Learning Action Primitives From 3D Stereo Vision Measurements

A thesis submitted to the University of Aalborg
for the degree of Master of Engineering in
Computer Vision and Graphics

Ricardo A. Arango S.

Supervisor: Prof. Dr. Ing. Volker Krüger

Aalborg University
Copenhagen Institute of Technology
Department of Media Technology and Engineering Science

January 15, 2010

Abstract

Models defining the motion of objects in images are an important field in Computer Vision. A common drawback of many models it that sets of trajectories with different motions but sharing some common paths are described individually, therefore producing redundant and uncorrelated data. These common paths can be described as action primitives, and models linking action primitives are necessary to correlate them. This thesis defines a framework to track objects visually using color segmentation and Hidden Markov Models, record motion trajectories, identify action primitives and build a single model from different trajectories describing them jointly and efficiently. The action primitives model can be used as a learning model for a robot with a higher level definition of the actions performed. The framework is written as a C++ programming library and a complete implementation is provided which fulfills all the requirements for object detection, tracking, motion recording and model building.

KEYWORDS: Imitation Learning, Action Primitives, Hidden Markov Models, Motion Trajectories, Stereo Tracking, Color Segmentation, Blob Filtering.

Acknowledgements

I would like to express my thanks to:

- Professor and supervisor Volker Krüger for his continuous support with the project and making sure I had the right tools and assistance to deliver it.
- Sanmohan whose work is the foundation for this thesis. Also for his patience and motivation to work with me on the project. I am sure that without his help I would not have been able to complete it.
- Dennis Herzog I thank for sharing with me his knowledge on Hidden Markov Models.
- Daniel Grest, for giving me support to the questions related to my project.
- To all the teachers in the Computer Vision and Graphics course for all the things I have learned during these two years at AAU.
- Finally I want to thank my family for their unmeasurable support all this time, and specially my girlfriend Natalia for her patient understanding.

Accompanying CD-ROM

The thesis document, programmed library and editor program presented can be found on a supplementary CD-ROM. The library is programmed in C++ and has been compiled successfully in Visual Studio 2008. The editor has been run and tested in Windows Vista and Windows 7.

Contents

List of Figures

Chapter 1

Introduction

THE aim of this thesis is to implement a system capable of creating models of human actions as applied on objects using action primitives as proposed [**?**]. With a stereo camera setup the system is be able to track an object in three-dimensional space as it is moved and record it's motion trajectory. A group of motion trajectories are used to build the action primitives model as an directed graph. A motion trajectory can also be used to identify the action primitives and the entire action performed if it is defined in the model, or discard it as an unknown action.

The project has three areas of interest. The first one is focused in detecting and tracking moving objects and recording their trajectories. The second area is interested in training a single hidden Markov model (HMM) that represents a set of many different trajectories. The third and last area looks at infering the motions as sequences of action primitives, atomic symbols of activity, from the HMM. Using these symbols and the HMM a directed graph is built that defines the grammar of activities as a sequence of actions. This graph together with the action primitives and the HMM compose the complete model which can identify human actions and provide the sequence of sub actions as action primitives performed.

1.1 Motivation and Objectives

Teaching a robot how to mimic an action on an object as done by a human is a complex task. Take for example a person showing a robot the movements required to serve a cup of tea. Assuming all the items needed are at hand's reach, the process can be separated in this steps: 1) Pour tea, 2) Pour milk 3) Add sugar. As simple as it looks, each one of these steps is composed of many layers

of abstraction. At the lowest level are the minute details of the position and rotation of each body part. Then a set of middle layers of hold the information of the sub-actions defined by those movements, such as picking the object, moving it towards the cup and inclining it so the liquid drops in the cup. Finally the top layers identify the action as a whole with possibly the use of the context in which the action is performed ie. serving tea over a planar surface. All that information must be dealt with in order for a robot or computer to fully understand and learn what a person is doing. Therefore, solving or partially solving some of those layers greatly aids in the solution of the entire problem.

One of the simplest way to model such actions is using stochastic models of the low level details of the motion, which by definition, allows for some of the flexibility inherent to the nature of human beings as they move. The most common example of an stochastic model is Hidden Markov Models (HMM). These models usually describe the position and orientation of each tracked object and allows for generation of new motions and identification of the motions. One drawback of HMM is that there is no in-between knowledge, they jump from the lowest to the highest level of abstraction of the action.

