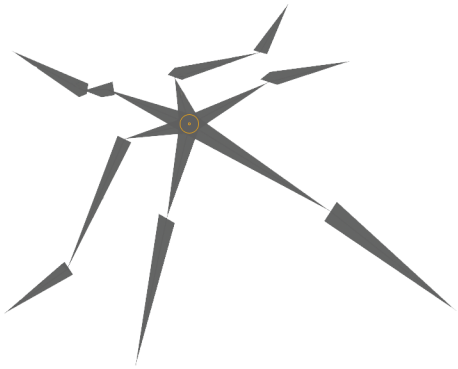
figurate

An application for rigid skeleton modelling



Jonathan Cheung

Supervised by
Andrew Davison
Duncan Gilles

Imperial College London

June 19th, 2012

Abstract

Animating a 3D character by hand can be a long and difficult process. Technologies such as motion capture exist that can detect pose of an actor with accuracy but these systems are typically too expensive and difficult to set up for the average user.

In this project we propose a novel solution using computer vision techniques to building a system that can detect the pose of a real-world object. The system is easy to use and does not require anything other than an inexpensive web-cam to operate. Some of the uses for the system include being used as an electronic drawing aid for artists or for electronic stop-motion animation.

Acknowledgements

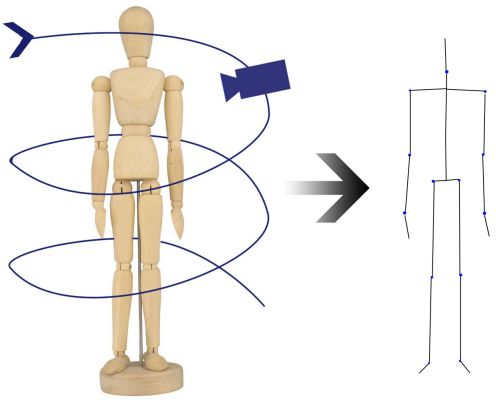
First and foremost, I would like to thank Dr Andrew Davison for accepting my project proposal and for his continued guidance and support throughout this project. I would also like to thank Duncan Gillies for his inital feedback during the early stages of the project. Finally I would like to thank my family and friends for their continued support.

Contents

1 Introduction

1.1. Objective

The objective of this project is to develop a system that utilises computer vision techniques to estimate the pose of a real-world poseable model. For instance we can rig a humanoid skeleton from a drawing mannequin of the same proportions. The system will be able to determine the pose of the real-world object based on a number of markers placed on the object.



In order for the system to accurately estimate the pose of the model, the estimations of 3D positions of the markers must be as accurate as possible. Furthermore the system must be able to pose the model as best as it can given imprecise or missing information about the 3D positions of parts of the object due to noise or occlusion. Computer vision techniques will be used to detect, recognise and estimate the location of the markers in 3D space. The system should be designed to be able to work with any cheap off the shelf web-cam.

Users should also be able to configure their own model definitions in order to estimate pose of custom models that may not be humanoid in nature. The model definition

is a specification that describes the structure of the model that the user wants to run through the system. For instance in a humanoid model it would contain information about the length of specific bones, which bones join together, the degrees of movement of joints that join bones and the angle of movement of the joints. An intuitive GUI will also be created that will allow the user to create this model definition.

The scope of this project is to create the above system and to make it as easy to use as possible for the end-user, a fully fledged UI will be designed and the application should provide real-time feedback throughout the process of estimating the pose of the skeleton model.

1.2. Motivation

The motivation behind this project is to make the act of posing animation skeletons by hand easier. The primary uses of this system is to provide a better drawing aid for artists and for fast compositing of static scenes for 3D artists.

Many traditional artists use image references to aid them in drawing. Drawing humanoid subjects can be especially difficult for amateur artists. Traditional scaled human mannequins can be used to help but the system described in report improves this as it can be scaled to any body type and can be saved electronically. A key advantage is that a static viewpoint of the model can also be saved as an image meaning that the artist can be referring to the exact same image and viewpoint every-time. This can be very useful if the user is using digital photo editing tools such as Photoshop where they can directly import the final result.

Existing solutions are either too complex or expensive for the majority of artists. Hand-animating a 3D skeletal rig can be a long and cumbersome process and requires training and expertise to get a good result. Existing motion capture systems used to rig 3D skeletons are typically very expensive and require extensive calibration and clean-up to achieve good results.

The main area of research in this project is in computer vision. Recently there has been a lot of interest in computer vision due to the ubiquity of cameras on many computing devices such as smart phones and tablets as well as the advances in computer processing power allowing many computer vision techniques to run in real-time. Taking the computer vision approach to the problem of rigging the skeleton automates a lot of the work required and makes it faster than traditional approaches such as hand-rigging, furthermore it is more user friendly as the user does not need to know a lot of the details involved in 3D animation.

The system proposed in this project aims to make it easy and cheap to pose a static skeleton using a real-world counterpart such that anyone without knowledge of 3D skeleton posing can use it.

1.3. Existing technologies

The field of pose detection has been well explored as it is commonly used in motion capture technology. Motion capture can be described as the process of recording the movement of an object or person over time. In film making and games it is used to record the movement of body parts in order to animate characters in computer animation. Newer motion capture systems can even capture facial expression.

Traditionally motion capture systems[7] used markers in order to detect the 3D positions of the object. Multiple high-speed cameras would be strategically placed in the performance area and lighting conditions would be controlled. Every frame of data captured from the cameras would be synced and the 3D positions of parts of the body would be triangulated from these markers. These systems typically offer very accurate pose detection however the downside is the need for very precise calibration and set-up as well as the cost of multiple high resolution speed cameras (normally upwards of 8 cameras), the large space and the software needed to run such a system. It can typically cost hundreds of dollars in order to build a motion capture not to take into account the cost of specialists needed to clean-up and process the data from the output of the system.

A more recent approach comes in the form of the Microsoft Kinect[4, 10]. The Kinect is a low cost ($150) depth+RGB camera that is capable of detecting pose of a humanoid subject with less than a 4cm error for each body part. It works by generating a 3D point cloud using its IR depth sensor which it then feeds into proprietary software which is able to make sense of the point cloud and convert it into a skeletal pose in less than 10ms. When developing the Kinect, many terabytes of data of people in different poses were collected and body parts were labeled and fed into an 'expert' system running on a powerful cluster of computers. This resulted in a software package that was capable of utilising this past data to accurately estimate the pose of a human body from any frame of point cloud data from the sensor. The software was trained with other heuristics such as how the body is joined together, for instance that the hand is connected to the arm in order to make it faster and more accurate. The downsides of Kinect is its limitation to only work on humanoid subjects and that the subject must be positioned over 1m away from the sensor in order for it to work.

KINECT for XBOX 360.

The system we are proposing in this report would be both low cost due to the fact it would be able to work with any web-cam without the need for specialist hardware and it will be capable of modeling any rigid structure. Another difference from the above systems is that the camera itself can move around as well as the object we are capturing. Full motion capture is not in the scope of this project.

1.4. Issues

The problem is similar to a 3D reconstruction from motion problem, we will need to be able to determine the skeleton model pose from the information gathered from many views around the object. We must find a way to accurately do this using as much information as possible from the views that are collected.

Capture and mapping of object locations

Several computer vision techniques must be explored in order to determine the most suitable way to find and accurately place the locations of the markers which relate to specific parts of the object. The system will need to take into account partial occlusion of markers as well as adverse lighting conditions, both can have a significant effect on the accuracy of the pose estimations. Having a robust and accurate pose estimation of the markers will help reduce error further down in the system and will therefore result in a better end pose.

Noise and camera lens distortion can also affect the marker pose estimation. The system must be able to take these factors into account when detecting the markers and working out the pose of each marker.

Estimating the skeleton model pose

Since the marker detection step will most likely not generate a 100□% accurate poses of the markers and therefore of the parts of the model we must have a way to allow for these small discrepancies and hopefully use information from other views in order to make the final pose estimate as accurate as possible. The optimal solution to this problem should be able to work out the best pose estimate of the skeleton model based on both the marker locations and the information about the skeleton model provided by the user.

Missing markers during the marker capture stage must also be taken in account. Based on the structure of the model definition it could be possible to get an accurate estimation of the position of a part of the object even without having seen or only partially seen the marker relating to that part during capture.

1.5. Contributions

This report outlines the path it took to get to the final application which is a fully featured application that can estimate the pose of the real-world skeleton model in real-time using only a web-cam. We will explore the initial approaches we tried before getting to the final solution and evaluate what worked and what didn't work at each stage. We will then go into depth about the methods and techniques we used in the final implementation and also do an in-depth analysis about how accurate the system is. Finally we will do a guided walk-through of the system, explaining the different features available at each stage. The final application features a novel approach to the problem using bundle adjustment techniques to solve the problem of estimating the pose of the real-world object, the results will show the estimations are very accurate and can produce results which are only a few millimeters off.

Background 2

In this section we will take some time to explain some of the concepts and methods that will be later used in the implementation stage of the report.

2.1. Computer vision

Computer vision is the field that looks at extracting information from images and/or video, typically from a camera of some kind. This project specifically deals with using a web-cam to reconstruct the skeleton pose of a real-world model. In a typical computer vision flow we start from low level information such as detecting of edges and regions in an image and follow through to more high level reasoning about the scene such as how we can group together lines to form objects.

In this application we will be using a standard web-cam in order to capture images, these images will be processed to find markers and we will make sense of the object in the real-world through these markers. Since the application is designed to be able to work with any standard off the shelf web-cam the choice of web-cam for the project was not that important. Below we will go through some of the typical characteristics used to differentiate between cameras:

Focal length [15] in an optical system refers to the measure of how strongly the lens converges light. A short focal length bends light stronger and as a result allows the optical system to bring objects closer to it into focus. In photography focal length is measured in millimeters. The below image shows the effect of different focal lengths:

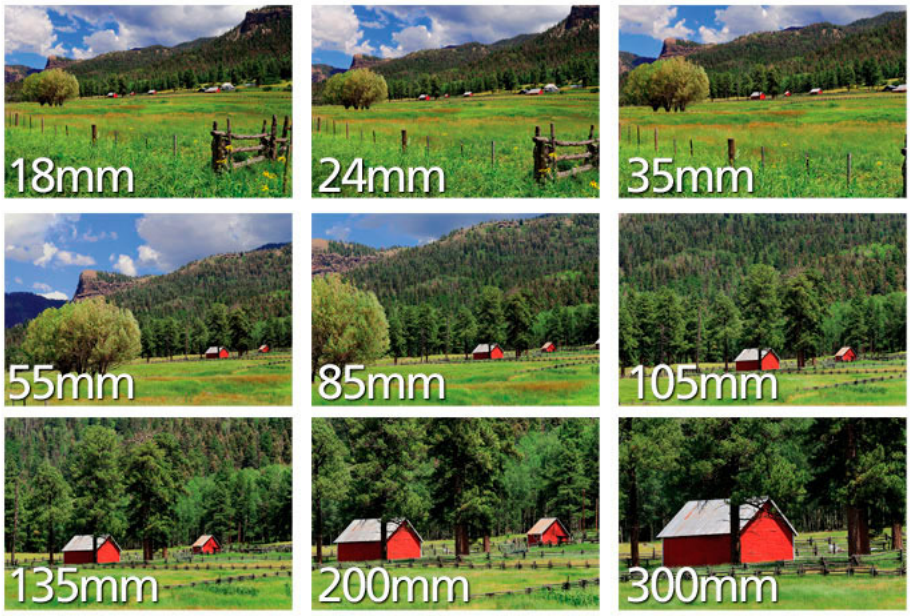


Fig. 1: Focus length differences

For our system a large focal length would be optimal however we don't expect the web-cams that will be typically used for the system will have a large focus length, most off the shelf web-cams have quite a small focal length.

The **frame-rate** of a video camera system refers to the rate of capture. This is measured in frames per second (FPS). A high FPS is preferred in computer vision applications as a high FPS implies a fast shutter speed which results in less motion blur. A common downside of a high FPS is that exposure is reduced. Exposure is the measure of how much light falls on the image sensor in one shutter cycle. Having a low exposure means the image is less bright and can result in the image being underexposed, the image will have loss of detail in darker areas.

Image noise can be a problem in computer vision applications as it can result in loss of detail and or incorrect labeling in certain computer vision techniques. In the context of digital video and photography, noise is generally accumulated in the image due to photo-diode leakage in the image sensor. Low-light conditions can cause greater noise as the camera system must compensate by increasing the sensitivity of the image sensor which can cause more noise due to leakage.

2.2. Camera model

It is essential in most computer vision systems to be able to model how the camera projects the real-world view onto its image plane[9]. The standard model that is used for this is the pinhole camera model. The pinhole model assumes the real-world view is passed through a single aperture point before hitting the image plane behind the aperture point. The relationship between the 3D point $[X, Y, Z]$ in the real-world and the image plane in (u, v) coordinates is shown in matrix form below:

$$\lambda \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

It can also be represented in the following two equations:

$$-u = f\frac{X}{Z}$$
$$-v = f\frac{Y}{Z}$$

The focal length of the camera is represented by f in the formulas above. We can also convert the u and v values to x,y coordinates on the image plane by taking the product of u or v with the width or height of the image plane respectively.

The pinhole model is useful for very simple systems however normally we wouldn't rely on the pinhole model by itself. Having a single point that the light passes through would not allow enough light for short exposures which is essential for video capture systems. Cameras use lenses to focus more light into the aperture point to get around this problem however this introduces another problem. In the manufacture of lenses, it is inevitable that some sort of imperfection will occur due to the near impossibility of creating a 'perfect' lens. We will need to take into account a way to correct these imperfections in the camera model. The first imperfection that may occur is that the optical axis or center may not match up with the center of the image plane, this can be caused by improper fitting of the lens body to the sensor and aperture hole. We can use the following revised formulas that take this into account:

$$x = f_x(\frac{X}{Z}) + c_x$$

$$y = f_y(\frac{Y}{Z}) + c_y$$

In this form f_x and f_Y are the product of the focal length and the new parameters $s_x$and s_y which represent the size of the sensor or image plane. c_x and c_y represents the point where the optical axis intersects the image plane also known as the principal point. Putting the (x, y) coordinates into homogenous form allows us to represent the above formulas in matrix form as follows:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$$

This is known as the intrinsic matrix as it encompasses the intrinsic parameters of the camera model. These parameters encompass focal length, image format, and principal point.

The next step is to look at correcting for the aberrations that may be present in the lens itself, we call these the extrinsic parameters [3]. The two most effecting imperfections is the radial distortion and the tangential distortion. The radial distortion is the effect where the further you move away from the image center, the more stretched the image appears, this is typically caused the fish eye effect which can be seen in Figure 2.

Radial distortion can be represented by an expanded Taylor series, typically we only consider the first few terms as the radial distortion tends to be dominated by the lower order terms in a typical camera system. In order to correct radial distortion we use the formula below:

$$x_u = x_d(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

$$y_u = y_d(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

Where x_u and y_u represent the corrected x, y coordinates, x_d and y_d are the distorted coordinates, k_i is the ith radial coefficient and $r = \sqrt{(x_d - x_c)^2 + (y_d - y_c)^2}$ where x_c, y_c is the principal point.

Fig. 2: **F**ish-eye

Tangential distortions occur when the lens is not fitted parallel to the image plane, it usually appears as a sort of shearing in the image. We can remove this using the following formula:

$$x_u = x_d + (2p_1 y + p_2(r^2 + 2x^2))$$

$$y_u = y_d + (p_1(r^2 + 2y^2) + 2p_2 x)$$

The next thing to do is to work out the all the unknown intrinsic and extrinsic parameters. A well known technique for doing this was invented by Zhang et al [18] . It involves taking many images of a multiple planar surfaces of known width and height and optimising the camera parameters until the model we estimate matches up with where the planar surface points are in the image. Many programs use his approach to work out the parameters, one of these programs is GML toolbox which we use for this application.

2.3. Bundle Adjustment

Bundle adjustment [17] is a term used to describe the problem of simultaneously refining the location of 3D coordinates seen in a number of viewpoints whilst also refining the parameters describing the camera and the 3D structure of the scene. Bundle adjustment is almost always the last step of any feature based 3D reconstruction algorithm. A number of factors need to be taken into account when deciding on the strategy to tackle this problem. For instance if we have zero-means Gaussian noise in each image, the problem becomes the maximum likelihood estimator. The problem essentially boils down to working out to minimising the re-projection error of points in each viewpoint. Non-linear least squares optimisation is typically the chosen method to tackle this problem.

2.3.1. Non-linear optimisation

Non-linear optimisation is the act of solving a set of equalities sharing unknown parameters. The aim is to try and find the parameters that best fit all these equalities, we use a objective function that tells us how well we are fitting the equalities. Generally we are looking to maximise or minimise the total cost of the objective function. The objective function is also non-linear in the case of non-linear optimisation. We can

visualise this as a n-dimension graph where n is the number of unknown parameters, each data value represents the cost of the set of n parameters. The lowest point in the graph is the solution as this is the set of parameters which corresponds to the lowest cost when put into the objective function.

A common approach to this problem is to use Levenberg-Marquardt [14], it is very common in computer vision applications that need to do non-linear optimisation. Levenberg-Marquardt aims to find a numerical solution to the problem of minimising an objective function over the space of a set of unknown parameters. The problem it aims to solve can be described in the below formula:

$$S(\beta) = \sum_{i=1}^{m} (y_i - f(x_i, \beta))^2$$

In the above formula β represents the set of unknown parameters we are trying to solve, (y_i, x_i) is the pair of independent and dependent variables and f represents the objective function.

LMA is a iterative algorithm, to start it the user must provide a set of guess parameters. One problem with LMA is that its will terminate when it reaches a local minimum due to the fact that it can't differentiate between the global minimum that we want to find and a local minimum. Therefore the choice of starting guess parameters can be very important depending on whether you will be expecting any local minimums or not. If there are no local minimums the choice of starting guess parameters is not important as the optimisation will eventually converge to the correct solution.

