-zero data values

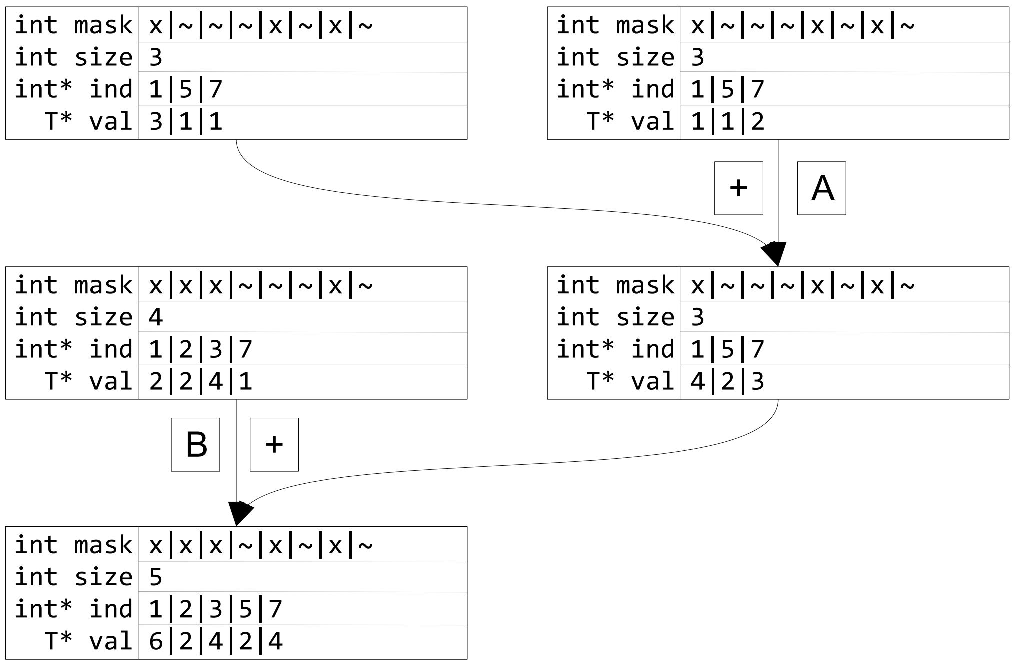


Figure 5.6: `SparseVector` representation with two modes of addition: fast(A), merge(B).

We keep with each vector a bit mask representing all the positions of the vector's data in a single integer. Lucky for us, our human body model has just 32 parameters in total. When we are adding two sparse vectors together we first compare their masks and if they match exactly, we can proceed to fast addition (A in fig. 5.6) otherwise we have to initiate a slower "merge" addition (B in fig. 5.6).

Good thing is that the fast additions actually constitute more than 95% of all the additions combined. One more thing to add is that when we want to perform larger optimizations - with more parameters than 32 - we usually disable the fast addition completely. The algorithm for the general addition is similar to the *merge* part of the *merge sort* algorithm, we have taken inspiration for our implementation in [37].

Also notice that the data member `val` uses a template for its type, so the sparse vector could store `floats` or `doubles`, even `Duals` if the need would arise.

5.6.2 Memory management

You may have noticed some dangerously looking pointers in the sparse vector representation, here we will describe our rationale behind using them and also the way we manage their life-cycle. It is being recommended these days to use the standard `vector` containers in modern C++ almost everywhere some memory allocation is necessary [34], in our case, however, we are dealing with lots of small allocations and deallocations happening constantly every time some new temporary variable is created.

Unfortunately there is a lot of such temporary variables appearing in our code at the moment, consider the simple example of the multiplication operator for `Dual` objects:

```
Dual<V,T> operator*(Dual<V,T>& x, Dual<V,T>& y)
{
  return Dual<V,T>(x.a*y.a, x.a*y.b + x.b*y.a);
}
```

Listing 5.13: Multiplication operator with `Dual` objects as arguments

Not taking into account some compiler optimizations, in general we have to allocate two sparse vectors for both operands `x.a*y.b` and `x.b*y.a` and finally there is one more allocation for the result. The two intermediate values then have to be deallocated. Considering that this process happens within every multiplication of `Duals`, any improvement here can be very significant.

Under such heavy load, the standard C++ operators `new` and `delete` can cause *memory fragmentation*, ultimately leading to slow response or fluctuations in performance. We have therefore decided to write a custom memory management scheme, that would cope with our demands.

5.6.2.1 Fixed-sized pool allocator

We strived to find an algorithm that would be very simple, but still faster than the default C++ memory management. Such requirements necessarily led us to a compromise. We found something called the *fixed-size pool* algorithm, which is being used often in videogames for realtime purposes [20] and it allows to allocate and deallocate chunks of memory of fixed size in very little time. The data structure for the memory manager is remarkably simple:

```
class MemoryManager {

  struct FreeStore {
    FreeStore *next;
  };

  int maxMemory;
  FreeStore* freeStoreHead;
  ...
};
```

Listing 5.14: Custom memory manager class inspired by [31]

Indeed, it is a simple linked-list. However, there is a trick in the algorithm that makes the simple linked-list a powerful memory manager. Let's say we want to be allocating sparse vectors of maximum number of elements `maxMemory`. First we allocate a continuous chunk of memory and we construct a linked-list in it with its elements representing free space for new vectors. In the next step we will allocate some sparse vectors and finally we will perform a deallocation. Notice that the data of the vectors will overwrite the space of the previous pointer:

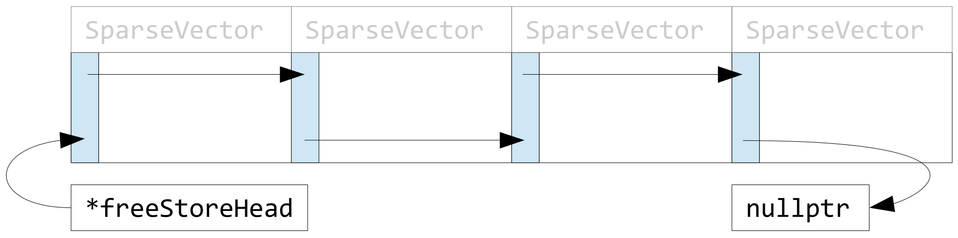


Figure 5.7: Memory manager **initialization**, here we have space for 4 sparse vectors. The blue rectangles represent pointers to the next free space.

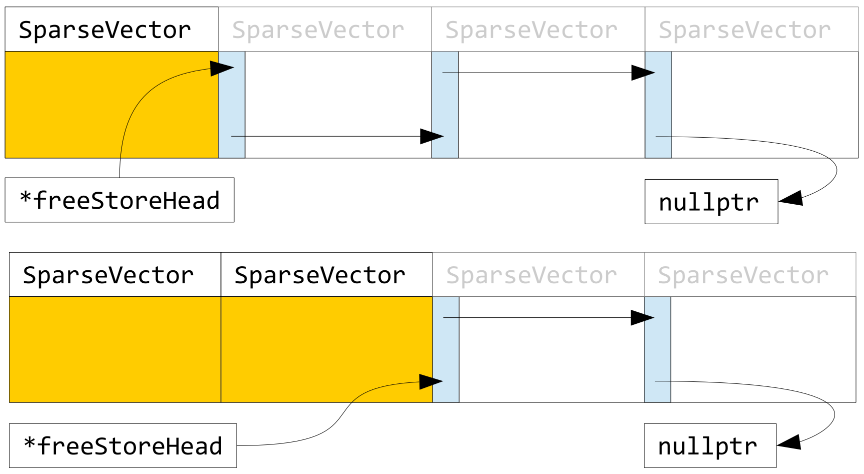


Figure 5.8: A series of two memory **allocations**, the space for them is taken from the beginning of the linked-list.

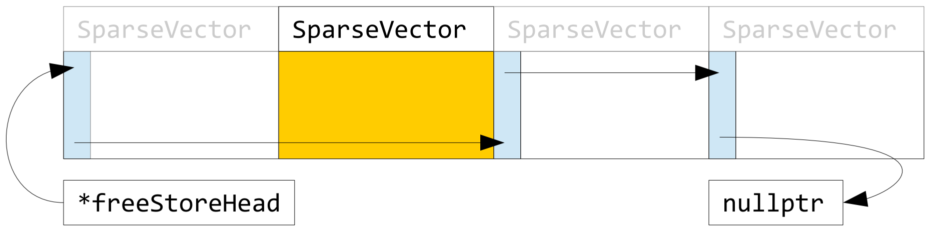


Figure 5.9: Performing **deallocation** of the first allocated sparse vector. The reclaimed space is returned to the beginning of the linked-list.

Introducing the custom memory manager indeed made the cost function computation faster, against the standard `new` and `delete` operators the difference was around 50%. And that is even though we actually allocate with our memory manager more memory than is needed - majority of the sparse vector will never get filled up completely. In this case the compromise on space was beneficial for run time.

In Kostilam we allocate the default memory manager to hold up to a 1000 sparse vectors but still, the maximum simultaneously allocated vectors took only around 600 chunks of memory.

5.6.3 Parallelization

We enabled parallelism for the computation of the main cost function 5.5 relatively simply by using an OpenMP [1] directive:

```
#pragma omp parallel for
for(int k = 0; k < views.size(); k++)
{
    ...
}
```

Listing 5.15: OpenMP enabled parallelism

This way each of the views should be calculated on a different thread. The only catch was that we had to instantiate one new memory manager for each such thread, because more threads accessing one memory manager wouldn't be safe. When we implemented this and ran the parallel algorithm on a processor with support for at least 6 simultaneous threads, we got a speedup of about 150% against the single-thread implementation.