A more interesting approach is to represent the actions as a series of substeps. As in the previous example, in terms of the action performed, pouring tea and adding sugar have similar motions. Therefore, it is a good idea to find the common sub-actions which are performed, to elaborate a more efficient and higher level model where every subaction can be treated as a symbol in an alphabet of motions and the grammar defines the rules for performing an action. These sub-actions or symbols can be defined as *action primitives*.

This proposed technique has been treated by Sanmohan [**?**] and the objective of this thesis is to extend his work with the implementation of a real-time system capable of learning and recognizing action primitives. In detail, the objectives include:

- Detecting and tracking objects in real-time while recording their features in 3d space.
- Defining an stochastic model of a set of many actions from several samples.
- Identifying the subactions in the model as action primitives.
- Obtaining the grammar of the model for validation and identification of new trajectories.

An additional objective of this thesis implicit to software development is to create modular components that can be reused by other people in future projects.

1.2 Thesis Organisation

The structure of the thesis is as follows:

Chapter 2 provides a description of the background to both the tracking and modeling of motion trajectories.

Chapter 3 outlines the requirements and defines the proposed solution to the problem. It also describes the work on action primtives upon which this thesis is based.

Chapter 4 presents the implementation of the proposed solution.

Chapter 5 describes the tests performed on the system and the results obtained.

Chapter 6 draws conclusions and contains a summary of the contributions of the work.

Chapter 2

Related Work

THIS section gives a theoretical background of the topics that relate to this thesis. It describes some of the techniques used in object tracking and action primitives using computer vision.

2.1 Object Tracking

Tracking objects is an important and current field of research and applications in the area of computer vision. It has many interesting and useful applications such as motion capture and recognition [**?**, **?**], surveillance [**?**], human-computer interaction [**?**], navigation [**?**], and augmented reality [**?**, **?**]. With the ever increasing power of computers and video cameras, and the decreasing prices it has become of more interest and feasible application the use of video cameras to perform object tracking.

There are two major components in a common visual tracker: 1) Target Representation and Localization, concerned with creating an abstract description of the target and detecting it in a frame generally performed as a bottom-up process and 2) Filtering and Data Association which is a top-down process dealing with the dynamics of the tracked object, learning of scene priors, and evaluation of different hypotheses [**?**]. An overall review in object tracking methods can be found in [**?**] and in which this section is based on.

In a general sense tracking objects using vision is the process of locating an object in each frame during a sequence of video frames. It means that in every new frame the tracker is identifying in the image plane a previously known object, and optionally depending on the context obtaining other relevant information including size, position, velocity, area, shape and orientation. Unfortunately

tracking objects in images is not free from difficulties such as: loss of information, occlusion, noise, changing shape of non-rigid objects, changes in illumination, complex motions of objects and real time computing requirements.

Given these limitations it is possible to apply some restrictions to the tracking enviroment to solve or avoid some of them. For example, the tracking can be performed inside a room with controlled artificial ilumination allowing for the reduction of some of the noise and illumination variances. Another example would be to have apriori knowledge of the motion of the objects being tracked, which can be used to assume a constant velocity, ignore acceleration or define the motion model more accurately. Other parameters such as number, size, shape, color or basically any other freatures of the objects previosly known can greatly simplify and/or make more accurate the results and choice of tracking methods.

2.1.1 Representing Objects

In order to reduce the amount of calculations and also for visualizing and easing the understanding of the properties extracted from the object tracked, it is common to define it into a simpler, cleaner representation. Some commonly used representations are: points, geometric shapes, sillouete or contour [**?**], articulated models [**?**] and skeletal models [**?**]. Points Mostly used for objects that occupy a small part in the images, or that can be abstracted to a point representation.

2.1.2 Features for Tracking

Extraction of good representative features and properties from the objects is crucial to identify them in a feature space. Whatever the object to be tracked may be, it must have some visually defining characteristics that allow a separation of the object from the background and from other objects. Those characteristics are not fixed but instead depend on the context of the problem. When looking for features it is also important to optimize the data by eliminating redundancy and increase the uniqueness of the feature parameter for each specific object. Feature selection is sometimes dependant on the object representation chosen, for example, edges are commonly used when the representation is based on contours. Some of the features that are commonly used for tracking are: color [**?**], edges [**?**], optical flow [**?**] and texture [**?**].