The algorithm starts by applying a new estimated set of guess parameters and working out the new objective cost:

$$f(x_i, \beta + \delta) \approx f(x_i, \beta) + J_i \delta$$

$$J_i = \frac{\delta f(x_i, \beta)}{\delta \beta}$$

If we reach the case where the cost function is minimised to zero we can see that the gradient of S with respect to δ will be zero as well.

$$S(\beta + \delta) \approx \sum_{i=1}^{m} (y_i - f(x_i, \beta) - J_i\delta)^2$$

If we now take the derivative of the above and set it equal to zero we arrive at the following formula in vector form:

$$(\mathrm{J}^{\mathrm{T}}\mathrm{J})\delta = \mathrm{J}^T[y - f(\beta)]$$

J is the Jacobian matrix of j where the ith row of J equals J_i. We can solve the equation for δ. Levenberg also added another contribution that helps speed up the algorithm. We can use a damping function that if the reduction of S is slow we can tend towards using the Identity matrix as a product of δ instead of the Jacobian. This brings in more in line with the gradient descent method. If the reduction of S is rapid we lower the damping function which allows the algorithm to perform more like the Gauss-newton method. The damped version is shown below:

$$(\mathrm{J}^{\mathrm{T}}\mathrm{J} + \lambda\mathrm{I})\delta = \mathrm{J}^T[y - f(\beta)]$$

The problem with this is that if the damping function is very large, the inverting of the $\mathrm{J}^{\mathrm{T}}\mathrm{J} + \lambda\mathrm{I}$ becomes insignificant. The final improvement therefore was to use a diagonal matrix consisting of the diagonal elements of $\mathrm{J}^{\mathrm{T}}\mathrm{J}$ instead of I. This gives us the final Levenberg-Marquardt formula:

$$(\mathrm{J}^{\mathrm{T}}\mathrm{J} + \lambda diag(\mathrm{J}^T\mathrm{J}))\delta = \mathrm{J}^T[y - f(\beta)]$$

2.4. PTAM

PTAM [12] is a method presented by George Klein and David Murray for estimating camera pose in an unknown scene using only a calibrated mono camera system. It stands for parallel tracking and mapping and is called as such because it processing on two parallel threads, one of which is performing real-time tracking of the scene whilst the other is building a map from the information provided from the tracking.

The tracking part of the method assumes that a 3D point cloud of feature points has already been created. It processing a real-time feed from a camera in order to maintain an estimate of the camera pose in the map. A basic overview of this is below:

1. New frame from camera - a prior pose is estimated from the motion model
2. Map points are projected onto the image based on this estimated pose.
3. 50 of the coarsest scale features are matched in the image
4. The camera pose is updated from the matches
5. 1000 feature points are now projected onto the image and searched
6. A final pose is obtained from these matches.

The tracking stage also detects FAST feature points. If the camera has moved a sufficient distance and the tracking quality is deemed to be good, a key-frame can be stored and passed to the mapping thread in order to improve the map.

The mapping thread is concerned with creating a 3D point cloud of feature points so that the camera thread can re-project these points out and calculate the estimated camera pose. It does this using bundle adjustment techniques to determine the camera pose of all the key-frames passed from the tracking thread and poses of all the map points seen in those key-frames. Levenberg-Marquardt bundle adjustment is used to do this. A problem occurs when we are rapidly exploring the scene and many new key-frames are being added. LMA is a $O(n^3)$ problem which means that any increase in key-frames would cause the time perform optimisation to increase rapidly. The method doesn't expect all key-frames to be sharing all feature points so it can use this heuristic in order to reduce the number of key-frames to perform optimisation on per run of the optimisation. This allows the method to concentrate on optimising newer key-frames as they arrive. Any time the camera is not exploring the mapping thread can use to refine older key frames. PTAM limits the local bundle adjustment to 5 key-frames which consists of the newest key-frame and the 4 key-frames nearest to it.

The final effect is an accurate camera pose tracking system for small AR workspaces. It works very well if it can find a lot of feature points to track in the scene as it has a lot of information to build an accurate map. One downside of PTAM is that moving objects in the scene can cause problems with the map as feature points may be lost or incorrectly tracked.

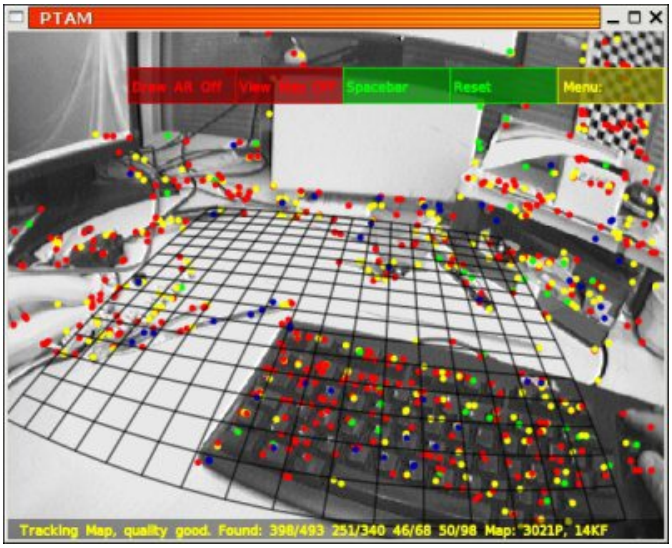


Fig. 3: PTAM in action

2.5. Marker detection

In computer vision it is common to look for certain elements in the scene in order to determine information about the total scene. One way of making a computer vision application more robust is to design the elements the application is supposed to look for to make them as easy to track as possible and/or to encode information about the scene. These are called fiduciary markers and are used as a point of reference or measure. In this application we will be using fiduciary markers in order to tell us about the pose of the skeleton model. We therefore need to use a marker design that will allow us to determine the pose as well as be able to distinguish between different markers. We also need to have a system that will be able to detect these markers from the web-cam feed. There have been many approaches to this problem, mainly in order to perform augmented reality. One of the first frameworks which solves the problem of marker detection and pose estimation is ARToolkit.

2.5.1. ARToolkit

ARToolkit [8] is a augmented reality framework that contains a fully featured marker detection process. It was first released in 1999 and since then there has been many enhancements and other AR frameworks based on the principles of ARToolKit. The markers that ARToolkit uses are black and white symbols surrounded by a black square

with a thick white border. An example of an ARToolkit marker can be seen below. The symbol inside is used to uniquely identify the marker. The use of the black square surrounded by the white border makes it easy for the detection process to find the marker in the scene.

The marker detection process follows a number of steps. Thresholding is first performed on the image to extract out the black square of the marker. This is a simple lighting binarisation, images with an averaged RGB value of less than the threshold become 0 (black) whilst those above the threshold become 1 (white).

Contouring can then be done to extract out the shape of the black square in the image. Line fitting is then performed on the contour to fit 4 straight lines onto it. These four lines make up a quadrilateral from which 4 corners can be extracted.

Given the four corners we can estimate the pose of the square using the co-planar posit algorithm. Furthermore we can recognise which marker we have detected by performing pattern recognition of the image inside the marker with those stored in a known dictionary.

ARToolkit has the benefits of being relatively fast as its computer vision step is not computationally expensive and it has been ported to many different platforms. However it lacks in some key areas, occlusion of the marker is a problem as the detection fails if even some of the marker is occluded due to the fact that occlusion will break the contour fitting of the black square. Adverse lighting conditions are also a problem due to the use of thresholding to extract out the features. Reflective materials used to make the markers as well as low-light conditions can cause the thresholding step to incorrectly label certain areas of the image.

2.5.2. AVLAR

We have chosen to use AVLAR marker detection [6] which is a framework based on the ideas of ARToolkit but with some improvements. The key improvement is the inclusion of the edge-detection method of detecting the outline of the markers as outlined by Martin Hirzer [5].

The process begins by dividing the image into regions of 40x40 pixels. Each region is divided further by scan-lines 5 pixels across and down. Edge detection is performed across these scan-lines to determine a potential edge of a marker. Edges that have are not black-white are rejected, we can check this by looking at the RGB colour values of the potential edge. A RANSAC approach is now used to group edges in a region

to form line segments [11]. Two random edges are chosen in the region. If the two edges match in orientation and this orientation matches the orientation of a line joining the two edges the two edges form a hypothetical line. The other edges in the region are then sampled to see whether they support the line. An edge supports the line if it matches the orientation of the line and is close to the line. This is repeated many times in the region and the resulting lines are compared to see which ones have the most support. The lines with the most support are chosen whilst the rest of the lines are discarded. The two furthest away edges supporting the line are chosen as the end points thus forming a line segment. The segments are then merged if they follow the same orientation and the pixels between them are considered edge pixels matching the orientation of the segment using the Sobel operator. Finally quadrilaterals are formed from joining up line segments that have end points close to each other and have a black inside the area formed by joining up the line segments. The corners of the quadrilateral are represented by the end points of the line segments.

Overall this approach is more robust than the threshold and contouring approach used in such frameworks are ARToolkit. It can handle adverse light conditions better as it is looking for an edge instead of simply relying on a single pixel value for thresholding. Furthermore it can handle occlusion to some degree as only 3 corners need to be detected to form a quadrilateral, the fourth corner can be estimated using the incomplete line segments forming off the existing corners.

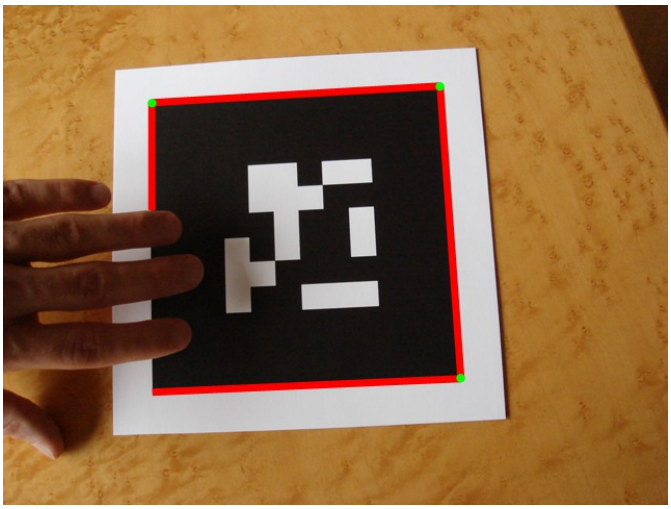


Fig. 4: Edge marker detection

Another improvement made was to use markers that encoded their own ID in the pattern. This was done using markers which have a binary grid as the pattern which encodes to a number ID. Instead of doing pattern recognition to find the ID, we can simply read the ID off the grid directly. Figure 5 shows an example of how the IDs are encoded.

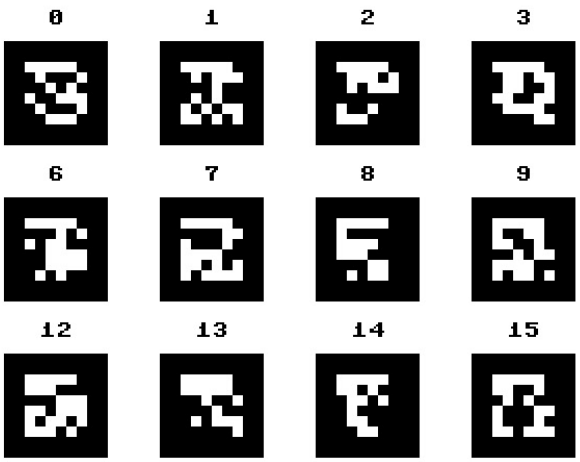


Fig. 5: **M**arker IDs

2.6. Motion capture file formats

Full motion capture over many frames is not within the scope for this project, however we plan to utilise the BVH file format [13] that motion capture systems use to store skeleton pose information as it allows the output of our system to be easily portable between a lot of 3D modeling tools. We begin by explaining some key terminology that will be used throughout this report in the context of character pose/animation:

Skeleton

This refers to the collection of all elements that make up our character.

Bones

A bone represents a basic entity in the skeleton. It is the smallest element that is subject to individual rotation and transformation. The bones can be labeled, for instance a bone in a skeleton can be called the femur.

Degrees of freedom

Each joint in the skeleton has a set range of movement that it can move in. For instance the joint between the upper and lower arm is limited to only being able to move in two directions.

BVH (BioVision Hierarchical data)

BVH consists of two parts. The first describes the hierarchy of the skeleton and the initial pose. The second describes the movement of each bone for every frame of animation. An example of the first section of a BVH file is below:

```
1  HIERARCHY
2  ROOT Hips
3  {
4    OFFSET 0.00 0.00 0.00
5    CHANNELS 6 Xposition Yposition Zposition
6                              Zrotation Xrotation Yrotation
7    JOINT RightUpLeg
8    {
9            OFFSET -3.910000 0.000000 0.000000
10           CHANNELS 3 Zrotation Xrotation Yrotation
11           JOINT RightLowLeg
12           {
13             OFFSET 0.437741 -17.622387 1.695613
14             CHANNELS 3 Zrotation Xrotation Yrotation
15             JOINT RightFoot
16             {
17                     OFFSET 0.000000 -17.140001 -1.478076
18                     CHANNELS 3 Zrotation Xrotation Yrotation
19                     End Site
20                     {
21                     OFFSET 0.000000 -4.038528 5.233925
22                     }
23             }
24       }
25   }
26 }
```

The above describes a skeleton with the root as the hips. The rest of the bones are described in a recursive fashion with each bones information and children bones are

encapsulated in curly brackets. In the example above we have a single upper leg bone appended onto the hips whose single child is the lower leg who has a foot as a child bone. End site refers to the ending point of the hierarchy. Note that the offsets are representing the joint location joining the parent to the child, this is shown in the picture below.

OFFSET refers to the relative translation in x,y and z that the child bone has to its parent. In the case of the hips it refers to its translation globally. Translation refers to the range of movement that the bone can have relative to its parent bone. Bone length can be inferred from the offset information.

The second section is shown below:

```
1 MOTION Frames: 1
2 Frame Time:
3 0.04166667 -9.533684 4.447926 -0.566564 -7.757381
4 -1.735414 89.207932 7.892136 12.803010 -28.692566
5 2.151862 -9.164188 8.006427 -5.641034 -12.596124
6 4.366460
```

The above example shows a single frame of motion. The numbers on the line below frame time are the rotations for the bones. They relate to the channel data provided in the first section. For instance the first 6 numbers are for the hips as it has 6 degrees of freedom, the next three numbers are for the upper leg and so on.

BVH is the most widely used format for motion capture and is supported by most major 3D modeling and animation tools. It has some downfalls however. Since bone length is not explicitly defined, there can be conflict if a bone has multiple children. Also the file does not give details of the environment such as which direction points upwards.

3 Implementation

This section deals with how the system was implemented to achieve the goals set out. We will go over the initial approaches to the problem, how the solution evolved over time and finally describe in detail how the final solution was implemented and the reasoning about the choices made over previous approaches.

3.1. User Defined Model

The first task was to define the model that the user would be inputting into the system. A decision was made to base this model on the standard BVH model explained in the background section. This model was chosen as it was simple to understand and was flexible enough to express almost any real-world rigid skeleton model. The model fully complies with the BVH standard and can be exported to a BVH file after obtaining a final estimated pose.

3.1.1. Bone model

A model will be defined from its 'root' bone, each bone can have any number of children bones. Each bone will have an associated x,y,z Euler rotation, an offset from its parent bone and a set of markers attached to the bone. It is easier to understand if you think of a bone as the bone joint positioned at the offset from the parent bone joint. The root bone will always be at position (0,0,0) and have 0 rotation in the x,y and z angles. The transform of a bone joint in the coordinate space with the center at the root (0,0,0) and with the Up vector represented as (0,1,0) can be represented with the following formula:

$$\prod_{i=0}^{n} R_i T_i$$

Where n refers to the current bone, i is referring from the root bone up to the current bone, R is the rotation matrix and T is the translation (offset) matrix. As such the transform is a affine transformation that preserves the lengths (offsets) of all the bones.

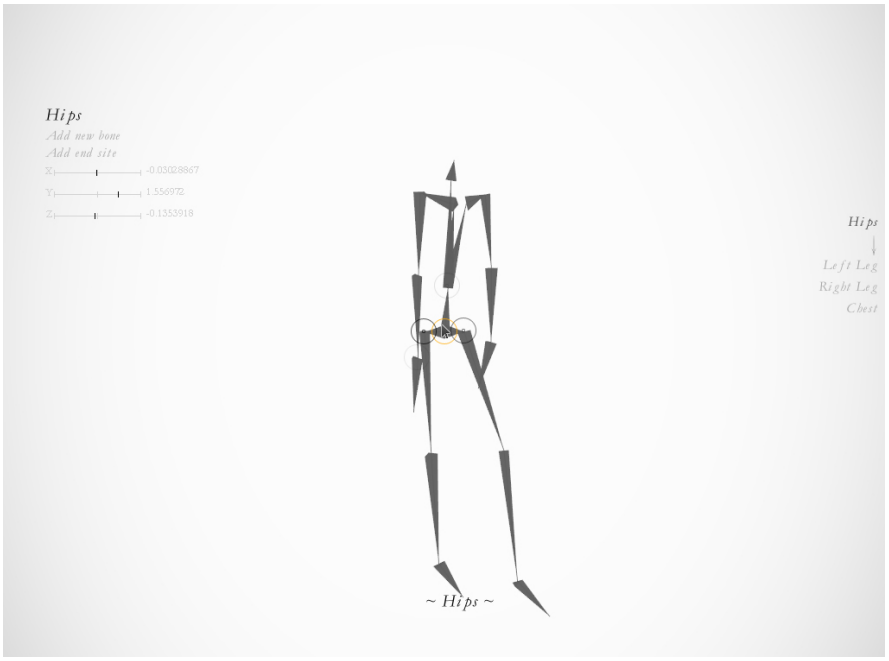


Fig. 6: **H**umanoid bone model

The **global coordinate frame** refers to the coordinate system that is centered at [0,0,0] and is orientated with the Up vector as [0,1,0], the Foward vector as [0,0,1] and the Right vector as [1,0,0]. The offsets of each bone refer to the [X,Y,Z] offset from the parent bone position in the root bone coordinate frame.

The **skeleton pose** refers to the complete set of rotations on the skeleton bone model. The aim of this application is to be able to detect and make an accurate estimation of the skeleton pose using the real-world model and a web-cam.

The entire bone structure is stored in both a list structure as well as a tree structure where each parent has a bi-directional link to its children. It is also worth noting that since we need calculate bone positions and marker positions quite regularly, we avoid computing the entire chain of transforms every time we want a specific bone by precomputing and storing the global transforms of parents before computing the global transforms for their children. The list structure helps us easily achieve this by always storing parents before their children in the list, we can simply iterate through the list to compute global transforms.

3.1.2. Markers

A total of 1024 markers with unique IDs are available to use, this is due to the number of unique rotational variant patterns possible on a 5x5 grid. These markers can be associated with a bone by marker ID, offset from the bone and the x,y,z Euler rotation of the marker in the bones local coordinate system. To find the position of a marker in the coordinate system of the root bone is as follows:

$$(\prod_{i=0}^{n} R_i T_i) M R_n M T_n$$

Where MR is the marker rotation and MT is the marker translation.

A marker also has 4 ordered corners, To get the 3D Vector location of a specific corner, the final marker transform needs to be multiplied by the offset of the marker corner from the center of the marker. The model defines the edge length in mm of the markers on the model.