5.6.4 Summary

On top of these optimizations, we used the new feature of C++11 called *move semantics* [34] and implemented both the *move constructor* and *move assignment* operators for `SparseVectors` and `Duals`. That way at least some of the temporary variables appearing in all calculations are avoided. Putting that together with the sparse vector implementation, custom memory manager and parallelization lead to a solid speedup of the tracking algorithm - where with the naive approach we had to wait more than 10 seconds for each frame, now we can reach up to 3 frames per second with the optimized algorithm.

Chapter 6

Testing

In this chapter we will show images from the sequences we managed to track with our algorithm. We have been testing our motion capture algorithm on a setup of 6 PlayStation Eye cameras which have a resolution of 640x480 pixels and a framerate of 60 frames per second. All of the tracked sequences are present on the enclosed CD, the results probably look a bit better in the video as the movement can hide some imperfections.

6.1 Various sequences

6.1.1 Sequence blue_exercise

We have successfully tracked 4000 frames of this sequence, up to its end. The whole process was finished without any manual corrections. When you look at frame samples on the following pages, on both sides you will be able to see the same pose, on the left in the model space, on the right in the image space.

6.1.2 Sequence red_exercise

We have managed to track 3268 frames of this sequence, up to its end. There was a minor problem, however - at one point the actor got out of the area visible by all the cameras, which caused the model to behave erratically. We have solved this by programatically turning off the contribution of the views from which the actor is at least half-absent. This turned out to work in this particular case.

6.1.3 Sequence take1

We have tracked this sequence successfully without needing any corrections. This sequence is 1280 frames long.

6.1.4 Sequence take3

We have tracked this sequence with success, not needing any manual corrections. This sequence is 1190 frames long.