2.1.3 Object Detection

When tracking objects it is necessary to have an object detection algorithm which can trigger and/or continue an already started tracking process. A common approach is to identify a new object to be tracked in one frame, extract the desired

features from it, and use them to follow it in subsequent frames. Other approaches update the initial features of the object during the tracking process and/or use temporal information to increase the likelihood of tracking the right candidate. Some interesting object detection techniques are: point detectors [**?**][**?**][**?**][**?**], background subtraction [**?**][**?**][**?**][**?**][**?**][**?**], segmentation [**?**][**?**][**?**] and supervised learning [**?**][**?**][**?**][**?**].

2.1.4 Tracking Objects

The objective of tracking is to produce the trajectory of the object as it moves in between frames. With aid of the detection algorithms it can also provide the region where the object is located in the image. To obtain the trajectory the tracker must identify a corresponding observation for an object for a new frame. Depending on the features and representation chosen the tracking may incorporate part of those as parameters for the correspondance and tracking. Some of the common tracking algorithms are: Point tracking such as Kalman FIlter [**?**] and Particle Filters [**?**] and Kernel trackers such as Mean-Shift [**?**] and KLT [**?**].

2.2 Modeling Trajectories

Modeling of human actions is a well studied subject. Of particular interest if to identify the common segments of motion in a series of samples. An atomic action, also referred here as action primitive, is a representation of a common set of observations that sum an entinre human action. It defines the state of the object tracked in a segment of the action. A robust model which uses atomic actions to model human actions is Hidden Markov Models.

2.2.1 Hidden Markov Models

A hidden Markov model (HMM) is a probabilistic model of a pattern generally built from many sampled observations. It allows to predict the hidden or internal state of a system using only the observations, by defining the likelihood of transitioning from state to the another and of an observation given a particular state. HMMs are of interest because they have been througly detailed an tested. In the field of human actions, HMMs have been sucessfully used in [**?**][**?**][**?**].

Chapter 3

Analysis

In this chapter the details of the different theory and algorithms researched and implemented for this thesis are presented.

3.1 Image Acquisition

The images for the system will be acquired for real time processing using two color cameras. The cameras will be located in a way that they both share a common viewing cone, in which the objects to be tracked will move. Two images showing the view from each camera in the stereo setup used during the tests project can be seeen in the Figure 3.1.

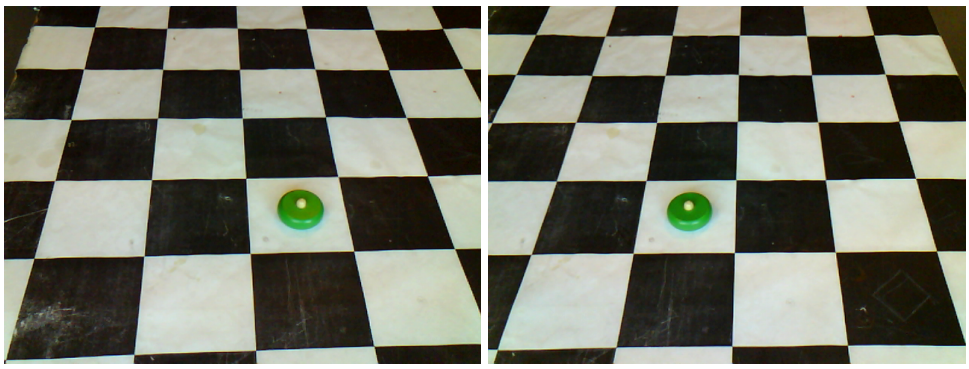


Figure 3.1: Images from two cameras in a stereo setup.

3.1.1 Background Substraction

The background subtraction module is in charge of defining and acquiring the parameters of a model for the background, and of classifying a new pixel as

either belonging or not belonging to the model. To increase the detection of objects a background substraction filter using the Running Average Gaussian was chosen.