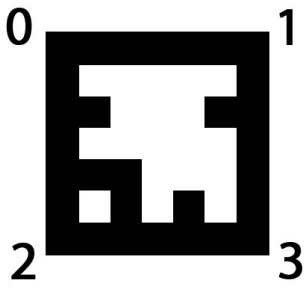


Fig. 7: An example marker with corners highlighted

Throughout this report we talk about the **marker coordinate frame**, this refers to the coordinate system with the marker center as [0,0,0], the Up and Right vector being planar with the surface of the marker the and the Foward vector coming off the top of the marker.

3.1.3. Constraints

The user can also define constraints on the rotation. For instance they can limit the movement of a bone to only rotate on the relative Z axis. This is useful in situations

where the real world model is actually constrained to less than 6 degrees of freedom in terms of rotation. The rotation of the knee joint is a good example, it is constrained to movement in only 1 axis as shown in Figure 8.

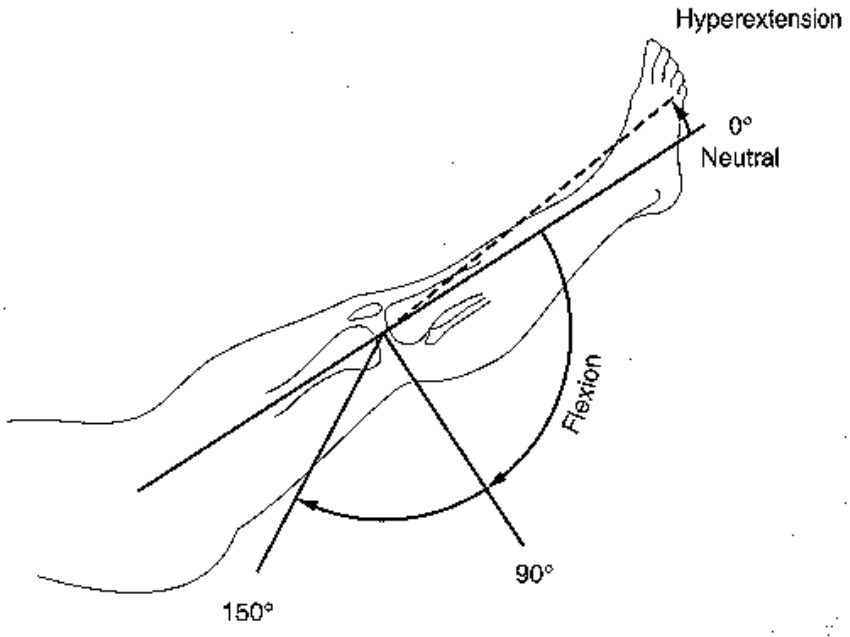


Fig. 8: Knee joint

The default constraint is +/- 180 degrees on X,Y and Z. The user can specify anywhere between these values for the constraint. The root bone is always considered to be constrained to all 0s in X,Y and Z for optimisation purposes, the rotation of the root bone does not particularly matter as we are only concerned with the rotations relative to the root bone. The benefit of constraining the angles in the model is that the estimation at the end is likely to be faster and more accurate as it will not search in regions with angles that are outside of the constraining range.

3.1.4. File format

The application will output models into text format when the user wants to save and load the models up for future use.The following Figure shows the specification for the bone model file:

```
[Name]&[ParentBoneName]*
[xLow][xHigh][yLow][yHigh][zLow][zHigh]
[xrot] [yrot] [zrot]
[offsetx] [offsety] [offsetz]
[endsitex]* [endsitey]* [endsitez]*
[no of markers]
[markerID] [xoff] [yoff] [zoff] [xrot] [yrot] [zrot]
```

Fig. 9: Specification for bone model file
The asterisks indicate an optional value in the specification

The file is exported with the .figure extension. A simple example with 3 bones is shown below:

```
-- Model definition for: testmodel --
Hips
-180 180 -180 180 -180 180
0 0 0
0 0 0
1
0 0 0 0 0 0 0

2&Hips
-180 180 -180 180 -180 180
0 0 0
-40 0 0
1
1 0 0 0 0 0 0

5&2
-180 180 -180 180 -180 180
0 0 0
0 40 0
1
2 0 0 0 0 0 0
```

Fig. 10: Model definition for testmodel

3.2. Video input

Directshow is used for the video input, the application accepts any standard web-cam that is capable at running at 30 frames per second and outputting at 640x480 resolution.

Each incoming frame is preprocessed to extract out information such as which markers are viewable and the relevant marker transforms. A single 640x480 frame and associated information will be referred to as a **view** throughout this report.

3.3. User Interface

A fully featured GUI was created in order for users to easily create, load and save their own models. Upon starting the application the user can choose to edit an existing model or create a new model. The GUI allows the user to add, edit or delete existing bones, associate, dissociate or edit markers on a bone and see their changes reflected in real-time with the interactive 3D model viewer. Upon finishing editing or creating the model, the user can save the model with a chosen file name. Interaction of the 3D model viewer can be controlled solely with the mouse or with the mouse and keyboard. The user can pan, tilt, rotate and zoom in/out using the mouse. Furthermore the UI provides an intuitive way to select/focus on a particular bone using the mouse.

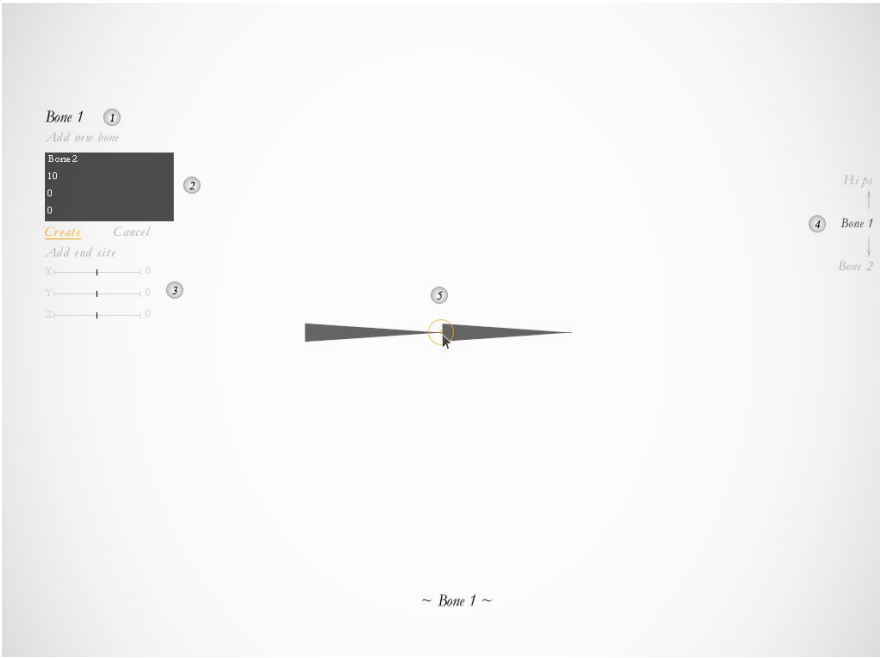


Fig. 11: **E**xample of adding a new bone

1. Bone name
2. Input for new bone offset and name
3. Rotation of selected bone

4. Connected bones of selected bone
5. 3D model viewer

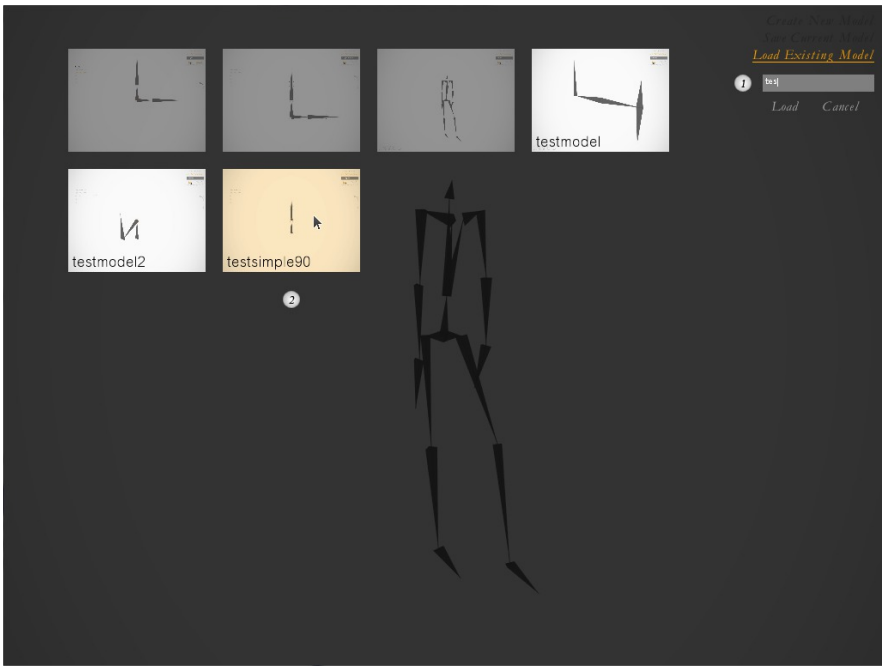


Fig. 12: **L**oading saved models

1. Text input box - Models sharing the prefix with the name in this box are highlighted
2. Model viewer

The graphics were implemented using XNA. Model files have the extension .figure and are saved into separate folders with the name of the model within the directory: <Documents>/figurate/models/<ModelName>/<ModelName>.figure. Users can overwrite existing models by specifying an already existing model name when saving, they will be prompted that they are overwriting before they confirm to save. The UI also provides a way to see the ordering of corners on a marker. Simply select marker mode on the top left corner and all the markers on the model will display the corner indexes (0-3).

After model creation is done or an existing model is loaded, the user can choose to run one of two options. The first is a test that simulates running one of the implemented rotation optimisation techniques explained in the next section. This can be run by

pressing the F1 button. The second option is to perform the optimisation using the actual real-world model and the web-cam. The user will be able to see the web-cam view on the right of the screen and the watch the 3D model being fitted to the real-world model in real-time on the left of the screen. This can be started and stopped by pressing the F2 button. On-screen guides to help the user obtain the best optimisation will be displayed depending on the optimisation technique chosen.

3.4. Marker pose detection

All the approaches required the system to know where the markers in the video frame were relative to camera. The AVLAR library by VTT was used as it provided methods to detect markers in the scene. AVLAR uses an edge detection approach in order to detect markers in the scene. A basic overview of the process it uses to detect markers is as follows:

1. Grab image from web-cam
2. Convert to Gray-scale
3. Use adaptive threshold to binarise image
4. Search for edges in image
5. Group edges into lines
6. Find sets of four joined lines to get potential marker
7. Verify marker by checking if outside is white and inside is black
8. Transform 2D quadrilateral into square
9. Determine ID from pattern in square.

A more detailed explanation is included in the background section. The four corners of a marker can be found in the scene using this routine, these corners can be ordered in a systematic way due to the fact that the pattern inside the marker is rotation variant and thus can tell us the rotational orientation of the marker in the scene. From this we can estimate the camera transform relative to the marker position using the coplanar POSIT algorithm [16], POSIT generates two approximate poses in the degenerate case where the four input points are coplanar. This routine is outlined below:

Algorithm 1 POSIT

1. $\varepsilon_{i(0)} = 0, n = 1$
2. Beginning of loop
   Solve for i, j and Z_0 using the below 2 equations. When the points are coplanar, the additional equality $i.j = 0$ must be used. Two poses are found
3. Compute $\varepsilon_{i(n)} = \frac{1}{Z_0} M_0 M_i \cdot k$, with $k = i \times j$. In the coplanar case, two sets of ε_i with opposite signs are found.
4. If $\mid \varepsilon_{i(n)} - \varepsilon_{i(n-1)} \mid <$ Threshold, Exit
   Else $n = n + 1$. Go to step 2.

M refers to the model points where M_0 is the reference point of the object and M_i is the position of the ith model point relative to M_0.

$$M_0 M_i \cdot I = x_i(1 + \varepsilon_i) - x_0,$$

$$M_0 M_i \cdot I = y_i(1 + \varepsilon_i) - y_0,$$

with

$$I = \frac{f}{Z_0} i, J = \frac{f}{Z_0} j,$$

Fig. 13: Solving for i and j

The second step is to discard one of the approximate poses. This is achieved by using the heuristic that all 4 corner points must have been in front in the camera ($Z_i > 0$). It can be seen that at each iteration, two poses are obtained, this means we have 2^n poses after n iterations. In practice we can generally discard one of the poses per iteration and therefore only have to travel done one branch of the tree of possible poses. After achieving the threshold, we obtain our final estimated pose.

3.5. Camera calibration

In order to get any sort of accuracy from the pose estimation of the markers, we must take into account the distortion of the camera lens. The model for camera distortion is explained in more detail in the background section. ALVAR can take into account camera distortion if the user inputs an XML document that tells it the camera's intrinsic and extrinsic parameters. GML toolbox was run with two marker boards of size 9x6 (box size 23.6mm) and 60 images in order to calibrate the camera. The results from GML toolbox were then inputted into the XML file for use with ALVAR.

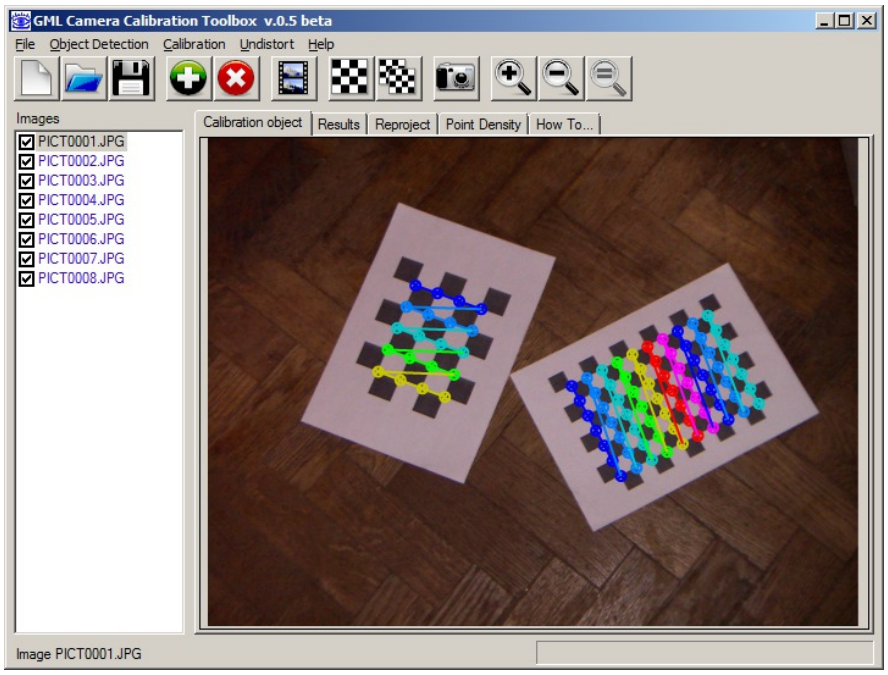


Fig. 14: GML camera calibration

3.6. Initial approaches

In this section we will explore some of the initial approaches tried before coming up with the final solution. The advantages and disadvantages of each approach will be listed and the features carried forward into the final solution will also be highlighted. Each of the approaches was based on the idea of solving bundle adjustment using non-linear programming, that is to solve a set of equations collectively over a set of unknown variables with some sort of objective function that must be minimised. This objective function is non-linear. The technique used to solve these equations and minimise the objective function was chosen to be the Levenberg-Marquardt technique which is explained in depth in the background section of this report. The reason why we chose to use non-linear programming was because we want to take as much information from the scene into account. Each view from the web-cam can only see certain parts of the model and we wanted a way to collectively use different views in order to estimate the rotations of each bone. Furthermore, the final transform of each bone is dependent on the transforms of its chain of parent bones all the way to the root. As such we know that each equation will take into account a set of bone rotations. We can therefore know that the unknown parameters we will be optimising over will be the x,y,z rotations for each bone over the range of +/- 180 degrees.

3.6.1. PTAM approach

The first approach involved using PTAM in conjunction with marker detection in order to generate a map of where the markers were in the unknown scene. PTAM would be used in order to estimate the global camera pose whilst exploring the environment with the web-cam. The camera model used is the modified pinhole camera model described in the background section. This approach is outlined below:

1. For every view obtained from the web-cam, markers are detected and each corner is stored as a 2D pixel coordinate as to where it appeared on the view. This is stored alongside the 6 camera parameters that describe the camera pose in the global coordinate frame. The $[X, Y, Z]$ look-at vector and the $[X, Y, Z]$ camera position.
2. If we see the root bone (we know we are seeing this if we see the marker associated with it), we can then work out the transform to get from the global camera coordinate frame to the root bone coordinate frame. We then know for every frame where the camera is in relation to the root bone position. We can determine from this where all the other marker corners should be on the screen based on the estimated rotation values.

3. The objective function aims to minimise the sum of the euclidean distance between the projected 2D pixel coordinate of a specified corner from the estimated model and the same corner as seen in the view of the web-cam across all selected views. Views are randomly selected from the set of all views for each run of the optimisation. The objective function is summarised in Figure 15.

$$\sum_{j=0}^{m-1}(\sum_{i=0}^{3}(ME_{ij} - MV_{ij})^2)$$

Fig. 15: Objective function to minimise

Where m is the view number, i is the marker corner index, ME_{ij} is the projection of the ith corner in the estimated skeleton model onto the screen using the camera pose in the jth view. MV_{ij} is pixel coordinate on the screen of the ith corner in the jth view.

We can find the projection of a specified corner by first finding the translation vector in the root bone coordinate frame from the estimated model (refer to section 3.1) and using the equation in Figure 16.

$$u = \frac{Xf}{Z}\ ,\ v = \frac{Yf}{Z}$$

$$x = \frac{1}{2}Width + u\frac{1}{2}Width\ ,\ y = \frac{1}{2}Height + v\frac{1}{2}Height$$

Fig. 16: Projecting corner onto viewing plane

Where (u, v) are coordinates on the viewing plane, (x, y) is the pixel coordinates on the screen, f is the focal length (in mm) and (X, Y, Z) is the 3D coordinate of the corner in the frame of the camera space. $Width$ and $Height$ refer to the width and height of the screen in pixels.

Each run of the Levenberg-Marquardt optimisation takes 30 views, therefore the total number of equations is equal to $30\sum_{i=0}^{b} 8M_b$, where b is the bone count and M_b is the number of markers on bone b. There are 8 equations per marker as we take the x and y coordinates for each corner as separate equations to optimise over. 30 was chosen as the view count as the Levenberg-Marquardt algorithm is of the order $O(n^3)$, 30 was a good compromise between number of views and computation time for the average model. There were also problems related to local minimums if we selected too

many views. The number of views taken can be tweaked in the **TestLevenbergPTAM.cs** class file if the model is very large or very small.