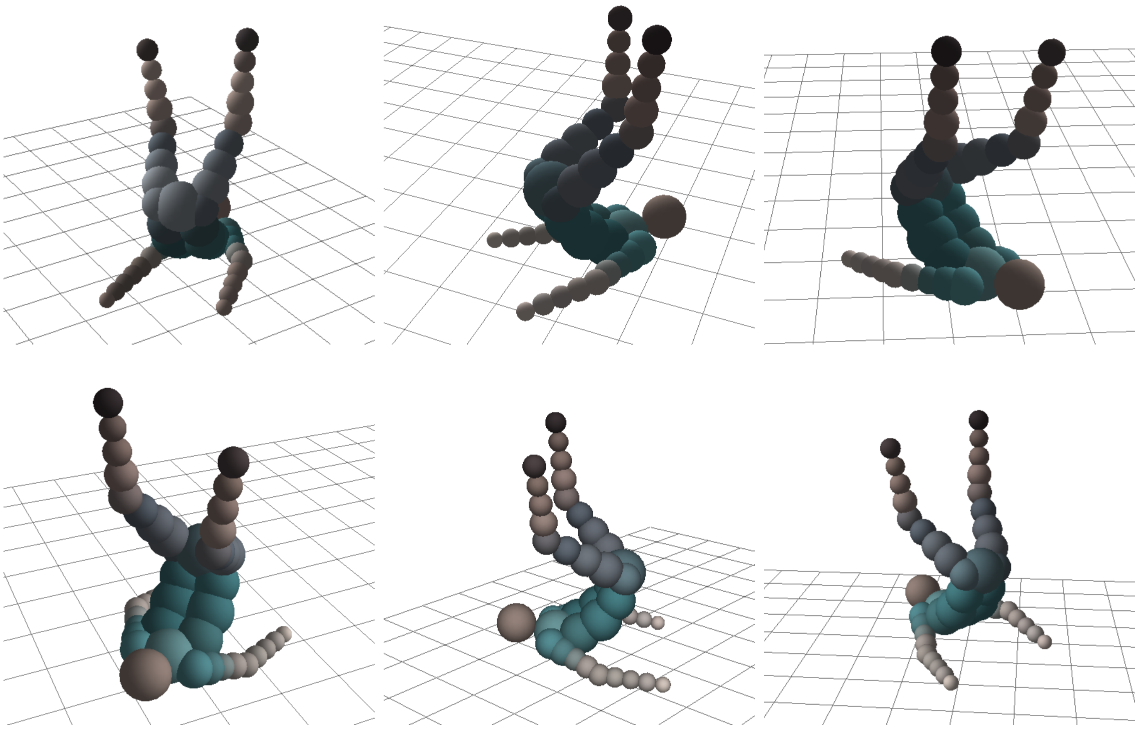


Figure 6.1: Frame 2091 of the blue_exercise sequence, model view

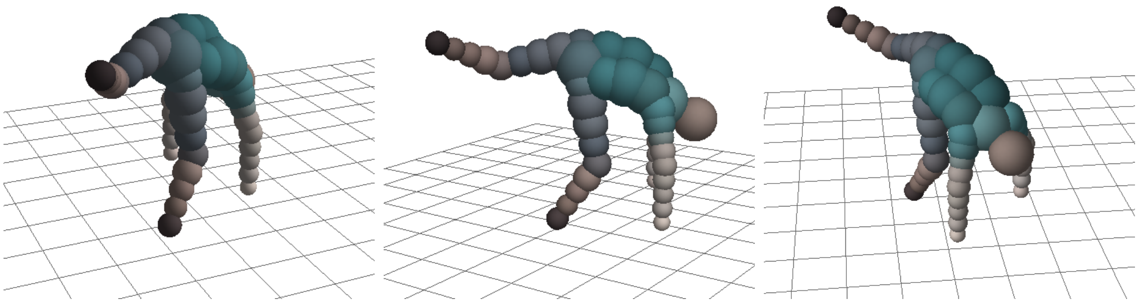


Figure 6.2: Frame 1078 of the blue_exercise sequence, model view

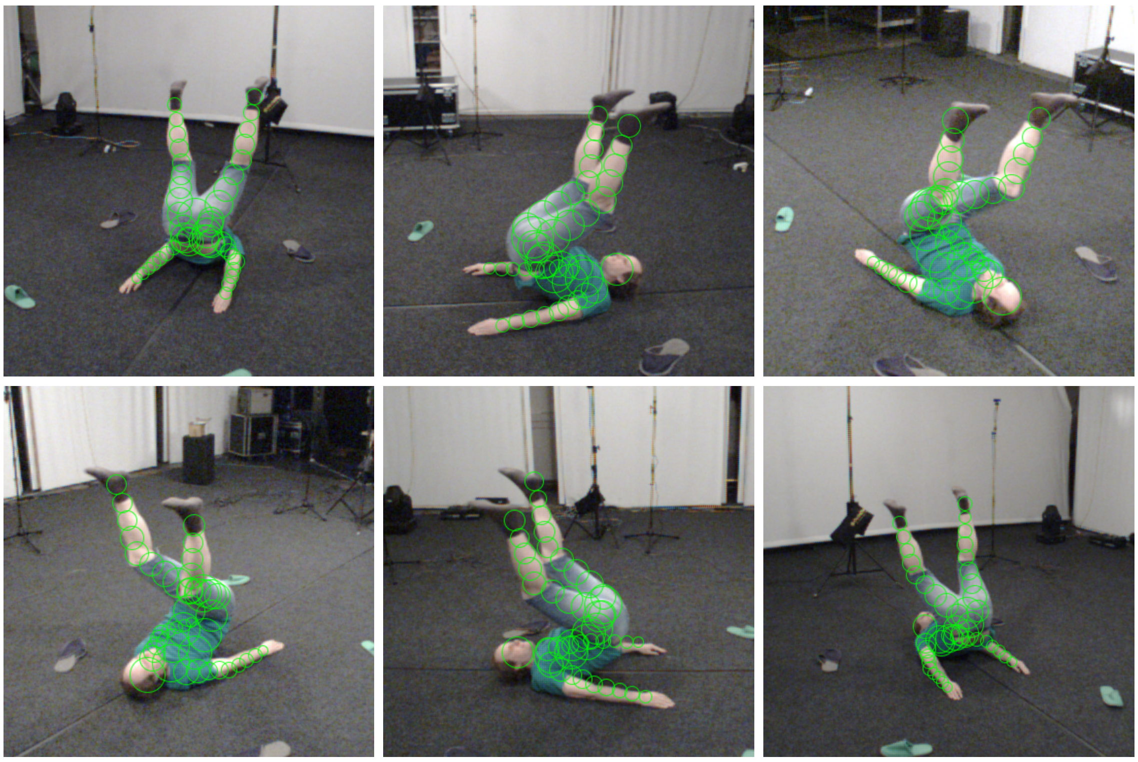


Figure 6.3: Frame 2091 of the blue_exercise sequence

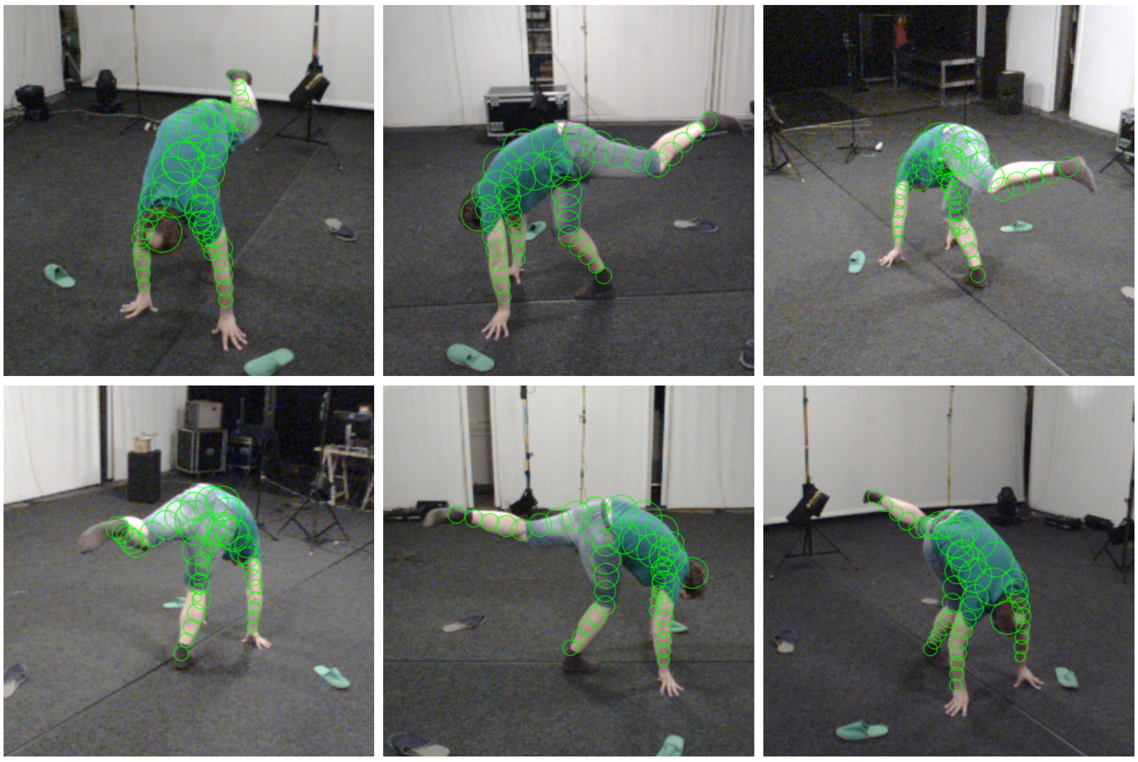


Figure 6.4: Frame 1078 of the blue_exercise sequence

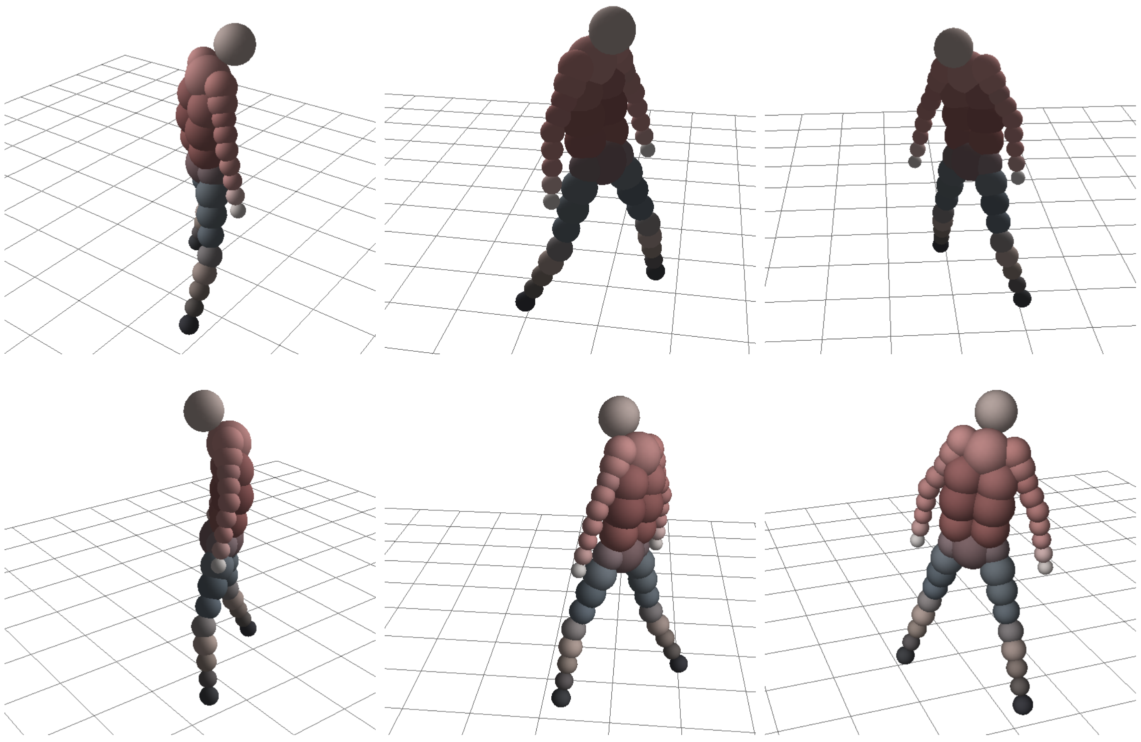


Figure 6.5: Frame 672 of the red_exercise sequence, model view

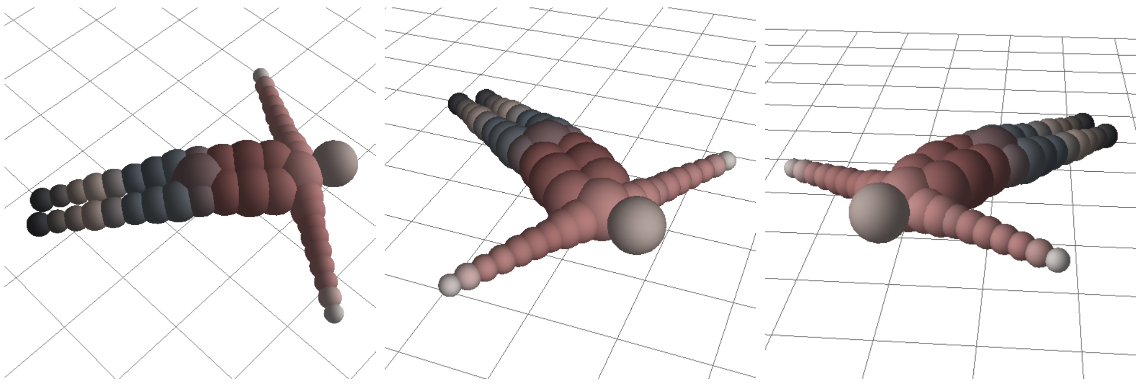


Figure 6.6: Frame 1300 of the red_exercise sequence, model view

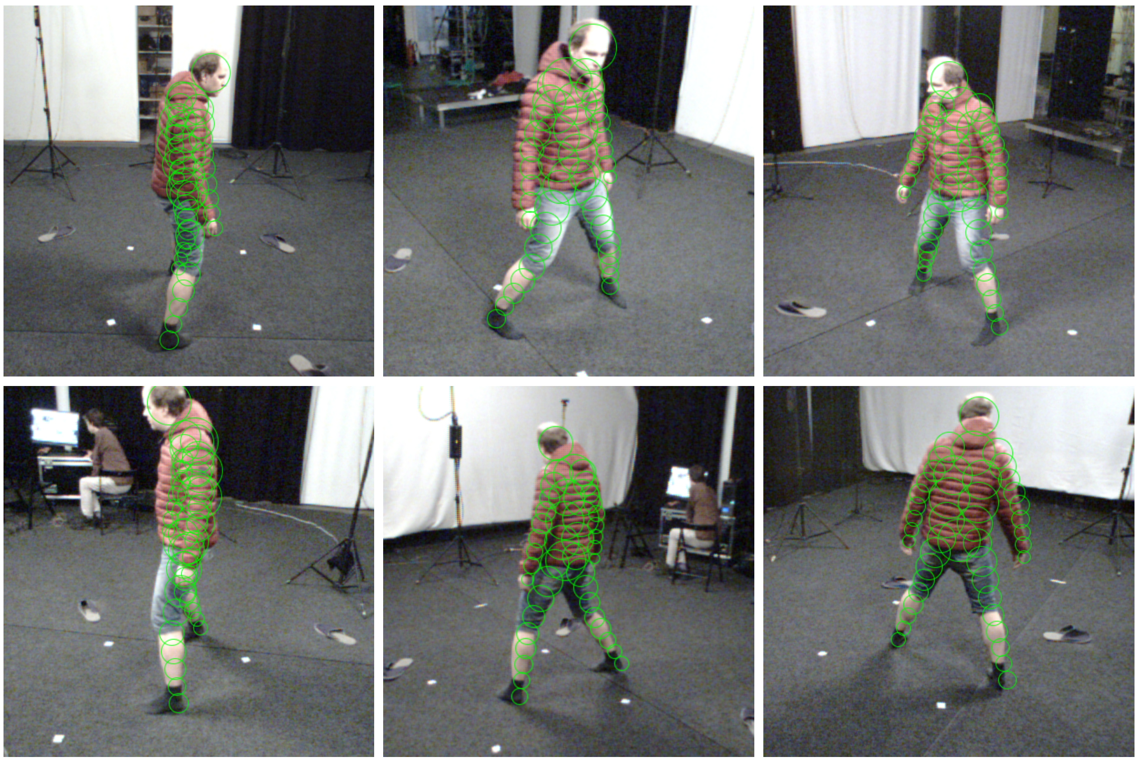


Figure 6.7: Frame 672 of the red_exercise sequence

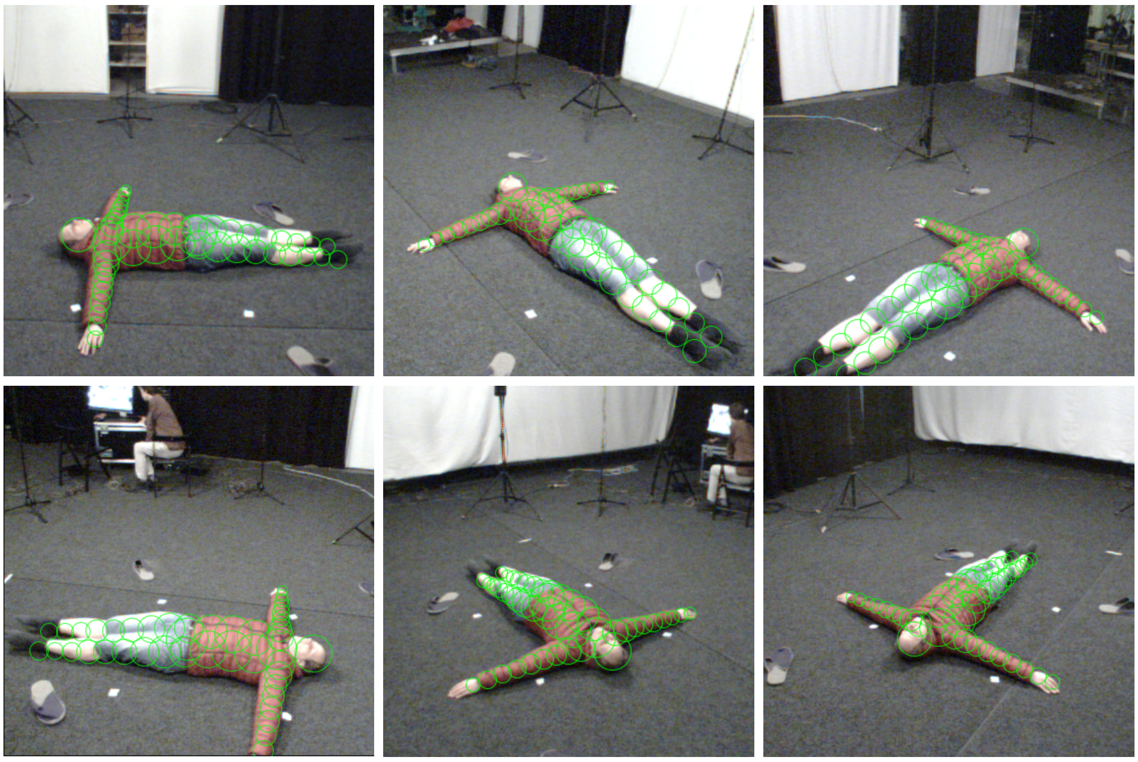


Figure 6.8: Frame 1300 of the red_exercise sequence

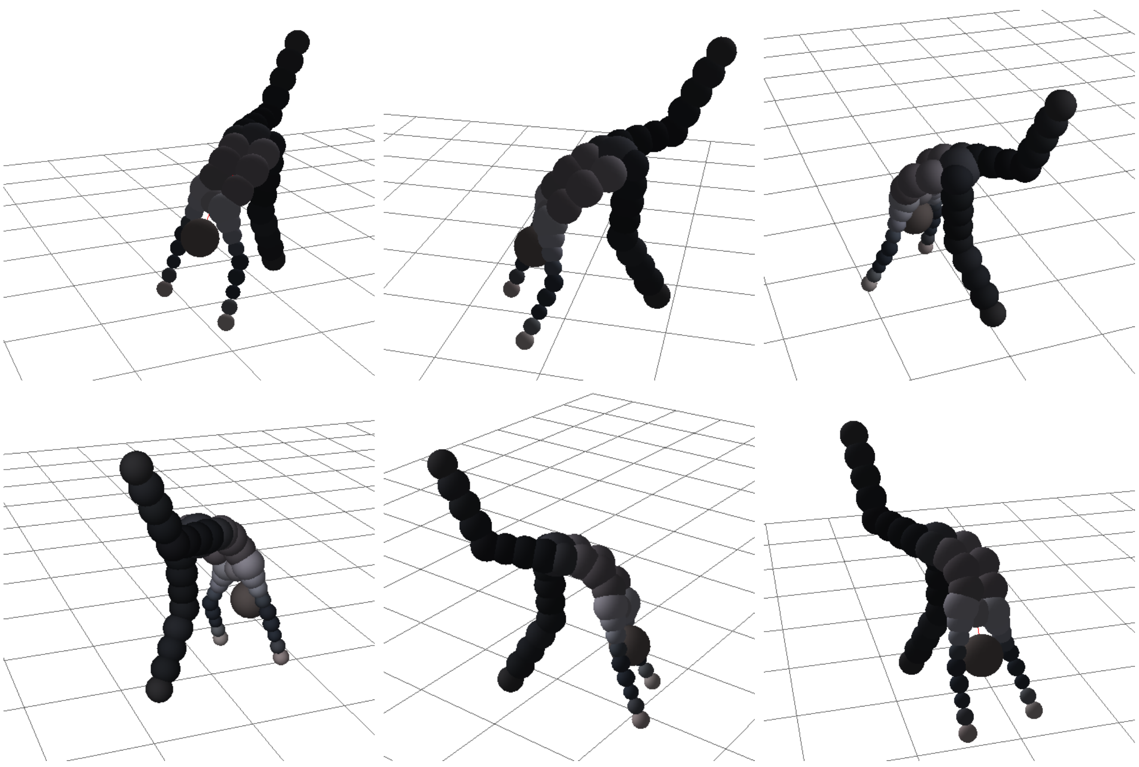


Figure 6.9: Frame 390 of the take1 sequence, model view

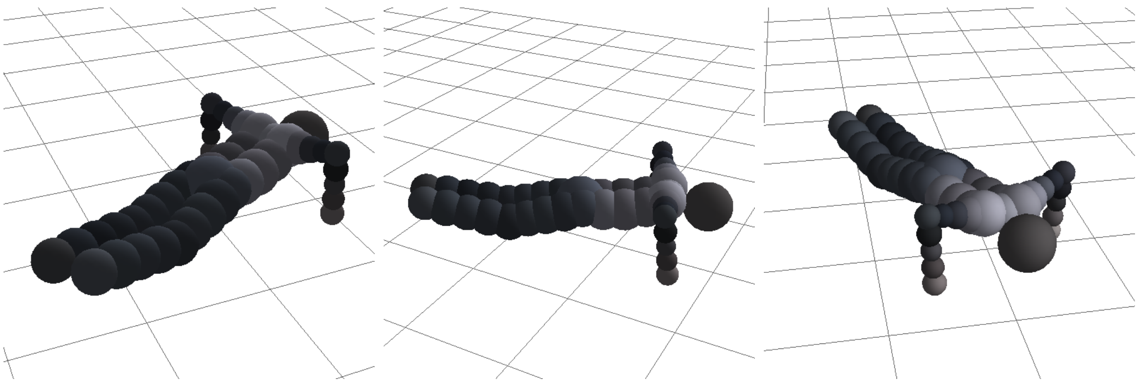


Figure 6.10: Frame 913 of the take1 sequence, model view

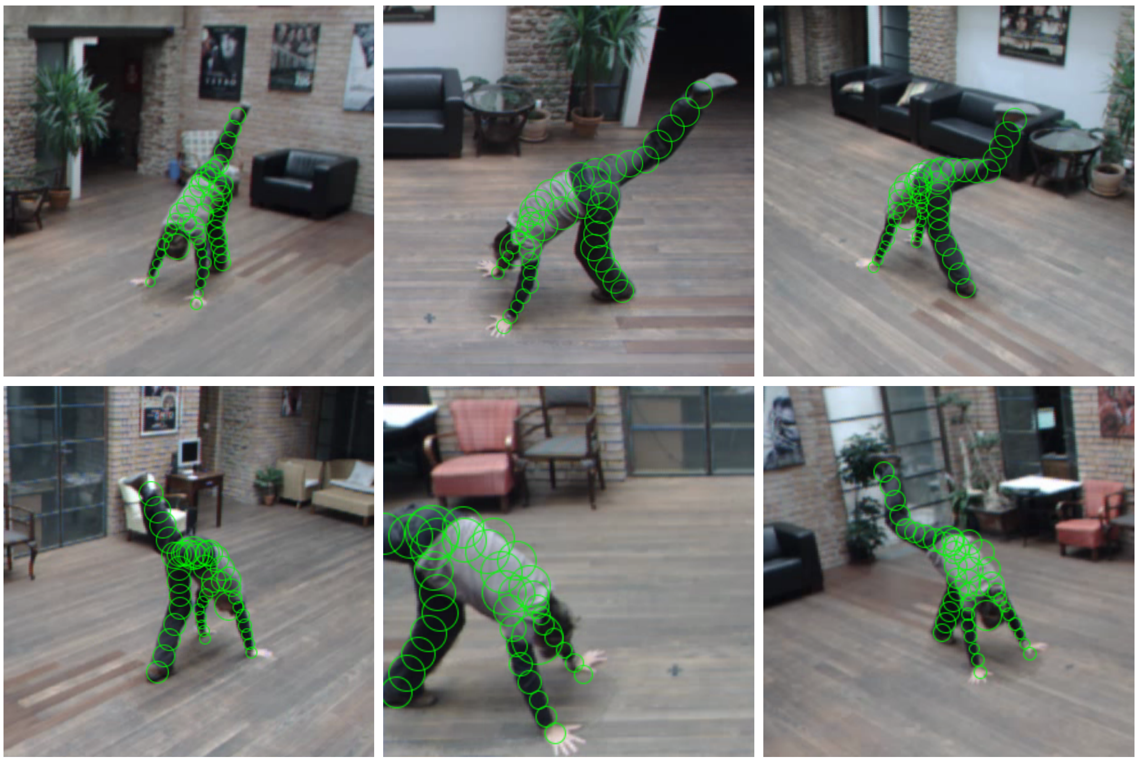


Figure 6.11: Frame 390 of the take1 sequence

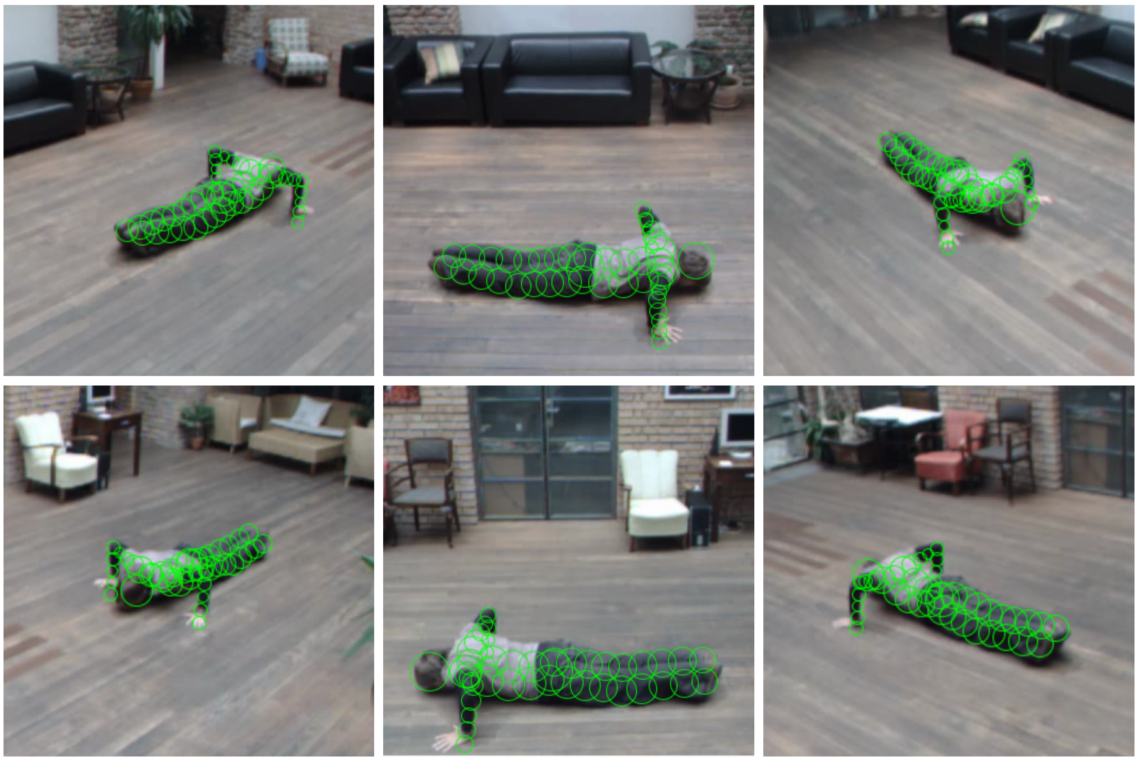


Figure 6.12: Frame 913 of the take1 sequence

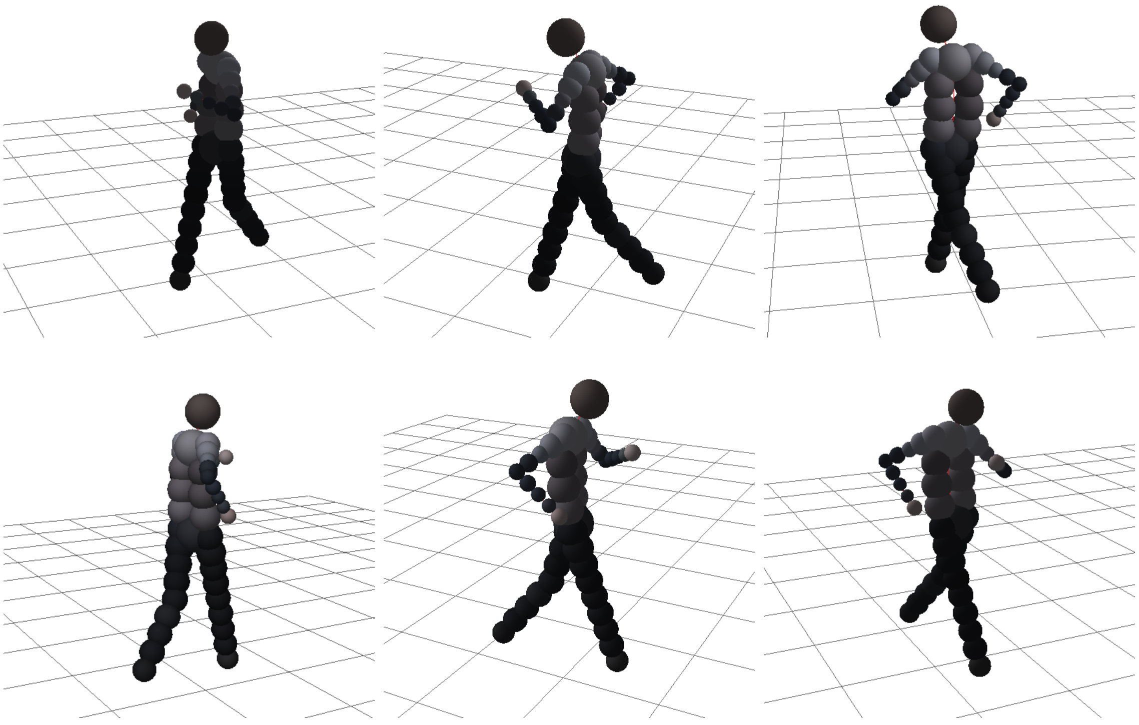


Figure 6.13: Frame 674 of the take3 sequence, model view

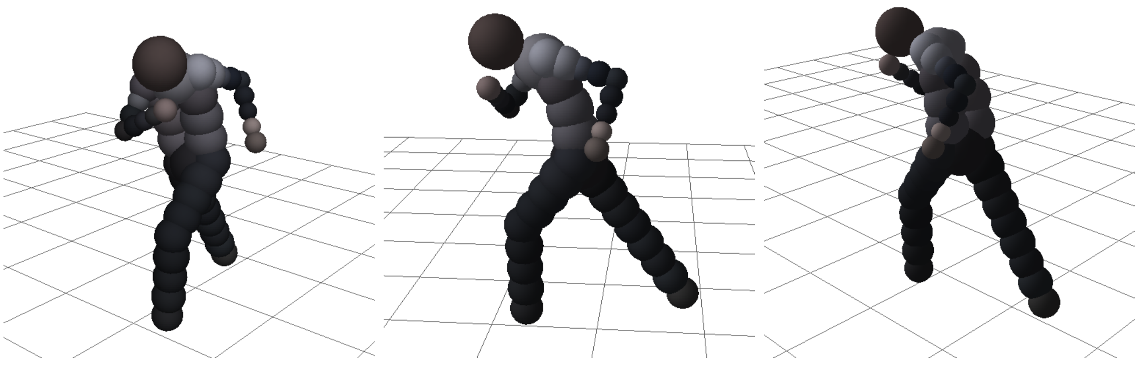


Figure 6.14: Frame 824 of the take3 sequence, model view

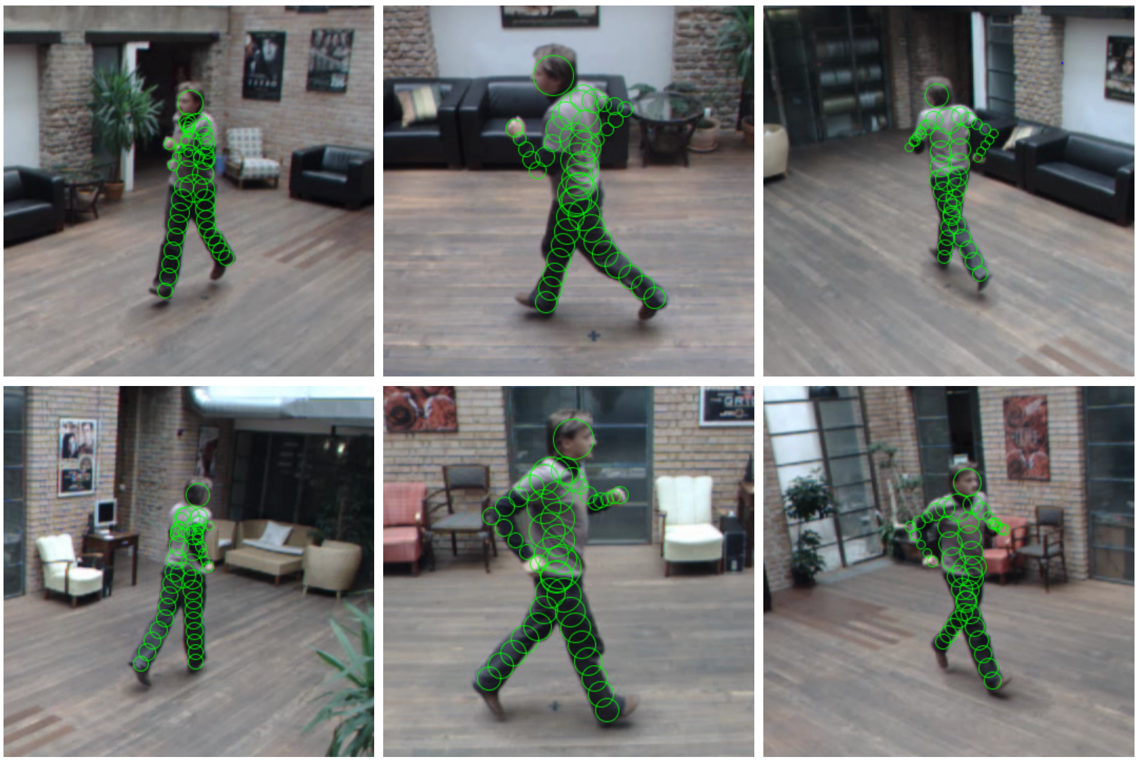


Figure 6.15: Frame 674 of the take3 sequence

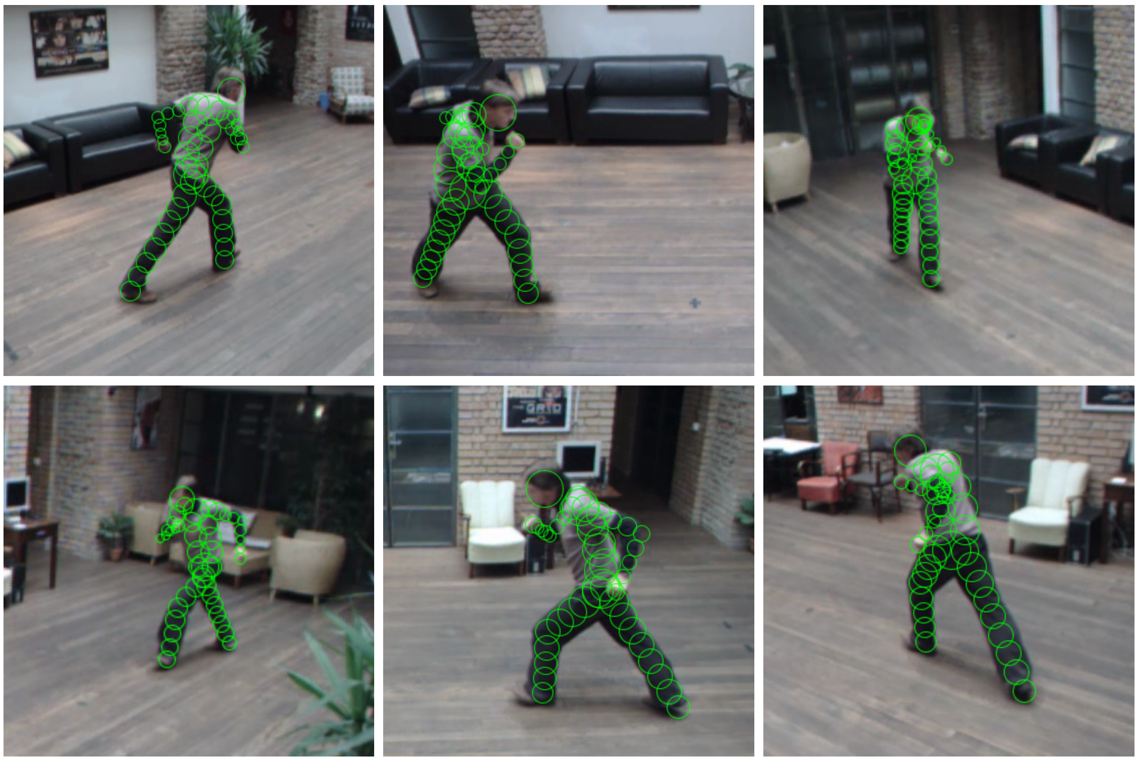


Figure 6.16: Frame 824 of the take3 sequence

Chapter 7

Conclusion