3.1.2 Running Gaussian Average

As stated in the object detection section 2.1.3 there are many different approaches which can been used to perform background subtraction. One of the requirements of the system is that it should work in real-time and this limits the computational complexity of the background subtraction method. A good candidate in terms of performance is the Runnning Gaussian Average algorithm. Eventhough it has a unimodal distribution of the background it is sufficient to represent it in the testing enviroment used. In a more complex scenario a multi-modal or more sofisticated background subtraction technique could give better results.

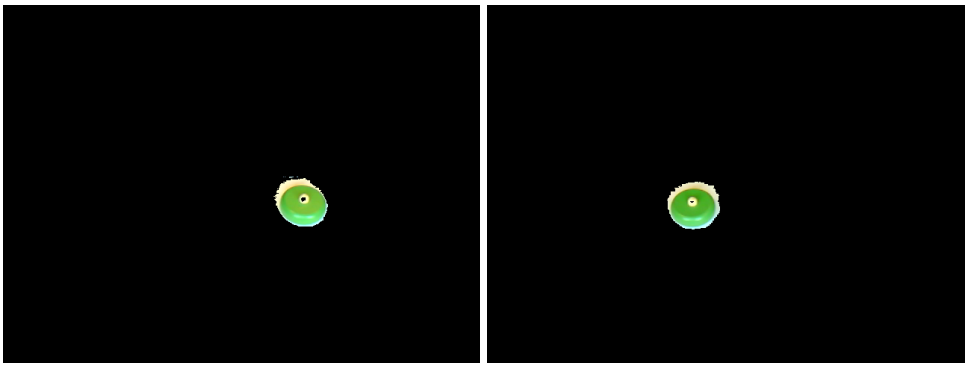


Figure 3.2: Outcome resulting from applying the running Gaussian average background subtraction technique.

As the name implies the Runnning Gaussian average uses a univariate Gaussian distribution to model the background. Each pixel in the image has a distribution with mean μ_t and variance σ^2 at at time t . The interesting factor is that the probability density function can be updated incrementally as new frames arrive, using a weighting value α for the new measurement using

$$\mu_{t+1} = \alpha F_t + (1 - \alpha)\mu_t \tag{3.1}$$

and

$$\sigma^2_{t+1} = \alpha(F_t - \mu_t)^2 + (1 - \alpha)\sigma^2_t \tag{3.2}$$

where F_t is the color of the new pixel at time t. If selectivity is used, the model is only updated when the new pixel is not in the foreground. To classify a pixel as being in the foreground or background, a simple gate

$$|F_t - \mu_t| > k\sigma^2 \tag{3.3}$$

can be used, where the threshold is defined by the covariance σ^2 and a scaling parameter k.

3.2 Color Filtering

The first step in the tracking process is that given an image each pixel has to be classified as either belonging or not belonging to the object. Using RGB colors directly presents the problem that changes in illumination can seriously affect this classification. To overcome this, color spaces other than RGB can be used which separate the chromacity information from the luminosity information of the pixels. The color spaces used in here with these characteristics are Normalized RGB, HSV, HSL, YCbCr and Hunter LAB. Each one of these color spaces has three components, one of them being the luminosity parameter except for Normalized RGB in which one of the values is redundant as $r + g + b = 1$. As the luminosity and redudant parameter are not needed they can be discarded and the color filtering is obtained using the other two remaining components.

3.2.1 Color Spaces

Here is a description of the color spaces implemented in the project. For all the color spaces only two channels defining the chromacity are stored. The other channel which has the illumination, or in the case of Normalized RGB with is redudant, are discarded. A more in-depth description of colour spaces can be found in [**?**].

RGB

This is the common Red, Green, Blue representation of additive color used traditionally in color display systems. Usually display devices use three individual light sources for each one of the RGB colors to represent a weighted combination of the three components. Each color represents an axis in a color cube, where the corner of the cube at the origin of the color space represents black, and the

corner at the opposite corner represents white. All colors are found inside the volume of this cube.

In computer applications the RGB color space is usually discretized to 8 bits, so that each channel has a maximum of 256 possible values numbered from 0 to 256. This gives a range of $256^3 = 16'777.216$ possible values for a color represented as an (R,G,B) tuple. In our case RGB is the color space in which the images are provided by the cameras.

It is mentioned here for clarity, but the color description of the objects tracked only used the color spaces defined sa follows.