Implementation The first step taken was to compile and run PTAM on Windows, this was done in Visual Studio 2010. After a build of PTAM was achieved, work was done to make PTAM as a separate process be able to communicate with the main application running in C#. Named pipes was chosen as the method of communication as it had a low overhead and was ideal because of the fact that both processes would be running on the same machine. PTAM was modified with an extra thread that acts as a server for the named pipe communication, additionally an extra procedure was added to the main tracking loop which detects markers in the scene and estimates the pose of the markers. Since PTAM already grabs frames from the web-cam it was not needed for the C# application to grab camera frames of its own.

Upon connection to the server as a client, the main application would start requesting views from PTAM. PTAM would be passing to the client a 640x480 RGB24 image representing the current frame as well as the associated camera pose and also the marker transforms and IDs in the current frame. This would then be stored in the main application as a new view for possible use in a future optimisation. The camera view from PTAM can be activated in the main application by clicking on the ``Activate PTAM'' button in the bottom left of the UI. When the user specifies to start the optimisation, any incoming frames are saved as views if the camera pose stability is deemed ``good'' by PTAM. When the user presses F2 again, Levenberg-Marquardt is used to estimate the rotations of the bones using 30 of the views selected at random from all views. The optimisation can be repeated with another set of 30 views by pressing F2 again. The old estimated rotations are used as the starting guesses for the new run of Levenberg-Marquardt. The camera images are displayed to the user in real-time.

A simulated test was also created for this approach. It started by rotating the bones at random, this would be the result we are wanting to get to at the end of optimisation. Next it generated 30 random camera poses around the skeleton model. For each view, the skeleton model with the rotations applied was projected onto the screen using the camera pose associated with that view. Gaussian noise was then applied to the 2D positions of the corners in each generated view. These generated views were used as input to the Levenberg-Marquardt simulated run. The reason for creating this simulated test was to determine whether this approach would be viable from an early stage, and to easily determine how accurate we could get with this approach. We could easily determine the rotational error for each bone and the 3D point difference for each marker corner between the estimated and actual model.

Evaluation From the simulated runs, we quickly determined a problem area for this approach. From running the simulation 50 times, about 5 of these runs resulted in very large rotational errors whilst the other 25 resulted in acceptably close (+/- 0.1 radians) rotations per bone. The cause of the bad runs was due to the fact that Levenberg-Marquardt was stuck in a local minimum when optimising over the set of all equations causing the algorithm to terminate with a bad estimation. A number of possible fixes were outlined to solve this issue:

- **Random starting parameters** - We would be running simultaneous instances of Levenberg-Marquardt with randomly chosen starting parameters and selecting the instance with the minimum objective function after all the instances return. The downside to such an approach would be the computational overhead of running multiple optimisations at once.
- **Use marker rotations to determine starting parameters** - The orientation of one marker to another marker on the parent bone allows us to roughly determine the rotation of the parent bone. We can use this as input into the LM optimisation.
- **Choose views that will produce better results** - Local minimums can be reached due to poor selection of views. If we never see any of the markers associated with a bone and the bone has both a child and parent, there can be multiple possible rotations possible resulting in many local minimums. We can use some heuristic to choose 'good' views to use for a run of the optimisation.

Some of these improvements would be implemented in future approaches.

The major problem associated with this approach was discovered when we attempted to gather the views from PTAM. The tracking information obtained from PTAM was not very stable resulting in bad estimation of the global camera pose and as a result a bad projection of the 2D marker corners. Another problem was that PTAM would lose the global map quite often, every time this happened, the root marker would have needed to be seen in the view of the camera in order to use the views captured with the new map. There were a number of reasons why this was happening:

- **Not enough feature points in scene** - PTAM works best when there are a lot of feature points to track across frames, the best feature points come from textured and cluttered environments. Since the application would be typically used when the web-cam is looking at a mostly white object on an empty desk, it was likely having difficulties getting a large number of good feature points to track. Additionally the camera would typically be closeup to the object and rotating around it. The feature points cannot be tracked when the object is occluding them which would happen a lot as the camera moves in a path around the object.

As a result of having only a small set of valid feature points, tracking quality was degraded.

- **Wide angle lens** - PTAM works best with a wide angle lens as it can see more of the environment in a single frame and therefore find a larger number of valid feature points to track. The application is made to work with only a standard web-cam with a normal FOV, this issue compounded with the above issue further degraded the tracking quality.
- **Relying on the root bone marker** - Views in a map that hasn't seen any of the markers attached to the root could not be used in the optimisation step. The camera pose used in the optimisation was the global camera pose transformed into the root bone coordinate frame meaning that the quality of a view relied heavily on the map tracking the pose of the root bone accurately. It would be possible to make the optimisation root bone pose invariant by not specifying that the root bone was at position [0,0,0] with no rotations and allowing optimisation to run with views using the global coordinate frame. However this introduced more problems as it caused more local minimums due to the fact that the skeleton root bone was not fixed in position.

Some of these problems could have been remedied by having the user trained in how to move the camera as well as modifying the environment to make it easier to get good feature points (such as placing a checkered pattern underneath the model) but it was decided that this would be against the principle of the application being able to work (almost) anywhere and by anyone without knowing any technical knowledge about it. This approach was ultimately abandoned due to the after-mentioned problems, although some of the ideas discussed above would appear in later approaches.

3.6.2. Marker distance approach

The next step was to look for a objective function that did not rely on global camera position. The upside of this would be that we would not need to reply on any sort of accurate tracking of the global camera pose, the downside was that we would need to rely on having 2 or more markers in the frame at a time. There would be no way of conferring information about a view with only a single marker due to the fact we don't know how the camera is orientated in relation to the other markers on the model.

The intuition behind this approach was that the distances in 3D between the markers would give us some idea about the rotation of the bone in the model. Based on this an initial objective function was devised:

$$\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(\mid T_i - T_j \mid - \mid D_i - D_j \mid)^2 \quad i \neq j$$

Fig. 17: Distance cost function

Where T_i refers to the 3D vector location of the center of marker i detected in the real-world view in the camera coordinate frame, D_i refers to the 3D vector location of the center of the marker i from the estimated model in the root bone coordinate frame and m, n represent the marker count

This new cost function was inputted into the simulation and test data was generated for a simple 6 bone model in order to test the viability of the approach. Upon running the tests it became clear that the cost function had no way of determining the local rotation of a marker since it was only relying on the distances between the marker centers. This problem was most obvious on the end sites of the skeleton model where if the marker was positioned at the bone joint, the bone could an infinite number of possible rotations that would lead to the same cost.

The next improvement was therefore to not rely on the distances between the center of the markers but to look at the distances between associated corners on markers. Associated corners are corners that have the same corner index on different markers. This allows the optimisation to take into account local rotation information that may have been lost by only looking at the distances between the centers of the markers. The objective function is shown below:

$$\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\sum_{c=0}^{3}(\mid T_{ic} - T_{jc} \mid - \mid D_{ic} - D_{jc} \mid)^2 \quad i \neq j$$

Fig. 18: Corner distance cost function

The variable c refers to the corner index (0-3)

Implementation

Instead of looking at individual views and minimising the distance cost for each view, it was decided to minimise the average distance of all views for a pair of marker corners. This was done to reduce computational cost, instead of optimising $Viewcount \cdot \sum_{i=0}^{m-1}\sum_{j=0}^{n} 4MN$ equations, we would be optimising over $\sum_{i=0}^{m-1}\sum_{j=0}^{n-1} 4MN$ equations. The application keeps a running average of distance for every corner association of every marker to marker relationship, this information is stored in the

DistanceStorage.cs class file, each corner for a marker to marker relationship has a bin that holds its running average. When a new frame is captured, the markers are detected, the distances are calculated and then these distances are added to the running average. The rotation of the root bone was constrained to all 0s as the orientation of the root did not matter for this approach, likewise the rotations for the bones were constrained to $-180 \leqslant r \leqslant 180$ degrees. The flow of the tracking part of this approach is shown below:

1. Capture frame from web-cam
2. Identify markers in frame, compute transforms for markers
3. Work out corner distances for markers seen, add new distances to the relevant running average of distances.

Similar to before, the new cost function was simulated with test data. Test data was generated by taking the actual distances from an already rotated skeleton model and applying Gaussian noise to these distances. Since a pair of markers may never appear in the same view, we added in a probability that a pair of markers never have any recorded distance into the test data as well.

For the real-world running, we implemented another processing thread that continuously ran the optimisation in the background whilst updating the model in the 3D model viewer with the new estimated rotation values. This allowed the user to see how close the optimisation is getting to the real-world skeleton model in real-time. This optimisation thread is summarised below:

1. Copy the array of average distances for all marker-marker relationships - these will be used as our observed data for the optimisation equations.
2. Get previous estimated rotation parameters (Initial run will be set to all 0s).
3. Run optimisation for 5 iterations
4. Update 3D model in viewer with new estimated rotations.
5. Go to 1.

Evaluation

This approach produced adequate results very quickly but there were problems with generating very accurate results, especially at the extremities of the skeleton model. In order to test the real-world running of this approach, a couple of test skeleton models were created, known rotations were applied to the model and the application was run against the rotated model. A record of average rotational difference per axis on a bone was recorded against the number of iterations of the optimisation. Figure 19 shows the test model and the rotated test model. Figure 20 shows the results of a run with over

15,000 iterations which took about 30 seconds to complete. Notice how the rotations eventually converge down to about 6 degrees average difference per rotation. The biggest rotational differences are located at the extremities of the skeleton model as seen in Figure 21. The camera was constantly rotating around the model.

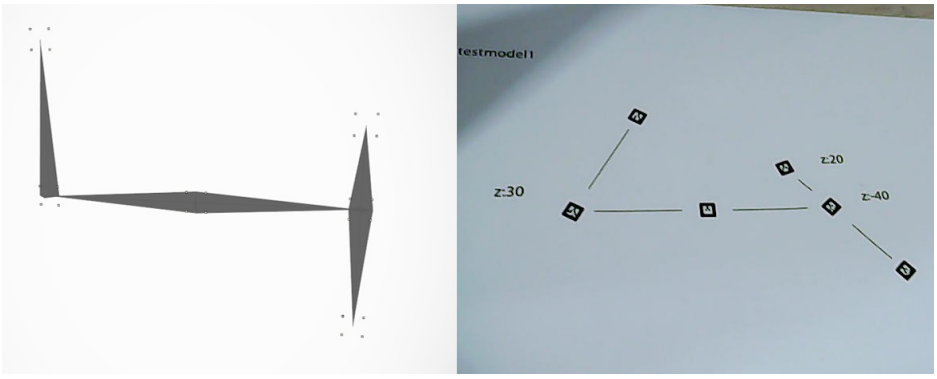


Fig. 19: **L**eft - Inputted model, Right - Model with rotations applied

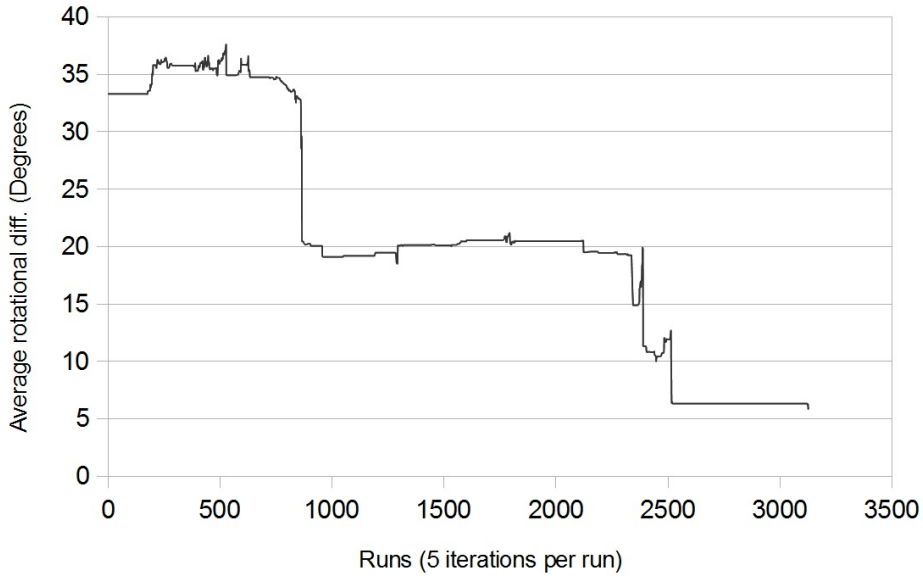


Fig. 20: **R**esults of the distance-corner optimisation

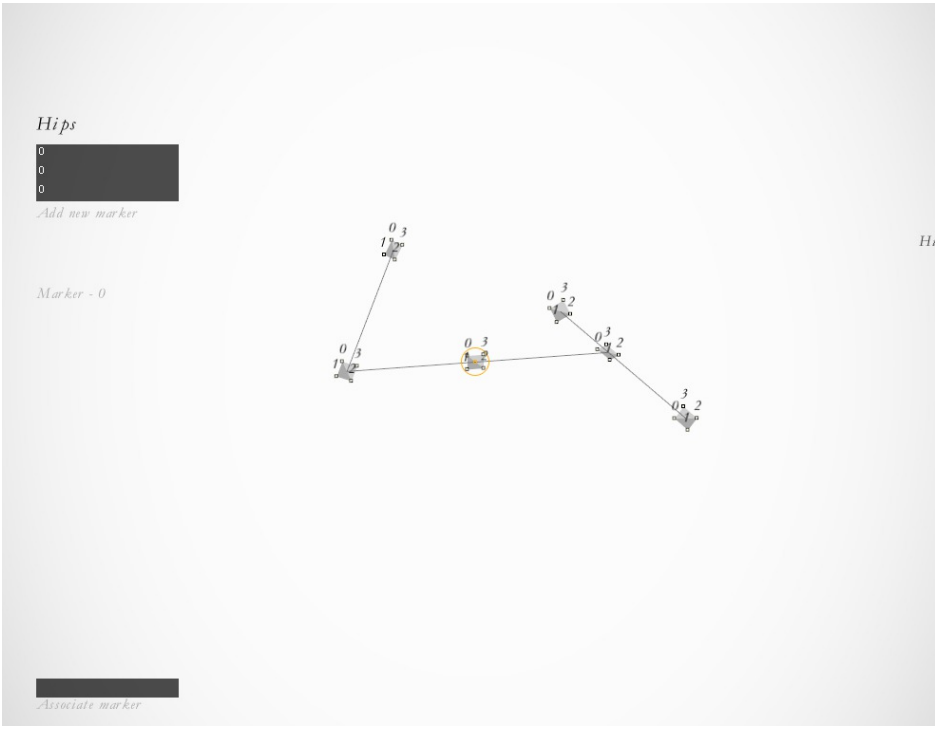


Fig. 21: Showing the final result of optimisation

The results are promising for this approach however there were some key issues involved with only looking at the distances:

- **Overestimation of distance -** Distances recorded between corners tended to be overestimated. The amount of error was dependent on the distance of the camera to the markers and also the angle of the camera relative to the markers. Overestimation of the distance caused problems for the optimisation and is the reason why the end sites of the skeleton model were the most error-prone. It was unlikely that an end site would have another marker on the other side of it to compensate for the overestimation from one side of the model, this caused the optimisation to change the rotation incorrectly in order to accommodate the overestimation of distance. This problem would be exhibited by any marker which is located in an area of the model where only a few other markers can be seen and these other markers are all located close together. The overestimation of distance is likely caused by not perfect calibration of the camera in combination with the fact that the marker detector cannot perform sub-pixel accuracy when detecting the marker causing the markers to be detected slightly too small. This error would increase as the marker appearing in the frame got smaller.
- **Local minimums** - There were still cases when the optimisation got caught in local minimas, the optimisation works best if all markers can see each other. If a certain set of markers cannot see another set there may be many possible rotations that would be valid in terms of reducing the objective function.

The upside of this approach was the speed of running, the problem size was essentially fixed as instead of relying on separate views, we relied on the average distances for each corner of a marker-marker relationship. The downside was that it required 2 or more markers on the screen in order to use that particular view. Another downside was the fact that there were still quite a few rotational errors in the final results mostly due to the overestimation of distance, these rotational errors increased dramatically if the optimisation did not have distances recorded for certain marker-marker relations.

3.7. Final approach: Marker projection from view

A final approach was devised that took inspiration from both initial approaches. We still did not want to be reliant on the global camera pose, however we wanted to be able to take each individual view into account and be able to optimise over a number of selected views. Instead of looking at the distances between corners in each view, we project out the marker corner positions from a selected marker in the scene and aim to minimise the 2D euclidean distances between the seen marker corners and the projected marker corners. Figure 22 shows the inputted model shown in Figure 19

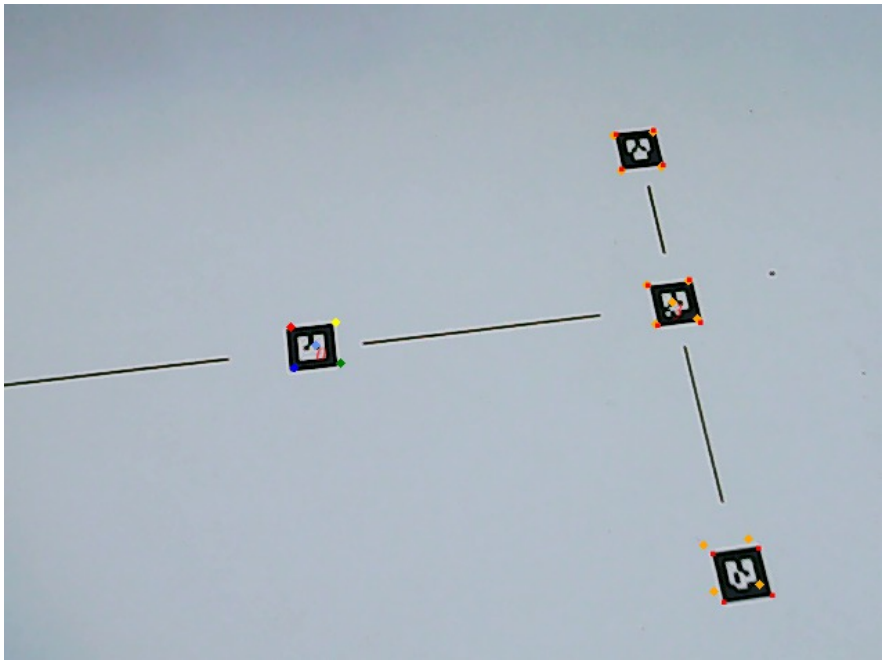


Fig. 22: **E**xample showing the projected marker corners based around the marker with ID 0

projected onto a real-world model without any rotations applied. The projection is based around the marker with ID 0, orange points are the projected corners, red points are the detected corners.