We have shown in this work the development of a markerless motion capture system, from its key concepts, to the little details that make it work. In chapter 2 we have shown the core idea we decided to base our software on - the cost function 2.15 first published in [33] by Stoll. Together with it went the development of a sum of Gaussians body model 2.2 that was designed to be able to represent different actors as well as possible.

Continuing on to chapter 3, that is where we introduced theory related to automatic differentiation, based on an article [26] by Piponi. We decided to use the automatic differentiation approach to minimize the above mentioned motion tracking cost function.

In chapter 4 we have shown all the supporting tasks that need to be implemented in order to make good use of our chosen tracking approach, e.g. finding a good initial solution or calibrating the body model to a specific actor.

In the longest chapter 5 we have shown the hard implementation details of our automatic differentiation framework, hopefully giving the reader a good idea how it works and perhaps an inspiration to implement it in their own existing solutions. Also in this chapter we have described the optimization techniques we used to speed up Kostilam as much as possible.

Finally in chapter 6 we have shown some results of tracking people's motion through recorded sequences.

Unfortunately we haven't managed to develop a solution for the light adaptation problem, partly due to the complexity of the whole task, this leaves perhaps as a topic for future work.

This work gave me opportunities to play with some very interesting materials, during our work we had visited two different capture studios, we had access to some fast computers where we could run our Kostilam software, which was always a joy to see, for that I'm grateful.