Normalized RGB

The first color space considered that separates illumination from chromacity is the normalized RGB space. In this space the values of a color remain the same even when the amount of illumination changes. A color in RGB can be transformed to the Normalized RGB space using the following formulas:

$$r = \frac{R}{R+G+B} \tag{3.4}$$

$$g = \frac{G}{R+G+B} \tag{3.5}$$

$$b = \frac{B}{R+G+B}. \tag{3.6}$$

It can be easily seen that $r + g + b = 1$, therefore any of the three values can be discarded as it is redundant i.e. $g = 1 - r - b$.

YCbCr

Next to RGB, YCbCr is one of the most common color spaces used in digital component video. It separates the lightness in the Y channel, and stores the color information in the Cr, Cb channels. The formula to obtain a YCbCr representation of an 8-bit per channel RGB color is:

$$Y = 16 + \frac{1}{256} * (65.738 * R + 129.057 * G + 25.064 * B) \tag{3.7}$$

$$Cb = 128 + \frac{1}{256} * (-37.945 * R - 74.494 * G + 112.439 * B) \tag{3.8}$$

$$Cr = 128 + \frac{1}{256} * (112.439 * R - 94.154 * G - 18.285 * B). \tag{3.9}$$

HSV and HSL

These two color spaces are a two representation of colors in the RGB color model that also separate the ilumination from the color information. HSV stands for Hue, Saturation and Value. HSL stands for Hue, Saturation and Lightness. Both are similar in what they describe and how they are defined. In both cases it is interesting for this thesis that Lightness and Value represent ilumination. The conversion from RGB to HSV and HSL can be seen in appendices A.3 and A.4 respectively.

Hunter LAB

The LAB color space is named after it's three components: L-luminosity, a-channel and b-channel. As in the previous color spaces and of interest here is that the luminosity is defined separate from the color and tone. The are many different standards for the LAB color space. The one chosen here is Hunter-LAB, which has a faster color conversion. The conversion from RGB to LAB is in appendix A.5.

3.2.2 Pixel Classification

To classify a pixel as belonging or not belonging to a certain color description, different parametric and nonparametric likelihood functions were considered: Thresholds, Mahalanobis distance, Back projection and Mixture of Gaussians. All of these methods required as a first step to sample some pixels from the object in the image after which the classifiers are initialized and ready. After a pixel is classified it is stored in a grayscale image as either black (not object) or white (object).

Thresholds

Fixed value thresholds that define a volume of colors in which the object's pixels are contained. Specifically the following conditions must be true for a given pixel using the two chosen channels for a given color space to be classified as belonging:

$$c_0 \geqslant c_{0min} \; and \; c_0 \leqslant c_{0max} \tag{3.10}$$

$$c_1 \geqslant c_{1min} \; and \; c_1 \leqslant c_{1max}. \tag{3.11}$$

The advantages of this system is that it is fast and the thresholds can be easily modified.

Histograms

Using a normalized histogram created by sampling pixels of the object, the pixels being classified are replaced by the value of the closest normalized bin in the histogram for each pixel color. This value represents the probability of finding that color in the histogram of the model, and can be thresholded to obtain the final classified image using

$$p_x(i) = p(x = i) = \frac{n_i}{n} \tag{3.12}$$

where i is the intensity of the pixel channel, n is the number of samples and $n_i$is the number of samples found in the *i-th* bin.

Mahalanobis

Another alternative is to use the Mahalanobis distance between the color and the previously known distribution, with a mean μ and covarianceΣ. This distance measurement takes into account the way the colors are distributed in the samples, for example, if the distribution varies a lot in one channel but not as much in the other, a small divergence from the mean in the first component is not as important as divergence in the second one. After defining a theshold with a maximum distance, a pixel can be determined as belonging or not belonging to the distribution.