3.7.1. Objective function

We first needed to define the new objective function to minimise. The objective function will be looking to minimise the 2D distances between where the model expects marker corners to be from the estimated rotations and where they are actual seen in the frame, this is a similar approach to the PTAM approach after-mentioned. The key difference is that instead of using a global camera pose, we will use a camera pose where a single marker is selected to be at the center. The estimated skeleton model will be transformed so that the same marker on the skeleton model will be matched up with the marker seen in the camera. If we know the transform matrix of a marker in the view, we can transform every marker corner in the skeleton model as in Figure 23 to obtain its coordinate in the chosen markers coordinate frame.

$$T = M(C^{-1})WP$$

$$V = [T_{40}, T_{41}, T_{42}]$$

Fig. 23: Getting vector position of a marker corner

Where T is the transform matrix of the marker corner in the camera coordinate frame and V is the 3D vector coordinate of that marker corner. M is the marker corner transform in the root bone coordinate frame. C is the marker transform of the chosen marker in the root bone coordinate frame. W is the world transform matrix, i.e. the camera transform obtained from the chosen marker and P is the projection matrix.

Knowing the vector position of the marker corner in the view, we can now do a projection to get it into screen coordinates using the formula in Figure 24.

$$u = \frac{Xf}{Z}\ ,\ v = \frac{Yf}{Z}$$

$$x = \frac{1}{2}Width + u\frac{1}{2}Width\ ,\ y = \frac{1}{2}Height + v\frac{1}{2}Height$$

Fig. 24: Projecting to screen coordinates

Where (u, v) are coordinates on the viewing plane, (x, y) is the pixel coordinates on the screen, f is the focal length (in mm) and (X, Y, Z) are the 3D coordinate of the corner in the frame of the camera space. $Width$ and $Height$ refer to the width and height of the screen in pixels.

The actual marker corners seen in the view can be extracted from the relevant marker transform matrix. Since we know the edge size of a marker, we can take the corner as a translation from $[0, 0, 0]$ and multiply this translation matrix by the camera transform from a specified marker. We can then use the same projection formula as in Figure 24 to get the position in screen coordinates.

Now that we found the two marker corner screen coordinates, the objective function is to minimise the distances between them. This leaves us with two problems, firstly how do we choose which marker is to be used as the chosen marker in each view. Secondly how do we choose which views to use?

The first problem was solved by ranking the markers in each view depending on how accurate its pose estimation was likely to be. The intuition was that since some of

the marker corners were far far away from the chosen marker position, any error in pose estimation would be exacerbated the further the point was away from the chosen marker. Choosing the most accurate marker meant that we reduced the likeliness of a very bad pose being used which would result in a lot of error when calculating the 2D point difference. 2D size of the marker in the view was used to rank how accurate the pose estimation for a marker was likely to be. The larger the area on the screen, the less likely that sub pixel inaccuracies when detecting corners would occur. Figure 25 shows the formula to get the area of the marker in the view.

$$A(\Theta) = \frac{1}{2} \mid (V_2 - V_0) \times (V_3 - V_1) \mid$$

Fig. 25: Area of a 2D quadrilateral

Where V_i is the 2D coordinate of the ith point.

We can define the objective function in terms of the set of equations for each view. We use the above ranking to obtain the marker that is most accurate within a view, the next step is to project the estimated skeleton model based on the chosen marker. The next step is to project out the corners for each seen marker in the view. Finally we can get the 2D distance as shown by the formula in Figure 26.

$$\sum_{i=0}^{n-1} \sum_{c=0}^{3} (E_{ic} - V_{ic})^2$$

Fig. 26: 2D distance between marker corners

Where n refers to the marker count, c is the corner index, E_{ic} is the calculated screen coordinates of marker i and corner c from the estimated skeleton model and V_{ic} is the calculate screen coordinates of the marker i and the corner c in the view.

Since the computational time for the optimisation increases exponentially with the problem size as it is an $O(n^3)$ problem, it was decided to limit the views per run of the optimisation to 30 views. Another improvement is to not always run a global optimisation looking to optimise the rotations for all the bones in the model. We can run localised optimisations that only looks at a subset of bones to optimise over. The skeleton model makes it easy to divide up the optimisation into localised sections since we know that to estimate the rotation of a bone, only the bones in the route to the root bone would affect its final position. This is illustrated in Figure 27.

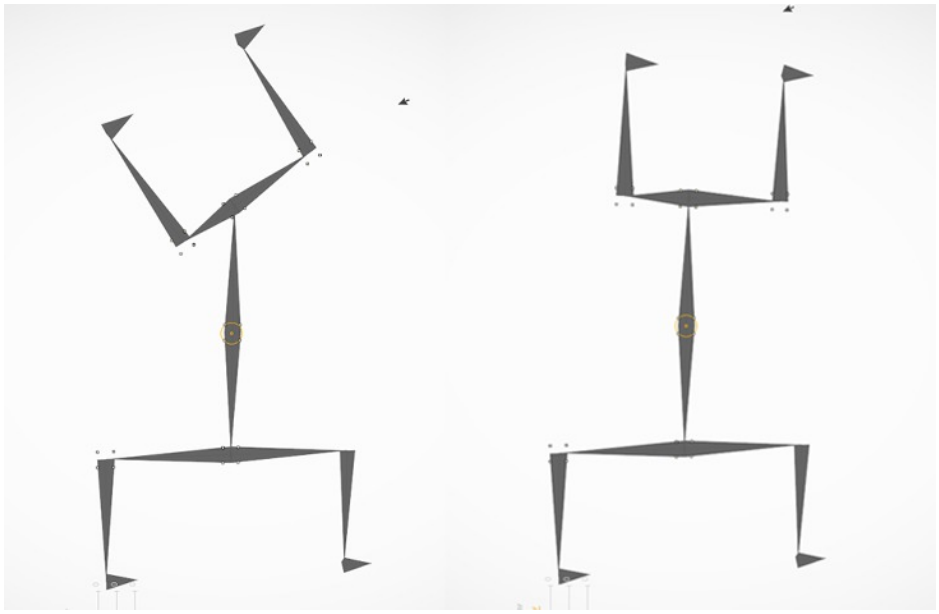


Fig. 27: Local optimisation
Notice how a rotation of the parent bone of the top half will not affect any of the positions of bones in the bottom half.

If we represent the skeleton model as a tree, we can perform a local optimisation by moving from the rootbone and picking our way down the tree until we reach a leaf node. A leaf node represents an end site of the skeleton model. This path through the tree can be further reduced in size by picking a connected section within it to perform the optimisation over. We would only need to consider markers that are related to the bones in this selection. The views that we would use for the local optimisation must have 2 or more of these markers in them otherwise they would be worthless to the local optimisation. Preferably we want the views where the most accurate marker in the view is one of the markers in the selection in order for the results from the view to be less error-prone.

An outline of the localised selection is below:

1. Start at root node
2. Select one of the children, add to selection
3. 25% of adding any of the other children to selection
4. Travel to first selected child
5. If child is end site terminate, else go to 2.

Since the computation time of the optimisation is exponential with the number of bones, by doing a local optimisation over a subset of the bones we are drastically reducing the time it takes per iteration of the LMA optimisation.

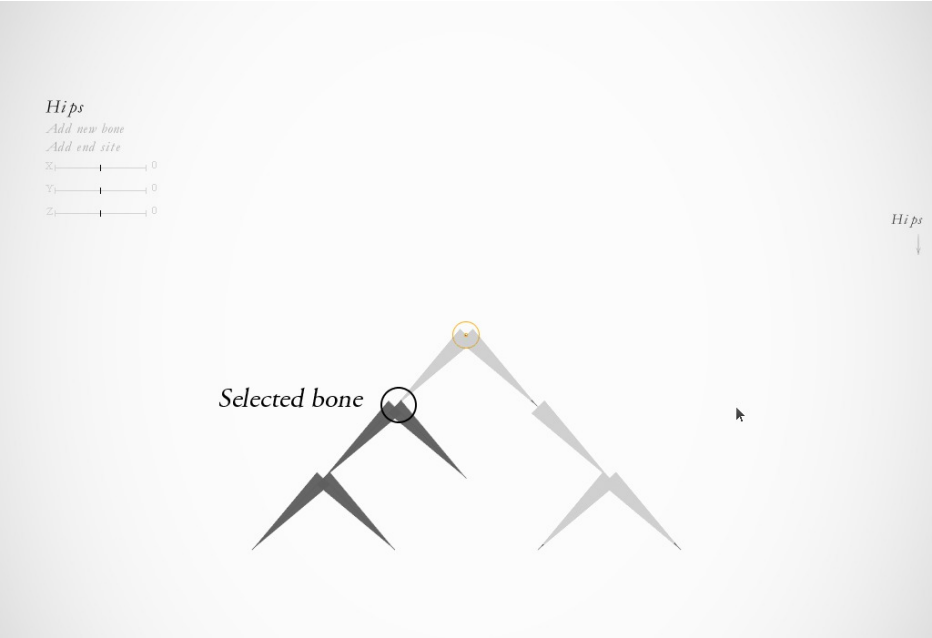


Fig. 28: **B**one tree representation
Only the highlighted bones will be affected by the rotations of the chosen bone

There is one more heuristic we can take into account to minimise local minimum problems with the optimisation step. We can take advantage of the fact that we can estimate the orientation difference between the two connected bones by comparing the camera transform matrices between the markers on both bones. We can use this value to constrain the search for the rotation to up to +/- 20 degrees from the estimated value. We only want to do this if we have adequate data to support that the rotation estimation is correct so we take the average of all views showing the rotation and only use it as a constraint if we have over 120 views adding up to the average. We also keep a running variance value for each rotation, based on this variance we can limit the constraint even further, a low variance can make the constraint as low as +/- 8 degrees for the rotation.

$$(M_i R_i^{-1})(M_j R_j^{-1})^{-1}$$

Fig. 29: Finding rotation transform from one marker to another
Where M_i refers to the marker transform of i and R_i refers to the rotation matrix part of the transform of i.

The equation count for optimising across all bone rotations is therefore $Viewcount \sum_{i=0}^{n} \sum_{c=0}^{4} (E_{ic} - V_{ic})^2$ Whereas a local optimisation is the same except that n represents only those markers included in the local optimisation.

3.7.2. Implementation

We can now outline the entire algorithm in two sections. The first section deals with the tracking part where we detect the markers in a view and process the view in preparation to be used in optimisation. The second section deals with how the optimisation chooses the views it is going to use and how it determines the constraints to use for each rotation parameter.

Tracking

Whenever a new frame is captured the following happens:

1. **Detect markers in scene** - We use ALVAR marker detection in order to detect markers and associate an ID and camera transform to them.
2. **Check distances** - We can know the maximum possible distance two markers can be apart from the skeleton bone model. This is essential the max distance that the markers can be apart in 3D that the skeleton model allows. We can precompute this distance for each marker-marker relation before the optimisation begins. If a distance between two markers is larger than this maximum distance by more than 10mm we can discard the entire view as it is likely that the marker detector has incorrectly labeled one of the marker IDs. If we know that at least one marker has been incorrectly labeled, it is likely that the tracking of the view was not ideal anyways. Similarly, we can discard any view where the closest marker appears more than 300mm from the camera, the pose estimation is unlikely to be very accurate.
3. **Work out rotation between connected bones** - If two markers on connected bones appear in the frame, we add the [X,Y,Z] rotation between them to the running average of rotation for that particular bone pair in the **ViewStorage.cs** class file. We also increment the count seen for that bone pair rotation.

4. **Rank marker accuracy** - We rank markers within the view by the area that the quadrilateral formed by its corners in 2D make, it is more likely for a marker pose to be accurate if it appears larger in the view.
5. **Construct view object** - A new view object is created that includes information such as the markers seen in the view and the associated camera transforms. This view object is stored along with all other views in the **ViewStorage.cs** class file. The view is placed into a bin that corresponds to its most accurate marker. A reference of this view is also placed into a bin corresponding to its 2nd most accurate marker.

Optimisation

Optimisation happens in a separate thread. Every run of the optimisation does 5 iterations with 30 views.

1. **Decide whether to do local or global optimisation** - There is a 50/50 chance of doing either. If we choose to do a local optimisation we use the selection process in order to select the bones to optimise over. From this we can then modify the equations to only take into account markers on the chosen bone and the bones in its sub tree.
2. **Determine constraints** - There are two constraints that can be applied. Firstly we can look at the user inputted constraints, secondly we can look at the constraints implied by the average rotations seen in the tracking stage. If the two constraints are not overlapping we take the user inputted constraint as the constraint for optimisation and we also clear the rotational average determined from the tracking stage, we assume that the user has constructed the model correctly and that the seen rotation is impossible on the model. Otherwise the final constraint is the lower and upper bound from the rotational and user inputted constraints that has the least difference.
3. **Collate views** - We select 30 random marker IDs and pick views that have these marker IDs as their primary or secondary accurate markers, we bias the selection of views to pick ones that have been seen more recently, this is to avoid not taking in new information once the view count gets too large. This gives us a good spread of views as the logic to pick the views is not skewed towards markers that are seen a lot. If we are doing local optimisation, we can only randomly pick marker IDs corresponding to the markers of the bones we are optimising. Also we don't pick views that do not have 2 or more of the markers we are optimising over.
4. **Run optimisation** - We can now perform the optimisation. The input estimated rotation parameters are the results from the previous run (all 0s for initial run).

For every view we take two sets of equations, the first set uses the most accurate marker as the chosen marker, the second set uses the second most accurate marker. If we are working out the objective function on a view that does not contain a marker that we want to include in the optimisation, the difference for any equation that includes that marker is set to 0.

5. **Update 3D model** - Finally we can update the 3D model viewer with our new estimated rotation parameters.

Another improvement over previous approaches is that we can now see clearly when the estimation is poor in the UI. It now overlays onto the camera feed the estimated marker corner positions centered on the most accurate marker in the current view as seen in Figure 30. The 2D points are re-distorted using the brown's distortion model so they match up with the marker corners on the screen.

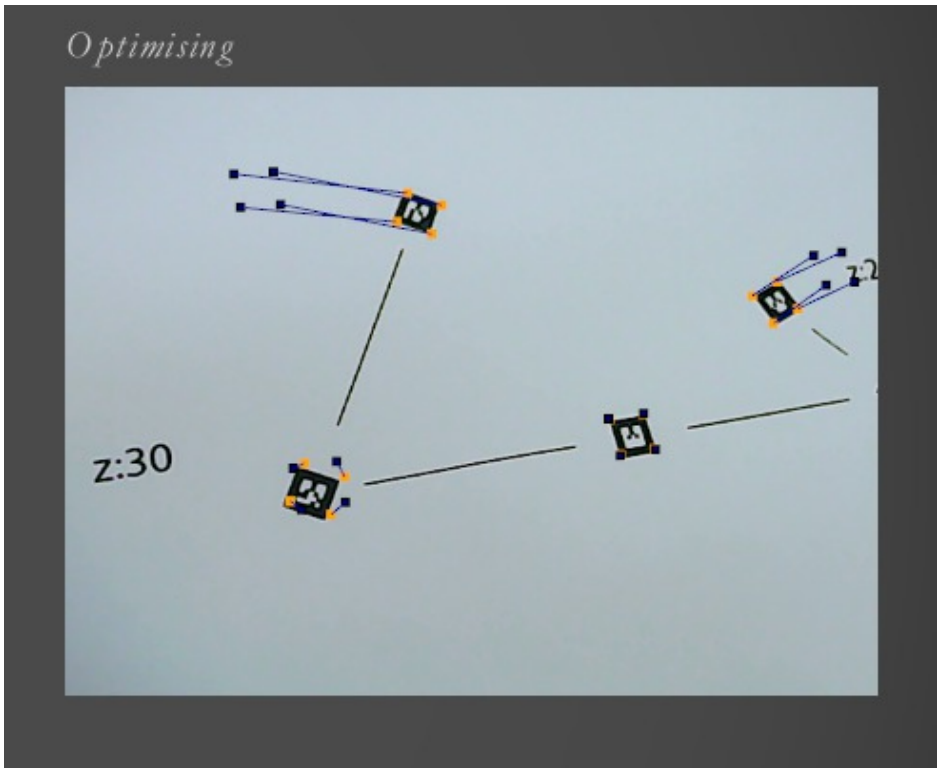


Fig. 30: **UI** showing estimated corner positions

4 Evaluation

The evaluation of the final application will be done with quantitative analysis as well as qualitative analysis. We will be looking at how accurate the estimation of the rotations gets on a number of skeleton models which we know the exact rotations of. We will also look at each of the improvements to the core optimisation explained in the previous section and see how they affect the final estimation result. Finally we will be doing a walk through of the final application on a more complicated model to show how intuitive it is to use.

4.1. Quantitative analysis

We will be performing the optimisation on each model for 35 runs of 5 iterations each. The attached graphs will show how the average rotational difference for a single rotation changes over the runs. The algorithm does not expect the same views for every run and there are non-deterministic parts of the algorithm as well so we do not expect every run to generate the same results. To help alleviate differences due to changing observations and randomness we will be doing 5 passes of 35 runs for each model and taking the average of the results of all passes. The camera will try and keep on a similar path for every pass performed in order to give the optimisation similar views for every pass, however there is still some randomisation due to noise and view selection. The camera will be moving in a circular path above the model whilst directed towards the model as shown in Figure 31. We will be using two main metrics for the evaluation, firstly we will look at the 3D point difference. This is done by looking at the average distance between where the bone is estimated to be in the skeleton model and where the bone actually by applying the pre-measured rotation values to the skeleton model, we take a reading of this every run. Secondly we look at the average rotational error for an axis of rotation on a bone, this is the difference between the estimated rotation and the actual rotation of the real-world model, this is recorded every run as well. We will also look at the variance against runs for both metrics.