Bibliography

[1] ARB. OpenMP. <`openmp.org/`>, 2015.

[2] Autodesk. 3ds Max. <`www.autodesk.com/products/3ds-max/overview`>, 2015.

[3] Autodesk. Maya. <`www.autodesk.com/products/maya/overview`>, 2015.

[4] Jonathan Blow. Inverse kinematics with quaternion joint limits. <`number-none.com/product/`>, 2002.

[5] Joseph Bray. Markerless based human motion capture: a survey. 2001.

[6] C. G. Broyden. The convergence of a class of double-rank minimization algorithms 1. general considerations. *IMA Journal of Applied Mathematics*, 6(1):76–90, 1970.

[7] Samuel R. Buss. Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods. Technical report, IEEE Journal of Robotics and Automation, 2004.

[8] Motion capture society. The history and current state of motion capture. <`www.motioncapturesociety.com/resources/industry-history`>, 2015.

[9] Harry H. Cheng. Programming with dual numbers and its applications in mechanisms design. *Engineering with Computers*, 10(4):212–229, 1994.

[10] Shinko Y Cheng and Mohan M Trivedi. Articulated human body pose inference from voxel data using a kinematically constrained gaussian mixture model. *Proc. CVPR 2nd Workshop on Evaluation of Articulated Human Motion and Pose Estimation*, 55, 2007.

[11] W. K. Clifford. Preliminary sketch of biquaternions. In *Proc. London Math. Soc. 4*, pages 381–395, 1873.

[12] CTU. Center for machine perception. <`cmp.felk.cvut.cz/`>, 2015.

[13] R. Fletcher. A new approach to variable metric algorithms. *The Computer Journal*, 13(3):317–322, 1970.

[14] D. M. Gavrila and L. S. Davis. 3-d model-based tracking of humans in action: a multi-view approach. pages 73–80, 1996.