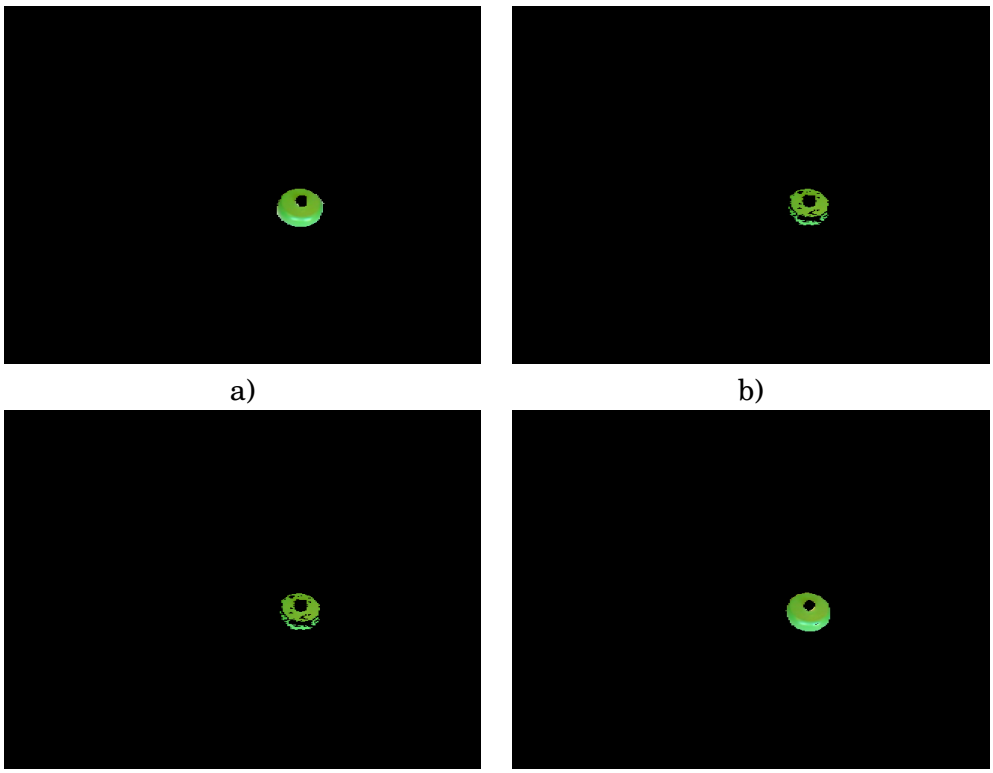
$$D_M(x) = \sqrt{(x - \mu)^T (\Sigma)^{-1} (x - \mu)} \tag{3.13}$$

Mixture of Gaussians

The fourth method chosen models the color distribution in the object as a Mixture of Gaussians, each gaussian defined by it's mean μ, covarianceΣ and a weighting coefficient c_i. The likelihood of finding a color in the mixture of gaussians is defined by

$$f(x) = \sum_{i=1}^{N} c_i \phi(x|\mu_i, \Sigma_i) \tag{3.14}$$

where c_i is the coefficient or weight and $\Phi(x|\mu_i, \Sigma_i)$ is the Gaussian color distribution for the *i-th* mixture. To obtain the parameters of each mixture using a set of sample colors, K-Means is used as the first clustering step and later the Expectation-Maximization algorithm is applied [**?**].



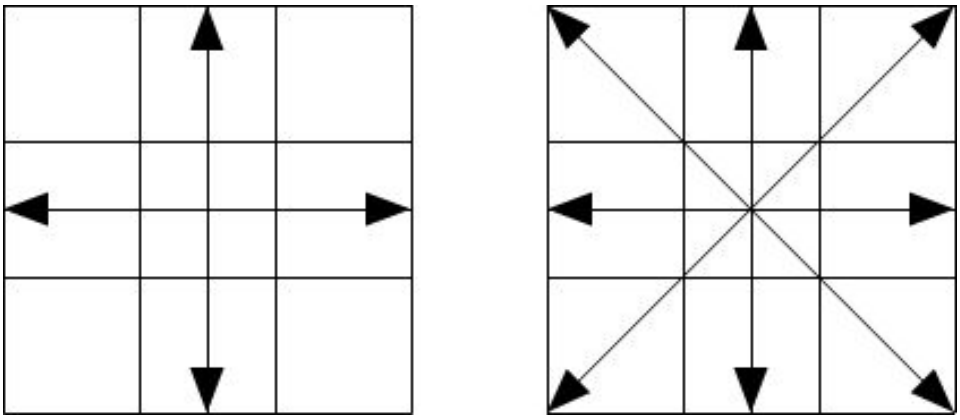
a) b)

c) d)

Figure 3.3: Results of applying the four different classification techniques: a) Thresholds, b) Histograms, c) Gaussian Mixture with three mixtures and d) Mahalanobis distance.