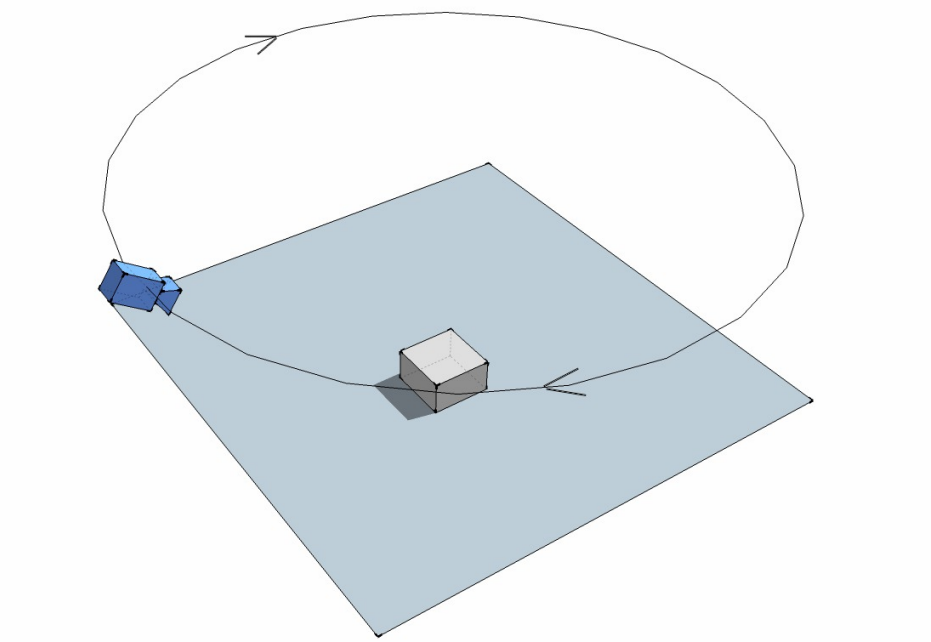


Fig. 31: Camera path

4.1.1. Base model 10 markers

We will start by testing on a model with 10 markers. Figure 32 shows this model with no rotations applied. No constraints have been applied to the bones since we want to test the accuracy on all axis (X, Y and Z).

Bone Index	X offset	Y offset	Z offset	X constraint	Y constraint	Z constraint
B0	0	0	0	0	0	0
B1	0	40	0	N/A	N/A	N/A
B2	-20	20	0	N/A	N/A	N/A
B3	-30	0	0	N/A	N/A	N/A
B4	20	20	0	N/A	N/A	N/A
B5	-40	0	0	N/A	N/A	N/A
B6	-30	0	0	N/A	N/A	N/A
B7	0	-30	0	N/A	N/A	N/A
B8	40	0	0	N/A	N/A	N/A
B9	-20	20	0	N/A	N/A	N/A
B10	0	-20	0	N/A	N/A	N/A

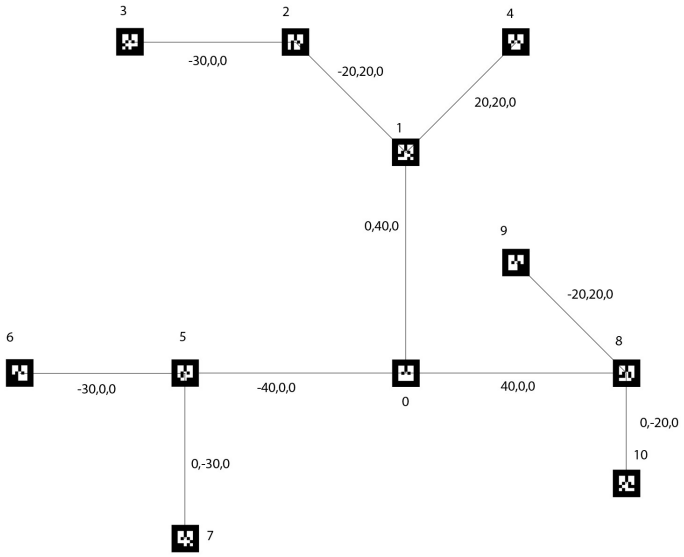


Fig. 32: 10 marker model

We will now test with the rotations as shown in Figure 33.

Bone Index	X	Y	Z
B0	0	0	0
B1	0	0	-15
B2	0	0	-30
B3	0	0	0
B4	0	0	0
B5	0	0	100
B6	0	0	-50
B7	0	0	20
B8	0	0	50
B9	0	0	-50
B10	0	0	0

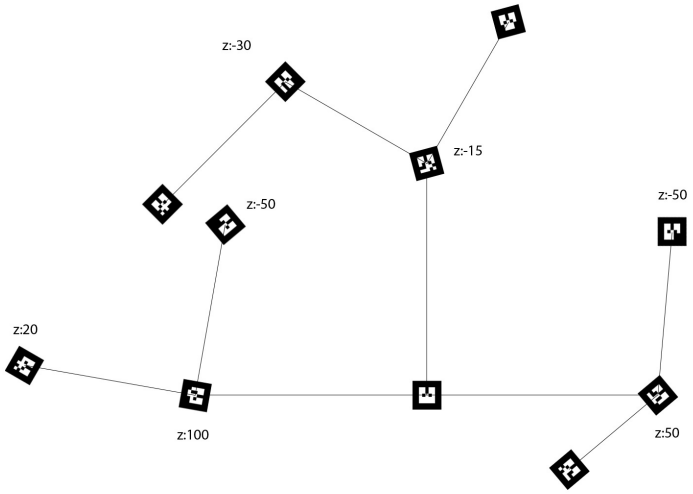


Fig. 33: 10 marker model, rotations 1

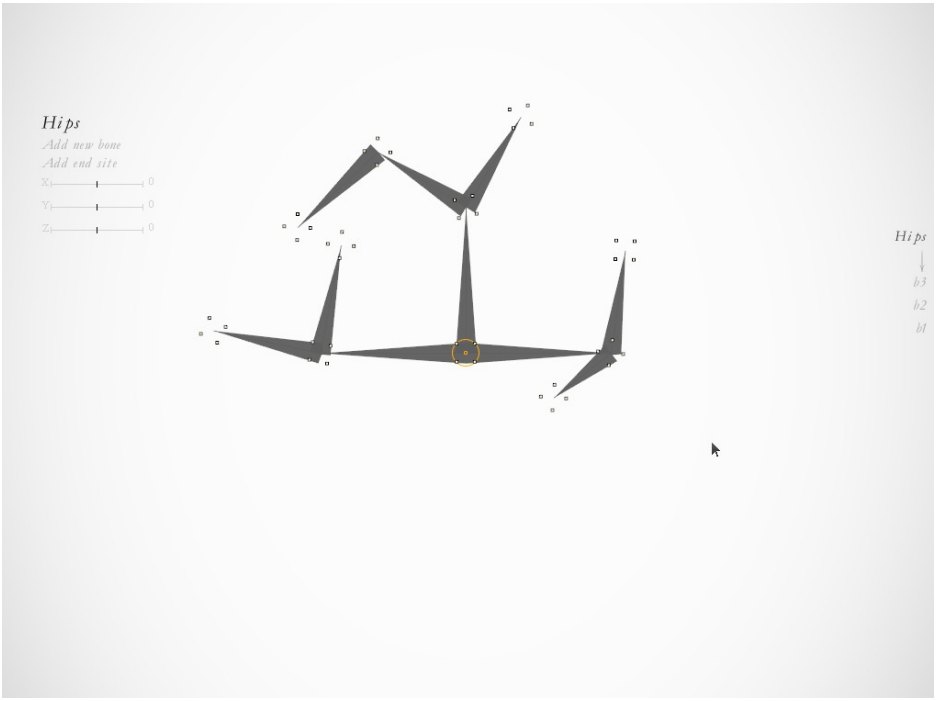


Fig. 34: **A**fter optimisation

The first graph shows the average 3D point difference for each bone in millimeters. Notice the rapid convergence after less than 20 iterations. This is mostly due to the rotation constraints that are applied to the optimisation. It is also worth noting that the variance of 3D point difference for each run was progressively getting low, each bone was exhibiting roughly the same 3D point difference due to the fact we are optimising over all markers in a view. If we only took into account rotational information across connected bones, we would expect a larger 3D point difference near the extremities of the model. At the end of the optimisation, we can see that bones came to roughly 1mm of the expected position which is an excellent result. The 1mm error could be due to a number of factors such as systemic error due to calibration or noise.

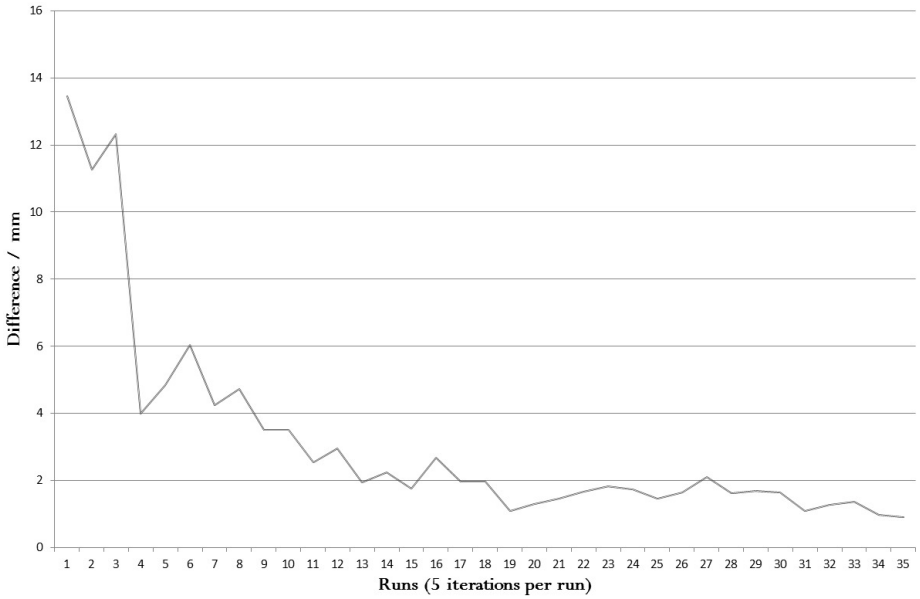


Fig. 35: 3D point difference

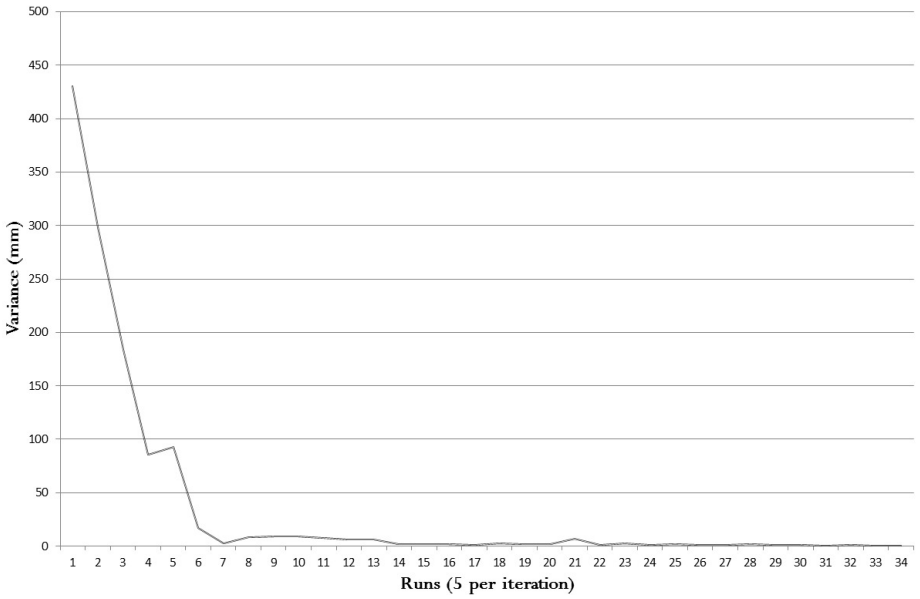


Fig. 36: 3D point difference variance

The graph in Figure 37 shows the average rotational error for each rotational axis on each bone over the number of runs performed. It roughly follows the same shape as the 3D point difference graph with a rapid convergence towards the correct pose around the 5th run (25th iteration) and the stabilisation towards the 30th run. It also shows we can expect the optimisation to reach a state where each rotation is roughly +/2 degrees off the expected rotation. Unlike the 3D point difference we noticed a greater variance on rotational errors, this is likely due to the self-correcting nature of the optimisation. The 2D point differences are reduced to obtain a more acceptable general pose, sometimes at the cost of worse rotational error on some of the bones in the model.

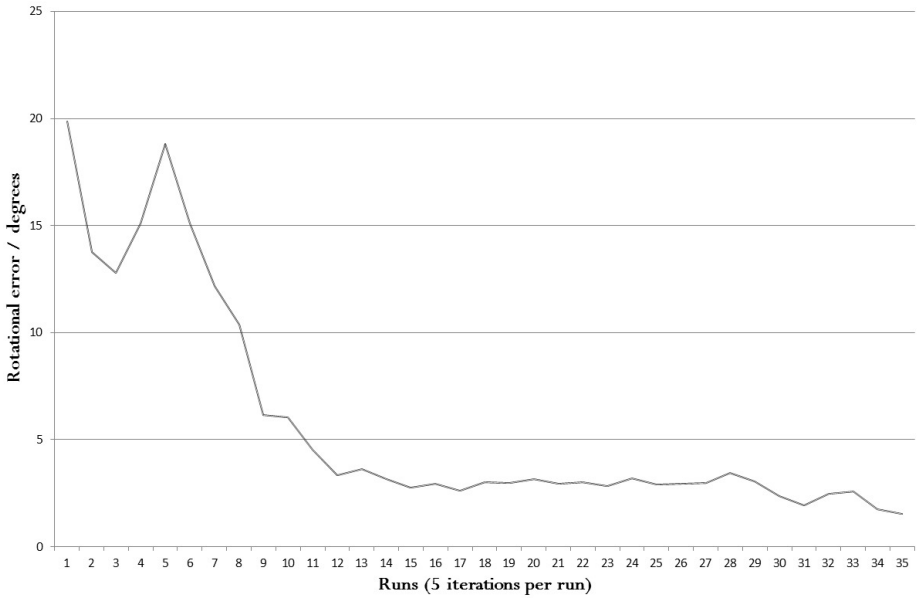


Fig. 37: **R**otational error

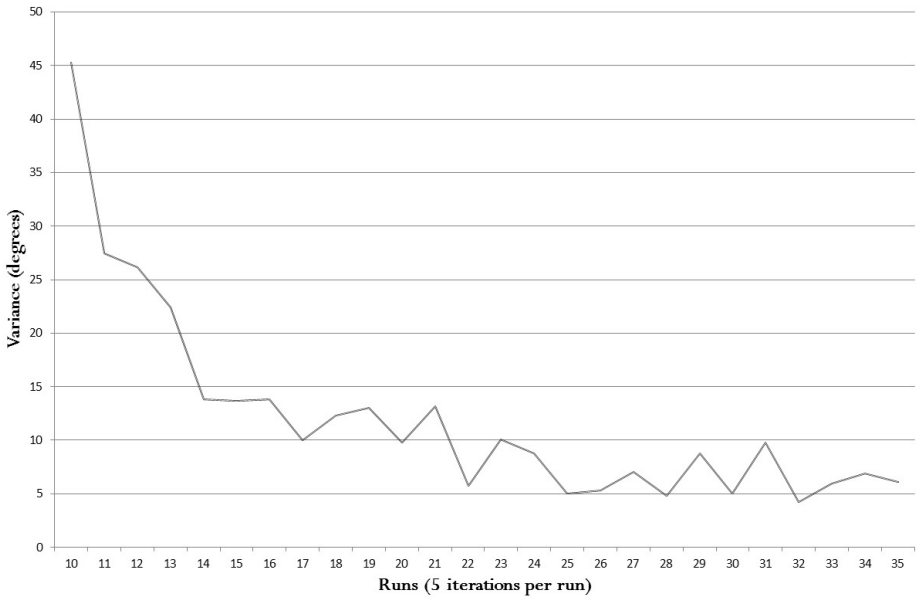


Fig. 38: **R**otational error variance

The second test will use the same model with different rotations applied as in Figure 39. This test was also performed under less than ideal lighting conditions.

Bone index	X	Y	Z
B0	0	0	0
B1	0	0	115
B2	0	0	120
B3	0	0	0
B4	0	0	0
B5	0	0	-40
B6	0	0	-30
B7	0	0	80
B8	0	0	-70
B9	0	0	100
B10	0	0	10

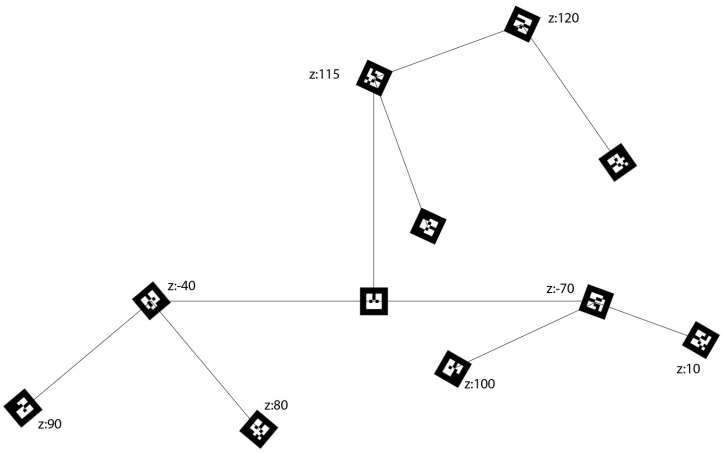


Fig. 39: 10 marker model, rotations 2

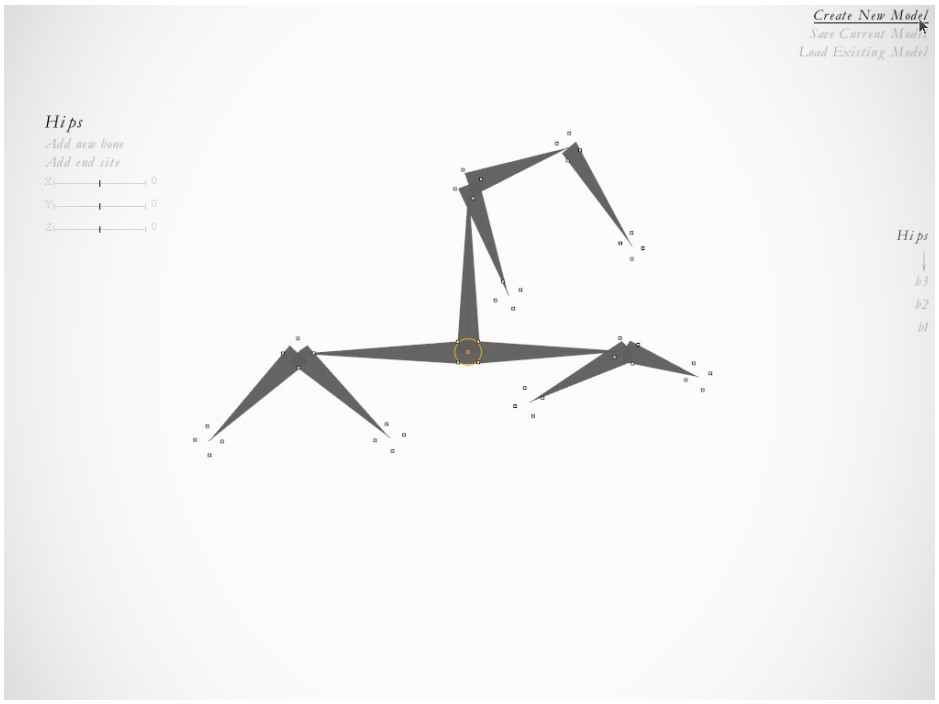


Fig. 40: After optimisation

Figure 41 shows the 3D point difference. It followed a similar shape to the 3D point difference from before however the convergence is not as pronounced. Notably this optimisation was done in less than ideal lighting conditions which would have caused problems with the quality of available views. Noise would have played a bigger factor causing the marker pose estimations to be of poorer quality. Another problem would be that some markers may have gotten mistaken for others. This is reflected by the less stable optimisation shown in the graph, having a good selection of views would have a much greater importance than before due to the fact that some views could adversely affect the final result of the optimisation.