[15] D. Goldfarb. A family of variable metric updates derived by variational means. *Mathematics of Computation*, 24:23–26, 1970.

[16] K. Grauman, G. Shakhnarovich, and T. Darrell. Inferring 3d structure with a statistical image-based shape model. In *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, pages 641–647 vol.1, Oct 2003.

[17] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Second Edition*. SIAM e-books. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 2008.

[18] Ian P. Howard. *Perceiving in Depth, Volume 3: Other Mechanisms of Depth Perception*. Oxford Psychology. Oxford University Press, USA, 2012.

[19] M. E. Jerrell. Function minimization and automatic differentiation using c++. *SIGPLAN Not.*, 24(10):169–173, September 1989.

[20] Ben Kenwright. Fast efficient fixed-size memory pool. Technical report, IARIA, 2012.

[21] Brice Michoud, Erwan Guillou, and Saïda Bouakaz. Real-time and markerless 3d human motion capture using multiple views. In Ahmed Elgammal, Bodo Rosenhahn, and Reinhard Klette, editors, *Human Motion – Understanding, Modeling, Capture and Animation*, volume 4814 of *Lecture Notes in Computer Science*, pages 88–103. Springer Berlin Heidelberg, 2007.

[22] G. Mori, Xiaofeng Ren, A.A. Efros, and J. Malik. Recovering human body configurations: combining segmentation and recognition. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 2, pages II–326–II–333 Vol.2, June 2004.

[23] Naoaki Okazaki and Jorge Nocedal. libLBFGS. <`www.chokkan.org/software/liblbfgs/`>, 2015.

[24] Optitrack. Motion capture technology. <`www.optitrack.com/`>, 2015.

[25] J. M. Papakonstantinou and Rice University. *Historical Development of the BFGS Secant Method and Its Characterization Properties*. Rice University, 2009.