3.3 Blob Filtering

After the background is subtracted and a pixel filtering technique from the ones mentioned above is applied, the result obtained is a mask where white pixels represent pixels that potentially belong to an object. This pixels should be grouped together to form an abstract representation of the object as an ellipsoidal blob, defined here as blob filtering.



4-connectivity 8-connectivity

Figure 3.4: Two rules for connectivity of pixels.

The first step in the blob filtering algorithm is to group close pixels as objects. A connected components algorithm with 4-connectivity was used for this, where the top, left, right and bottom pixels of each pixel are checked for connectivity. Other possibility is to use 8-connectivity, but the first choice is better at isolating objects which is the intention. A lot of work has previously been done for blob labeling [**?**, **?**], but the connectivity algorithm here developed is built from scratch for the specific purposes of this thesis.

3.3.1 Connectivity Algorithm

The algorithm walks through every pixel in the image, from left to right, top to bottom, and uses the binary image as the input data, an image to store the object id to which each pixel belongs and a queue to recursively "fill" each blob with an id. For every pixel position x,y, the algorithm pushes the coordinate pair x,y to the queue. It iteratively pops values from the queue and checks the corresponding value in the at the x,y coordinate in the binary image to compare if it is black or white, therefore acting as a mask. If it is black it is ignored. If it is a white pixel it will read the value of the pixel in the same pixel position in the image with the blob ids, initially all set to unassigned. If it is unassigned, a new blob is created and the new id assigned to that blob and the pixel in the

blob's ids image. All neighboring pixels are also assigned the same id if they are different from zero in the input binary image and pushed to the queue if the id they had before was different from the new one. This operation will "fill" the region with the id of the object. The process is repeated until the queue is empty, and the loop advances to the next pixel position. The pseudo-code of the algorithm is shown in Algorithm 3.1.

Algorithm 3.1 Connected Components algorithm with 4-connectivity

```
blobId = 1
foreach pixel p in image
   queue.push_pixel(p)
   while(queue is not empty)
      p2 = queue.front()
      if(p2 != 0)
         if(imgBlobs.pixel(p2.pos) = 0)
            imgBlobs.pixel(p2.pos) = blobId
            blobId++
            foreach neighbor of p2
               if(neighbor != 0 && imgBlobs.pixel(neighbor.pos) != blobId)
                  imgBlobs.pixel(neighbor.pos) = blobId
                  queue.push(neighbor)
                  break
               else
                  imgBlobs.pixel(p2.pos) = 0
```

3.3.2 Mask Filling

One common ocurrence is that some of the pixels inside the contour of the mask obtained with pixel color classification that defines an object could different from white, or in other words, be non-object pixels. The problem this brings is that when the parameters of the object such as the area and position are estimated, they are affected by these 'holes' in the object. For example, consider a ball being tracked in a sequence of images. The inner pixels of the ball could be classified as non-object pixels because of bad sampling or lighting changes. The edges of the object nonetheless are correctly identified as object pixels. If the area is estimated, it would give a value very much inferior to the area the objecct acutally occupies. Consider again the same ball partially illuminated. The center of the object should be the same, but because of the shape of the mask it is moved to a different position.

A solution to both of these issues is to find all the pixels that are located inside the region defined by the borders of the object and set them as white, therefore as part of the mask. In this project a simple technique was developed that iterates over the already identified object mask, and fills gaps. The pseudo-code for the algorithm is in Algorithm 3.2. A second approach uses the already implemented convex hull algorithm from OpenCV [**?**] to fill the gaps with similar results.

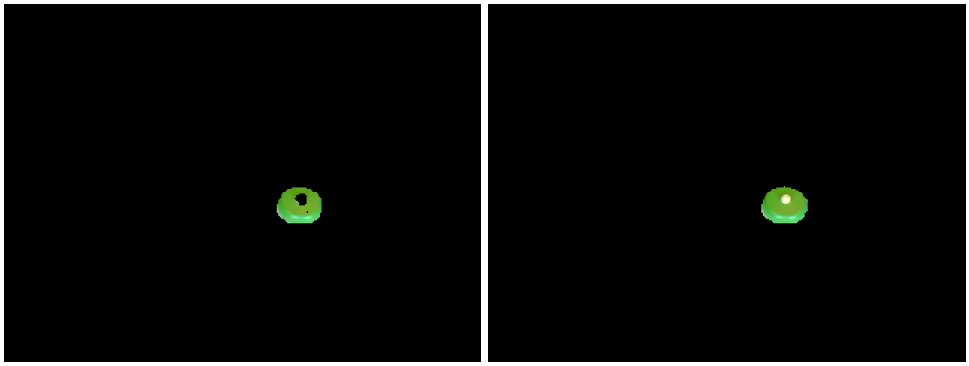


Figure 3.5: Example of Blob mask filled to remove holes.