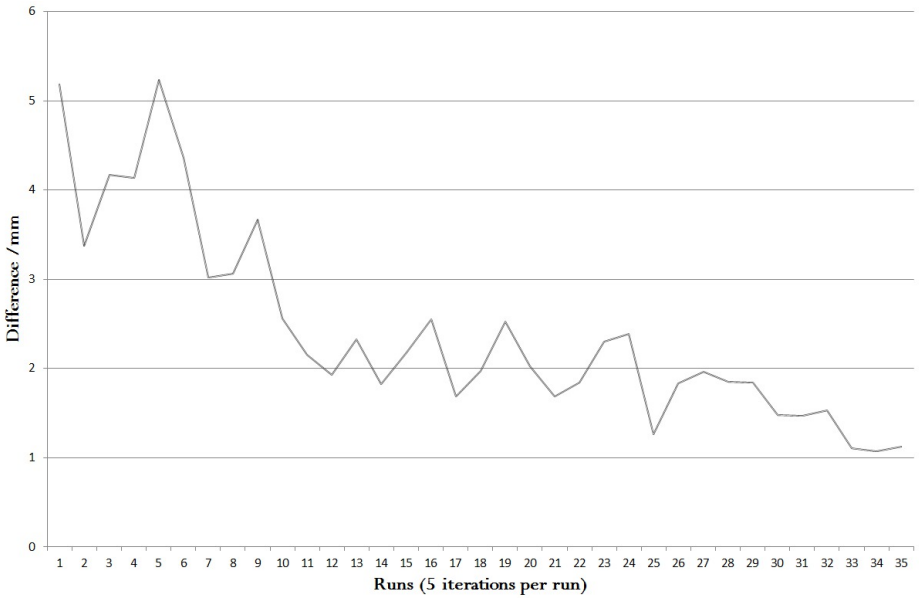


Fig. 41: 3D point difference

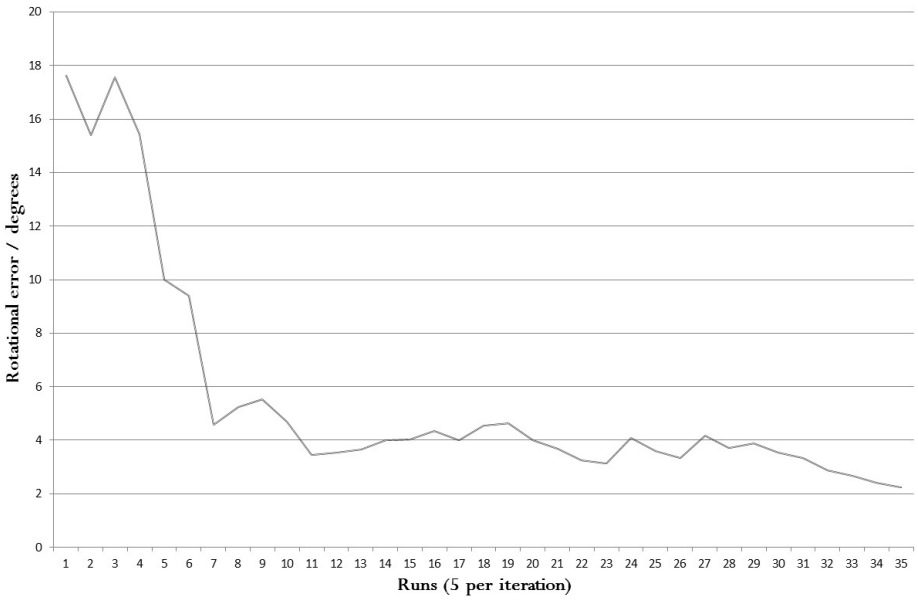


Fig. 42: **R**otational error

We will now look at how the rotation constraints affects the optimisation stage of the process. The next test will be using the same rotations as in the first 10 marker test. However we will be turning off the rotation constraints and allowing the optimisation to use any value between -180 and 180 degrees when estimation the rotations.

Fig. 43: **F**inal estimation with no rotation constraint

Figure 44 shows that even though the rotation constraints were removed, we have still obtained an accurate estimation of the skeleton pose. It may seem that the rotation constraints may not add anything to the process, however it does improve on how fast the optimisation converges to an accurate and stable solution. Notice how the optimisation took nearly 50 runs to become stable with an accurate result compared to the 30 runs needed for the optimisation with constraints applied. This is because the optimisation has to search across a larger range to get to the final estimated rotation. Using the rotation constraints can also reduce the likelihood of local minimums affecting the final pose due to the fact that the optimisation will start near to the actual skeleton pose instead of starting at all 0s for rotations. Figure 45 shows the rotational error, it is very high initially due to the fact that we are starting at all 0s rotation and eventually gets close to the accurate rotation values.

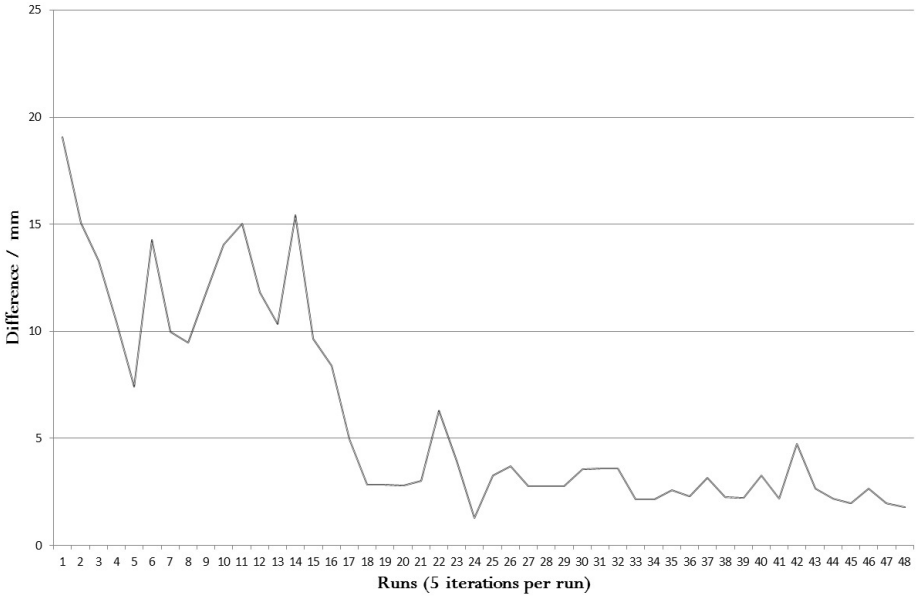


Fig. 44: **3**D point difference - No rotation constraints

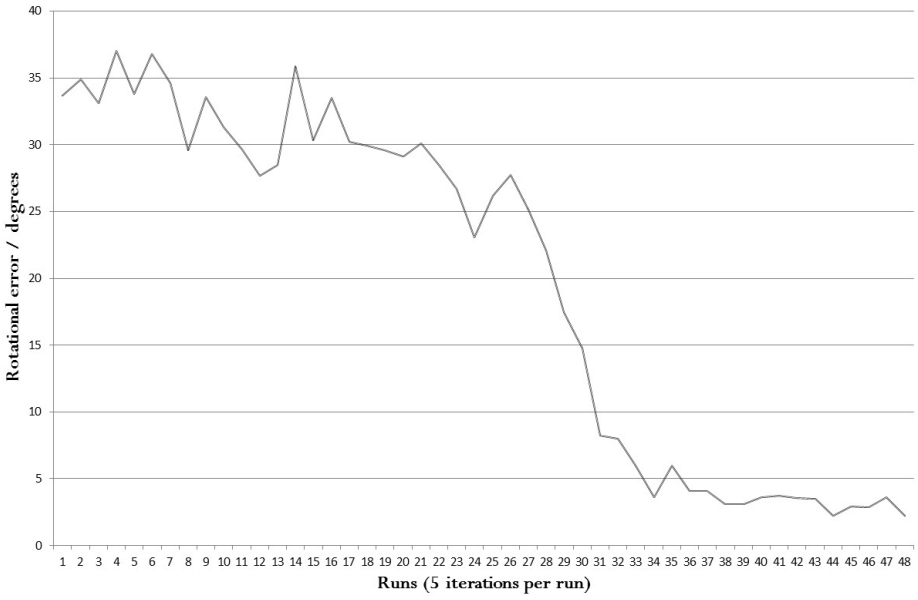


Fig. 45: **R**otational error - No rotation constraints

4.2. Qualitative analysis

In this subsection we will walk through the process of using this tool from end to end. We will be doing the walk-through on the model as seen in Figure 46 starting with the creation of the model in the UI through to performing the optimisation to estimate the skeleton pose. It has 10 bones in total, each with its own marker, 5 of the bones are connected by hinges that allow movement in only a single axis and 4 of the bones are connected using ball joints allowing complete freedom for rotations.

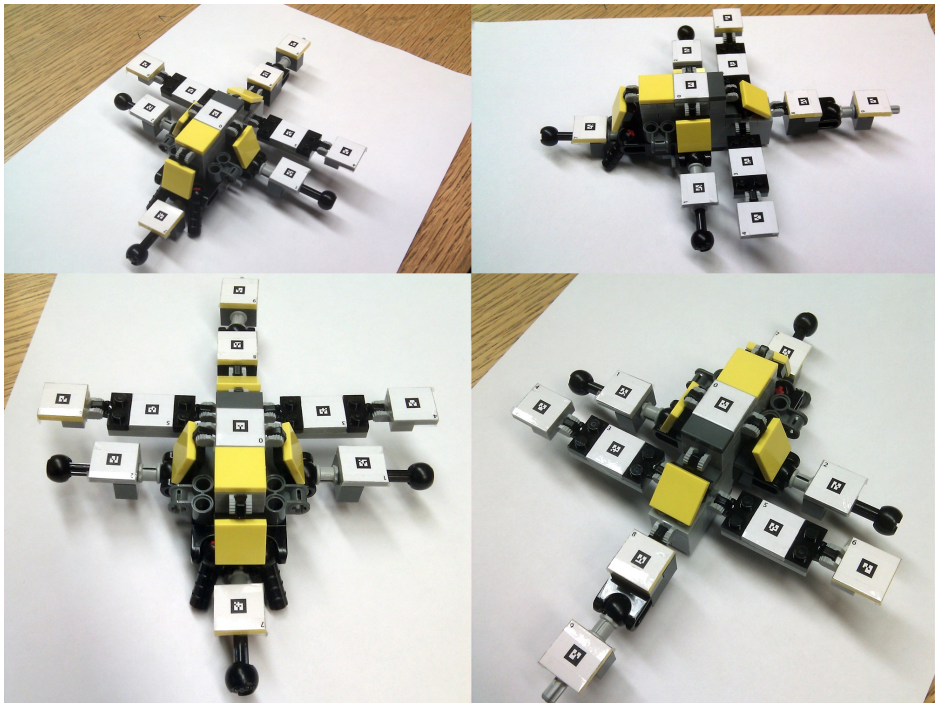


Fig. 46: **r**eal-world model

4.2.1. Creating the model

We have pre-measured the offsets of the bones and markers so all we need to do is input them into the program. We start by selecting ``Create a new model'' in the UI. To add a new bone, click on the parent bone to select it, click on ``Add new bone'' and type in the name of the new bone and the X,Y,Z offset from parent. Finally click ``create'' to finish adding the bone. The 3D model viewer will update to reflect the new bone added to the model and the new bone will automatically become the selected

bone. Also note you can change the X,Y,Z rotation orientation of any bone of the model by clicking on the bone in the 3D viewer and adjusting the sliders that pop up on the left of the UI. It is also possible to set the constraints on rotation on the bone by moving the constraints marker on each slider to the appropriate value. Another useful feature is the bone hierarchy viewer on the right of the UI, it shows the parent of the current bone and lists the children underneath as well. Clicking on the name of any bone in the bone hierarchy will make that bone the selected bone.

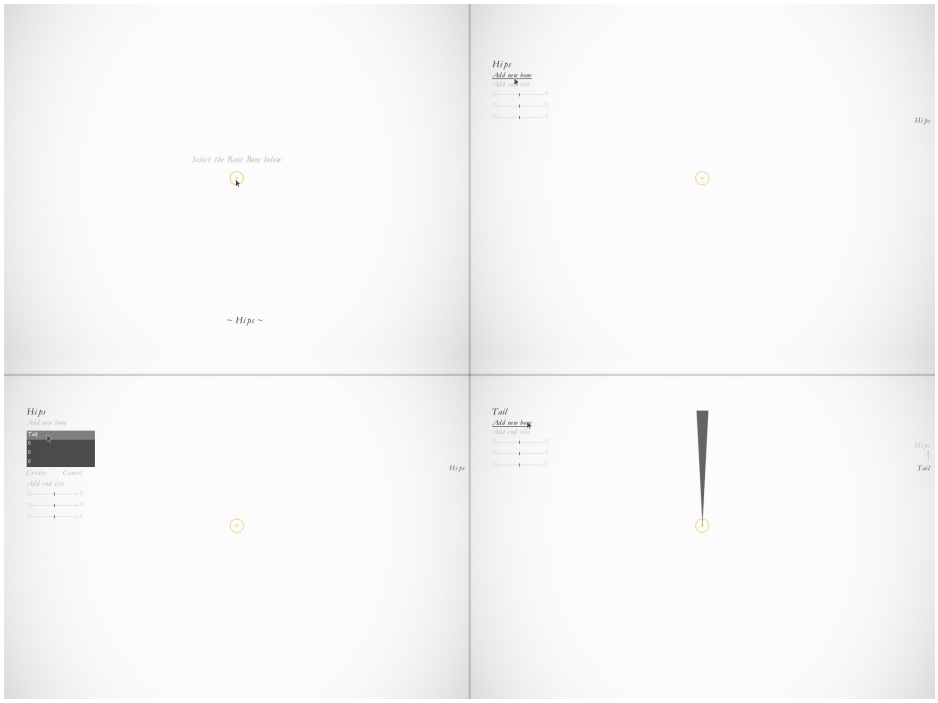


Fig. 47: **C**reating a new model

Top left - Starting a new model, Top right - selecting to add a new bone, Bottom left - Inputting the name and offset, Bottom right - after adding the tail bone.

After creating the complete skeleton model we will need to add markers in order for the optimisation to recognise the bone positions. First we will click on the marker mode button on the top left of the UI, this mode displays as the markers on the model showing the corner orientations and positions of each marker. To add a new marker to a bone, select the bone in the 3D viewer and type in the X,Y,Z offset of the marker in the input boxes on the left of the UI. Clicking on ``Add new marker'' will add the marker onto the selected bone. To modify the rotation of a marker on a bone select

the bone in the viewer. A list of markers attached to that bone will appear on the left of the UI. Clicking on the marker in that list will select the marker, this is reflected by the orange highlighting of the selected marker in the 3D model viewer. The X,Y,Z rotation sliders will appear on the left. The application will automatically assign each marker an ID based on its internally created bone model tree. It is possible to override this behavior and assign your own IDs to a marker by using the text input box on the bottom left whilst having a marker selected. When moving back to bone editor mode, the corners of the markers will still be visible on the model.

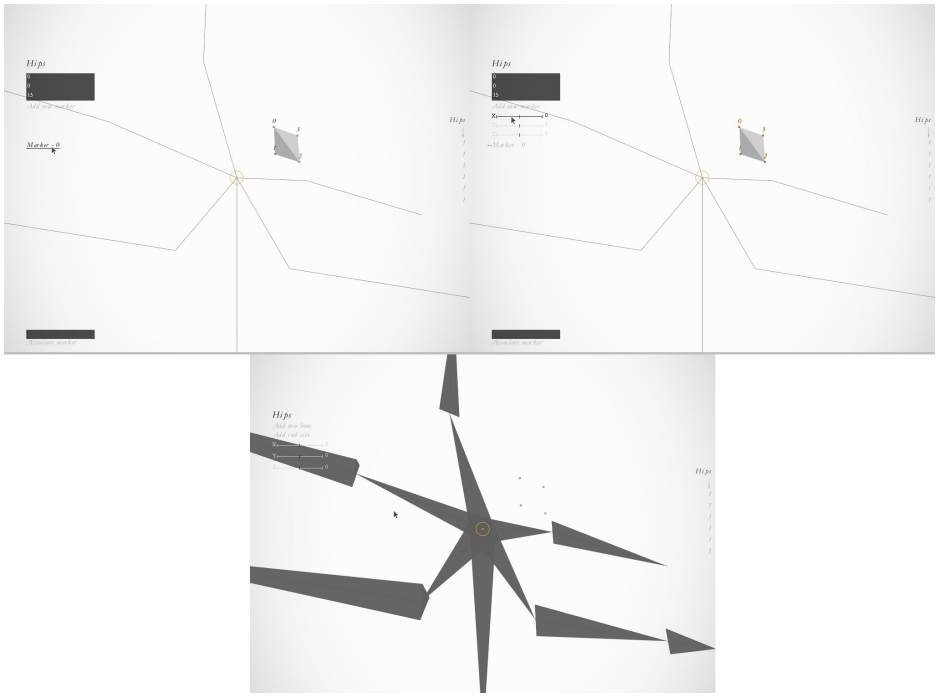


Fig. 48: **A**dding the markers
Top left - Adding a marker, Top right - Selected marker and changing orientation, Bottom - Marker appearing bone mode.