[26] Dan Piponi. Automatic differentiation, C++ templates, and photogrammetry. *Journal of graphics, GPU, and game tools*, 9(4):41–55, 2004.

[27] Ronald Poppe. Vision-based human motion analysis: An overview. *Comput. Vis. Image Underst.*, 108(1-2):4–18, October 2007.

[28] Qualysis. Motion capture technology. <`www.qualisys.com/`>, 2015.

[29] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *IEEE International Conference on Neural Networks*, pages 586–591, 1993.

[30] Jochen Schmidt and Heinrich Niemann. Using quaternions for parametrizing 3-d rotations in unconstrained nonlinear optimization. In *Vision, modeling and visualization 2001*, pages 399–406. AKA/IOS Press, 2001.

[31] Arpan Sen and Rahul K. Kardam. Building your own memory manager for c/c++ projects. <`http://www.ibm.com/developerworks/aix/tutorials/au-memorymanager/`>, August 2014.

[32] David F. Shanno. Conditioning of quasi-newton methods for function minimization. *Mathematics of Computation*, 24:647–656, 1970.

[33] Carsten Stoll, Nils Hasler, Juergen Gall, Hans-Peter Seidel, and Christian Theobalt. Fast articulated motion tracking using a sums of gaussians body model. In *Proceedings of the 2011 International Conference on Computer Vision*, pages 951–958, 2011.

[34] Bjarne Stroustrup. C++11 FAQ. <`www.stroustrup.com/C++11FAQ.html`>, 2015.

[35] Tomás Svoboda, Daniel Martinec, and Tomás Pajdla. A convenient multi-camera self-calibration for virtual environments, 2005.

[36] Daniel Sýkora, John Dingliana, and Steven Collins. As-rigid-as-possible image registration for hand-drawn cartoon animations. In *Proceedgings of International Symposium on Non-photorealistic Animation and Rendering*, pages 25–33, 2009.

[37] Jukka I. Toivanen and Raino A. E. Makinen. Implementation of sparse forward mode automatic differentiation with application to electromagnetic shape optimization. *Optimization Methods and Software*, 26(4-5):601–616, 2011.

[38] Kentaro Toyama and Andrew Blake. Probabilistic tracking with exemplars in a metric space. *International Journal of Computer Vision*, 48(1):9–19, 2002.

[39] Xiaolin Wei, Peizhao Zhang, and Jinxiang Chai. Accurate realtime full-body motion capture using a single depth camera. *ACM Trans. Graph.*, 31(6):188:1–188:12, November 2012.

Appendix A

Figures

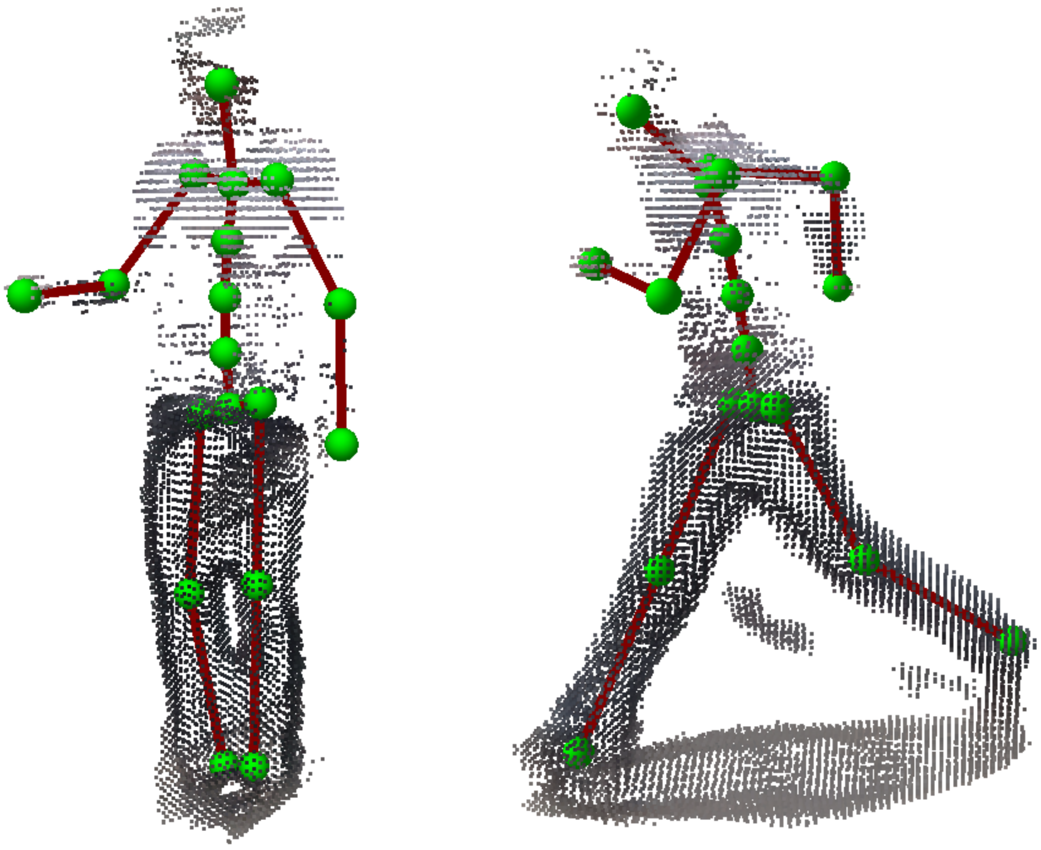


Figure A.1: Point clouds and skeletons

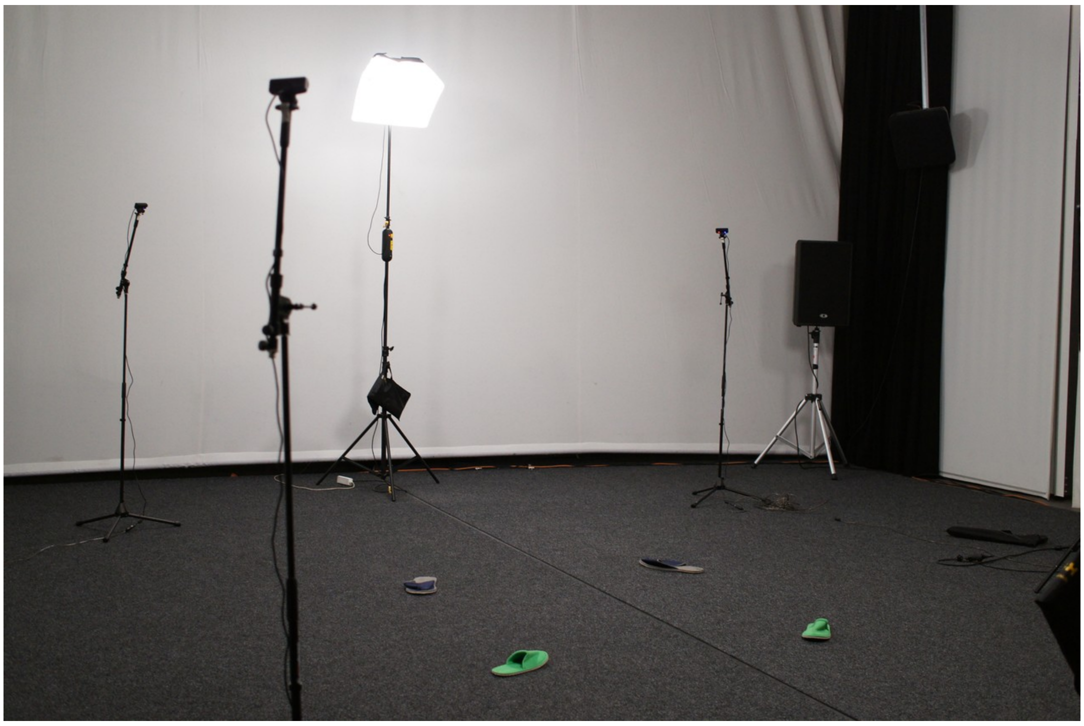


Figure A.2: IIM studio

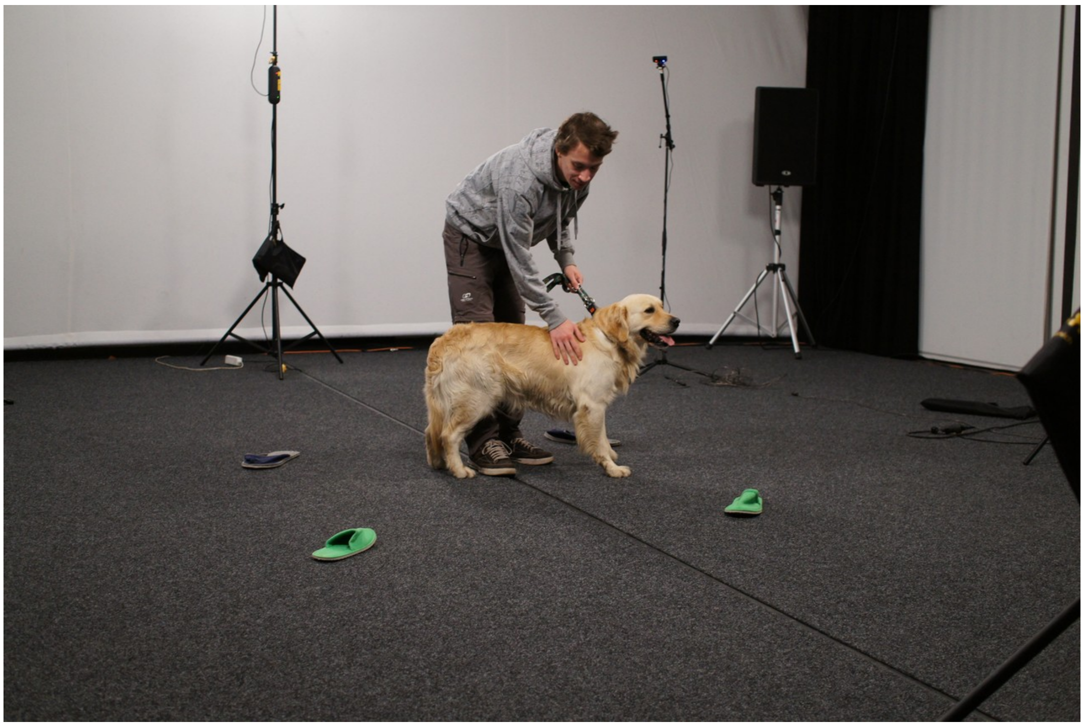


Figure A.3: IIM studio, the dog experiment

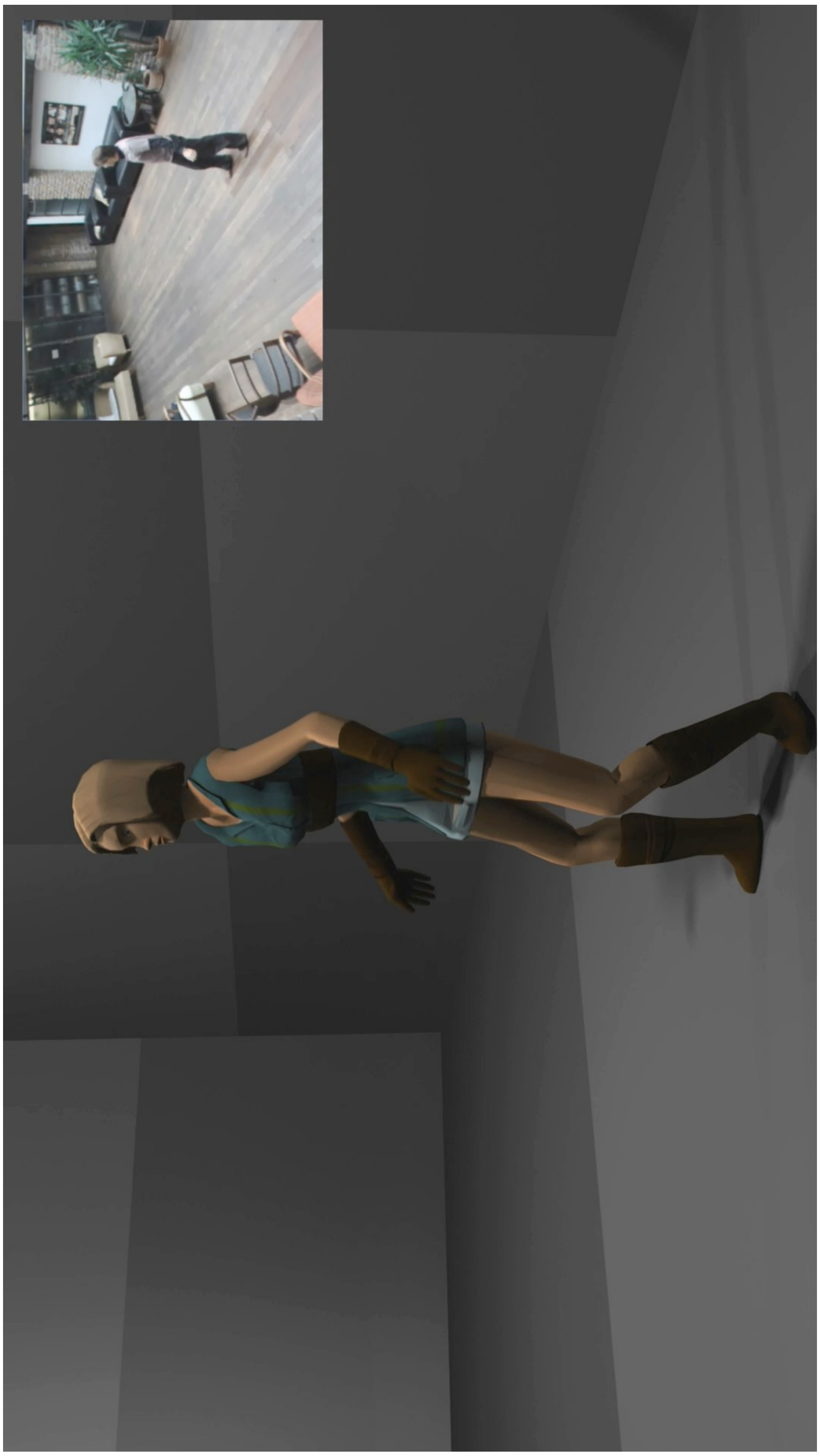


Figure A.4: take3 exported to Maya, courtesy of Radek Smetana

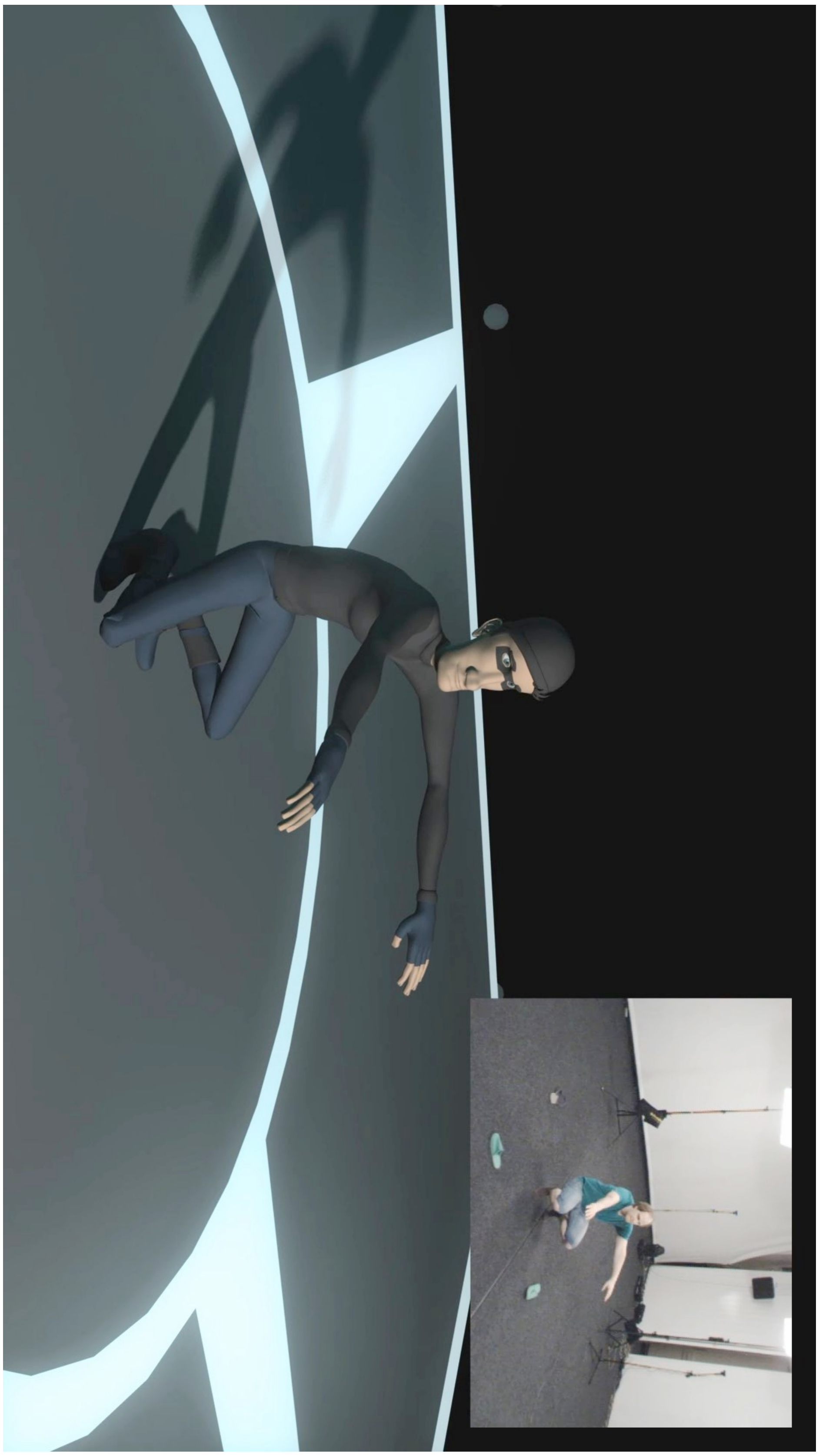


Figure A.5: take11 exported to Maya, courtesy of Radek Smetana

Appendix B

Contents of CD

Unfortunately, we cannot enclose the Kostilam software here, as it is targeted to be a commercial product. If you have inquiries about the software, please feel free to contact the author of this thesis or the supervisor.

Folder `video`:

- `take1.mov`
- `take3.mov`
- `iim-blue-exercise.mp4`
- `iim-red-exercise.mp4`

Folder `pdf`:

- `krupka-mt.pdf`