After we have added all the bones and markers we should save the model. To do this, click on ``Save Current Model'' in the top right of the UI. A text input box will appear, type in the name you wish to save the model under and click save to confirm. If we are overwriting an existing model, a dialog box will appear asking you to confirm you really want to overwrite an existing model, otherwise the application will take a screenshot of your model and save it into the <Documents>/figurate/models/<ModelName>/ folder. To load an existing model click on the ``Load existing Model'' button on the

top right of the UI. The UI will show display pictures of all saved models, simply click on the model you wish to load. You can also type the name of the model you want to load in the text input box, the saved models that have a prefix matching the typed name will be highlighted in the display. The saved model will be loaded with all the rotations and offsets specified as when you saved it.

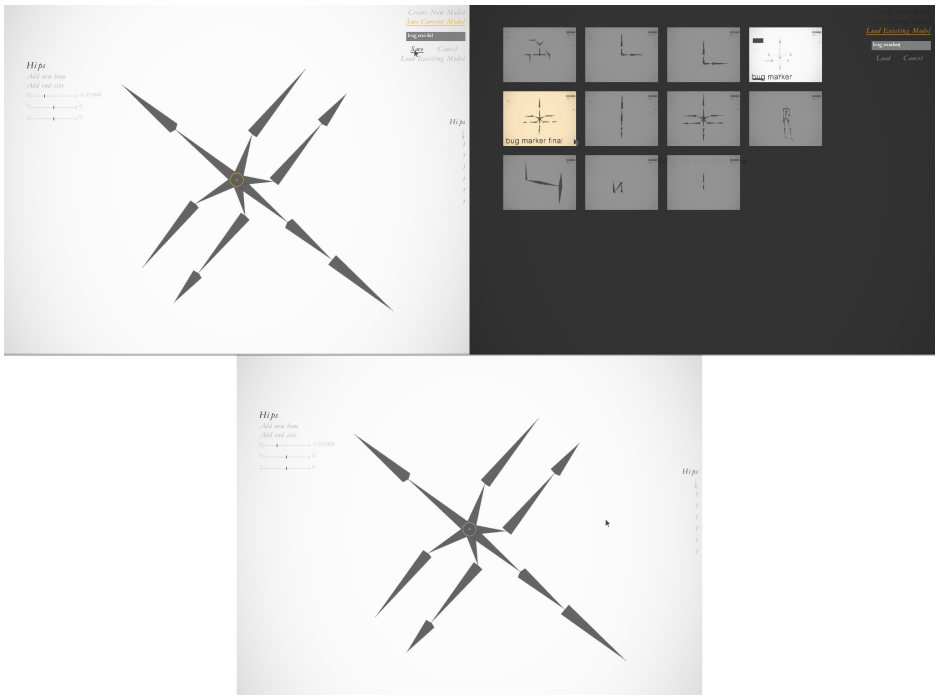


Fig. 49: Saving and loading
Top right - Saving the bug model, Top left - Loading the same bug model.

4.2.2. Estimation of model pose

We have completed creating the model and placing the markers in the previous section, we will not go on to explain how to estimate the model pose from a real world model. The rotated model we will be using for this example is show in Figure 50.

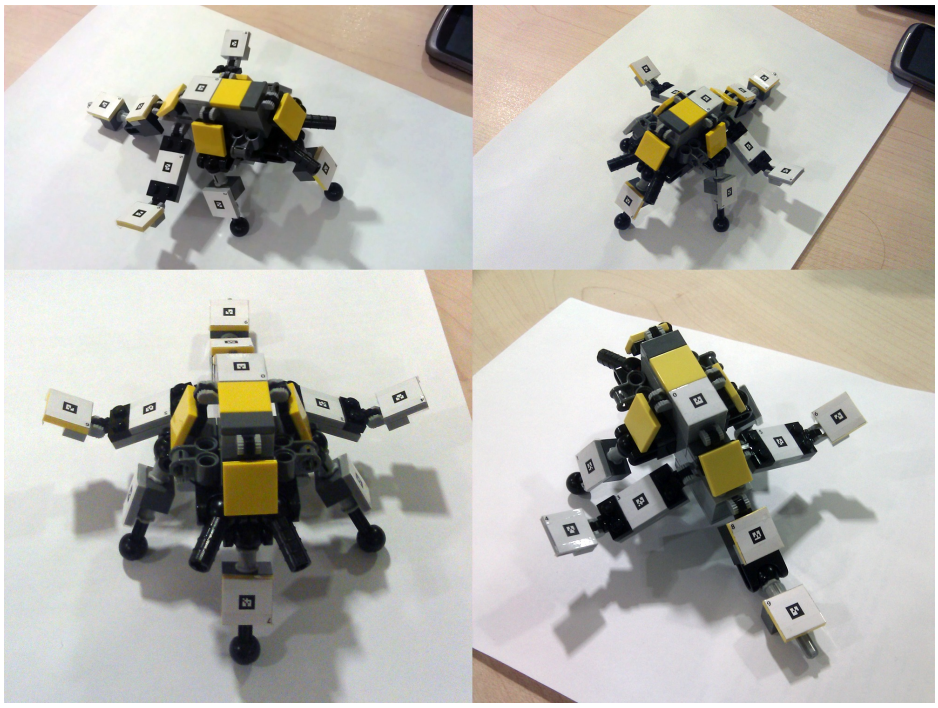


Fig. 50: **R**otated real-world model

To start the optimisation, press F2. The UI will change to show the camera feed on the right side and the 3D model viewer on the left side. You can still use the mouse to navigate and rotate around the 3D model during the optimisation. 300 usable views have to be collected before the levenberg marquardt kicks in. As you navigate the real-world model with the camera, the camera feed will be overlaid with the difference between expected marker corners and seen seen marker corners based on the dominant marker in the current view. This is to aid the user in navigating the model, markers exhibiting a large difference should be prioritised for viewing with the camera. Press F2 again to stop the optimisation, it is usually clear when the estimated model matches the real-world model as the expected corners should match up more or less with the actual corners in the camera feed overlay.

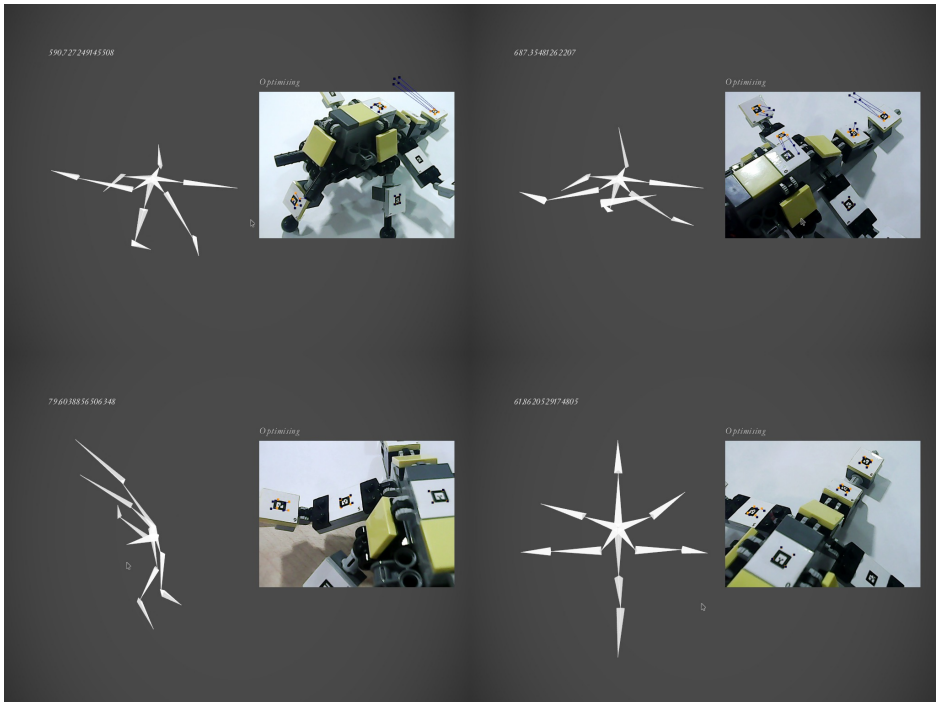


Fig. 51: Optimisation in action

Figure 51 shows optimisation in action. The top two images show the optimisation in the beginning stages, notice how the 2D point difference is large on the tail and wings as the optimisation is in the process of estimating those parts of the skeleton model. The bottom two images shows the optimisation towards the later stages, the model in the 3D model viewer is nearing the rotations present in the real-world model. The 2D corner seen and observed has almost completely matched up in the overlay.

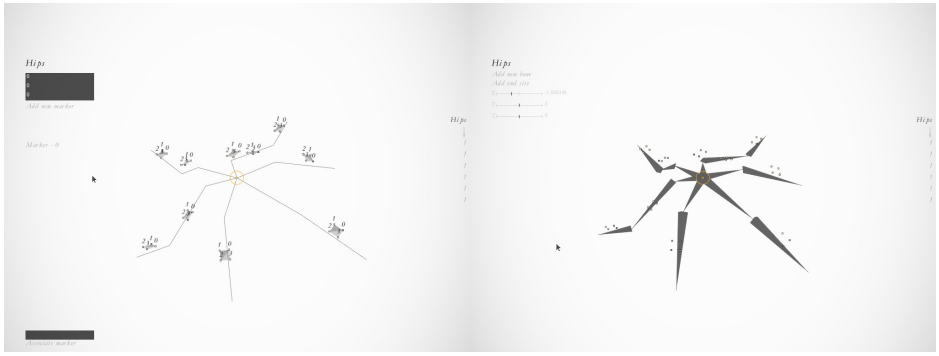


Fig. 52: **F**inal rotation model

Figure 52 shows the final estimated skeleton model pose after optimisation. The results obtained are very accurate, notice how it is almost completely symmetrical as it is in the real-world model. Upon selecting the bones and comparing it to the real-world model rotations, we see a similar sort of accuracy to the accuracy obtained during the quantitative analysis testing. The user can now save this rotated model.

Finally there is a tool included which allows the user to export this pose as a BVH file with a single frame of animation. This allows the user to import the result into a 3D modeling application such as 3DS max and attach a real 3D model to the estimated pose for further use. The user simply has to click the export to BVH button in the bottom right of the UI after optimisation has finished. The BVH file will be written out to the same folder that the associated model is saved to.

Conclusion 5

Throughout this report we have demonstrated an easy to use system that allows a user to generate a very accurate pose of a real-world object using just a web-cam in real-time. This has been achieved using computer vision techniques, in particular reconstruction from many views using a Levenberg-Marquardt optimisation scheme. In this section I will highlight a few important things to be taken from this project, detail some of the limitations of the system and finally talk about some future work that can be done in order to improve and add functionality to the system.

5.1. Summary

- Using bundle adjustment techniques can yield a very high degree of accuracy, typically bone positions were less than a few mm off, and in this case can also help recover from situations where not all the feature points may be visible in the scene. For instance in the situation of having a missing marker on a bone chain between two seen markers, we can typically recover the rotation of the bone with the missing marker based on the information about the markers either side of it. An example of this is shown in Figure 53.
- Initially we were having problems with some markers being mistaken for others when the camera was not in focus or the lighting conditions were not ideal. This can cause problems as the view collected would be bad and adversely affect the optimisation. We got around this by saying that a view is not valid if the marker is detected further away than the skeleton model would allow from any other marker. This check is an O(n^2) operation for markers in the scene, in practice this is acceptable as even if we had the full 1024 markers in the scene the time to perform this check would be minimal.
- Bundle adjustment is an O(n^3) problem by nature, using all the information from all the views simultaneously would mean it would take far to long for real-time application. We solved this problem by splitting up the tracking the markers from the optimisation of the estimated rotations. This is similar to the approach taken by PTAM. Furthermore we intelligently selected a limited amount of views we were using in order to make each iteration of the optimisation complete in a far faster time. This reduced the problem down to a O(nm) problem with increasing view count.

- We also found that doing a localised optimisation on certain parts of the model can helps in speeding up the overall optimisation. Reducing the number of bones in an optimisation pass reduces the computation time exponentially.
- Initially we were running into the problem of local minimums causing the optimisation to terminate on an incorrect result. The use of the rotation estimate heuristic allows us to get around this problem as well as speeding up the process of optimisation as we were starting closer to the final solution.
- Having the overlay on the web-cam feed meant that the user can have an easy way to see how well the optimisation is going. It proved to be very useful in quickly determining if the optimisation was working or not.

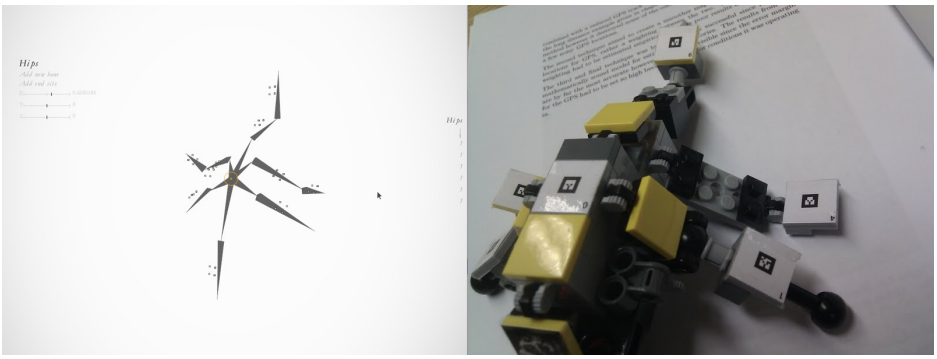


Fig. 53: **M**issing marker on wing, notice how the final estimate recovers the pose of the complete wing

5.2. Limitations and future work

One of the key limitations of the system is the need to accurately measure the model to input into the application. Having a badly measured model can yield bad results as the optimisation would think the rotations are off if it expects the markers to be somewhere else. A further improvement to get around this would be to have a self-calibrating system which can determine the lengths of the bones and possible rotations just be looking at the model with the web-cam. A possible way to do this would be to manually take a number of views with the bones rotated in a certain way around the real-world model in order to find out where the joints are on the model.

The application does not get around the problem that the optimisation would still increase exponentially with the number of bones in the model. Although it would probably be unwise to use a different scheme other than Levenberg-Marquardt optimisation, we could change the strategy related to how many views we use per run and the chances

of doing a local optimisation as opposed to a global one. This strategy could be based on an analysis performed on the model done before the optimisation begins. Further work is also possible on how we rank the views that we obtain, for instance we can rank views exhibiting large amounts of motion blur to be of worse quality and base our selection on more factors than just the number of markers in a scene and the most dominant marker seen.

Having markers on the real-world model is still not the most ideal solution. A big improvement could be made if we could make the system marker-less as well, it would rely on real-world features on the model instead of the fiduciary markers. We would need a good way to be able to tell the system what to look for in the scene and how to comprehend the features it finds in relation to the skeleton model. This would likely be the next step forward for the application.

Another are of future work would be to make the application be able to detect when the real-life model has changed rotation in the middle of an optimisation pass. We could then be able to perform real-time capture of the change of rotation. This can be further worked upon to make it retain information about the bones that have not changed rotation in order to make it determine the new pose without having to recapture information about the whole skeleton model.

It would be useful if we could visualise the full 3D model that will be used in programs such as 3DS Max in conjunction with the estimated pose. We could include this in the 3D viewer by allowing the user to import a valid rigged 3D model.

Finally it would be an interesting area to explore if we could model 'bones' that have changing lengths such as springs or fabric. Our optimisation would need to take the change in size of a bone into account as well as the rotation. This would allow us to move onto modeling not only rigid skeleton objects but objects with elastic properties such as clothing and ropes.

Appendix A

A.1. Hardware

The project was developed and run on Windows 7 running on a Dual-core i7 (2.8ghz) and 8gb of RAM. The camera used throughout the project was a non-branded 640x480 30hz web-cam. The camera intrinsic and extrinsic parameters are listed below:

```
Focal length : [ 898.003  896.062 ]
Principal point : [ 360.282  204.918 ]
Distortion : [ -0.081541  1.034180  0.006233  -0.004752 ]
```

A.2. Software

The entire application was developed using the C# programming language in Visual Studio 2010 Express edition. We used Microsoft XNA for the graphics API, Alglib for the Levenberg Marquardt implementation and AVLAR for the marker detection. We are using the AVLAR wrapper built from Goblin XNA, this must be included as a DLL in the project.

Bibliography B

[1] Bochkanov Sergey Anatolyevich. Alglib http://www.alglib.net.

[2] R Hartley B Triggs, P McLauchlan. Bundle adjustment a modern synthesis. page 153, 2000.

[3] D. C. Brown. Decentering distortion of lenses. *Photometric Engineering*, 32(3):444--462, 1966.

[4] MICROSOFT CORP. Visual target tracking, 2010.

[5] M. Fiala. Artag, a fiducial marker system using digital techniques. In *Proc. IEEE Computer Society Conf. Computer Vision and Pattern Recognition CVPR 2005*, volume 2, pages 590--596, 2005.

[6] VTT Finland. Alvar http://virtual.vtt.fi/virtual/proj2/multimedia/alvar.html.

[7] Maureen Furniss. Motion capture. 1999.

[8] KATO H. Artoolkit. *http://www.hitl.washington.edu/artoolkit/*.

[9] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[10] Mat Cook Toby Sharp Mark Finocchio Richard Moore Alex Kipman Andrew Blake Jamie Shotton, Andrew Fitzgibbon. Real-time human pose recognition in parts from single depth images. 2011.

[11] S. Carlsson J.C. Clarke and A. Zisserman. Detecting and tracking linear features efficiently. 1996.

[12] G. Klein and D. Murray. Parallel tracking and mapping for small ar workspaces. In *Proc. 6th IEEE and ACM Int. Symp. Mixed and Augmented Reality ISMAR 2007*, pages 225--234, 2007.

[13] UW Madison. Biovision bvh http://research.cs.wisc.edu/graphics/courses/cs-838-1999/jeff/bvh.html.

[14] Donald W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):pp. 431--441, 1963.

[15] Nikon. Understanding focal length http://www.nikonusa.com/learn-and-explore/photography-techniques/g3cu6o2o/1/understanding-focal-length.html.

[16] D. Oberkampf, D. F. DeMenthon, and L. S. Davis. Iterative pose estimation using coplanar points. In *Proc. CVPR '93. IEEE Computer Society Conf Computer Vision and Pattern Recognition*, pages 626--627, 1993.

[17] Bill Triggs, Philip F. McLauchlan, Richard I. Hartley, and Andrew W. Fitzgibbon. Bundle adjustment - a modern synthesis. In *Proceedings of the International Workshop on Vision Algorithms: Theory and Practice*, ICCV '99, pages 298--372, London, UK, UK, 2000. Springer-Verlag.

[18] Z. Zhang. A flexible new technique for camera calibration. 22(11):1330--1334, 2000.