Algorithm 3.2 Convex hull algorithm used to fill gaps in blob mask

```
for each row in bounding_box
   left = findLeftMostPixel()
   right = findRightMostPixel()
   for each pixel in row[left:right]
       pixel = blobId
for each column in bounding_box
   top = findTopMostPixel()
   bottom = findBottomMostPixel()
   for each pixel in column[bottom:top]
       pixel = blobId
```

3.3.3 Blob Features

Once the blobs have been identified in the image using the connectivity algorithm 3.2, it is possible to obtain information from each one of them that can be at later stages. Some useful and important features for typical tracking are the position and the area of the object. The position can be obtained using the average position of all the pixel that make the object, and the area by summing all of the pixel. These two values can be calculated using the image moments.

For a continuous function in the 2d space, the moment is defined as

$$M_{pq} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} x^p y^q f(x,y) dx dy, \quad p,q = 0,1,2... \tag{3.15}$$

Adapting this to the scalar space the moments are calculated with by

$$M_{pq} = \sum_{x} \sum_{y} x^p y^q f(x,y) dx dy, \quad p,q = 0,1,2... \tag{3.16}$$

Let $I(x, y)$ denote a binary variable mapping the values of an image, where zero denotes an empty pixel, and one denotes a pixel occupied by a blob. The zero order moment or area is

$$Area = M_{00} = \sum_x \sum_y I(x, y) \tag{3.17}$$

where x, y define the region in the image that is used. It is then necesary to know before hand the bounding box of the object. As the bounding box of the object is defined by a square region it might contain pixels from other blobs. Therefore it is necesary to check when the pixels are being counted that they belong to the blob for which the area is being measured.

The centroid or moment of interia of the blob is obtained also with the moment. The reason for using the centroid, is that it takes into account the distribution of the pixels. Therefore outliers will have less effect in the estimation of the center. The centroid $(\overline{x}, \overline{y})$is obtained with

$$\overline{x} = \frac{M_{10}}{M_{00}} = \sum_x \sum_y xI(x, y) \tag{3.18}$$

$$\overline{y} = \frac{M_{01}}{M_{00}} = \sum_x \sum_y yI(x, y) \tag{3.19}$$

which requires the the area M_{00} of the object to be calculated beforehand.

Other parameters that can be used are the timestamp of the blob state since it has been tracked and the width and height which are implicitely defined in the bounding box.

3.4 Tracking

For the tracking step the Kalman filter was chosen. The main reason for this selection was that the Kalman filter has been sucessfully been used in many tracking problems [**?**]. Also, the state vector of each object has parameters that can be easily modeled using the Kalman filter. Finally, the Kalman filter is flexible enough to allow different types of parameters to be tracked.

3.4.1 Kalman Filter

Even though a lot of information can be measured from the objects in the images, there are two problems that come from those measurements. The first one is noise. Whether it is from the physical camera or from changes in illumination the images contain noise that affect the values measured from each object. The second problem is that some parameters such as velocity and acceleration, are difficult to be accurately measured directly. The Kalman Filter can help reduce the effect of both.

In the most general sense, the Kalman Filter is a predictor-corrector algorithm. Given the current state of the system and a transition model it predicts the state of the system in the next iteration or time-step. Then, after receiving a new measurement, it corrects the state prediction to get closer to the real state. A very good introduction to the filter can be found in [**?**].

The Kalman Filter tries to estimate the current state of a system defined by a linear stochastic difference equation

$$x_k = Ax_{k-1} + Bu_k + w_{k-1} \tag{3.20}$$

with the measurement vector define as

$$z_k = Hx_k + v_k \tag{3.21}$$

where x_k is the system or process state and $z_k$is the state measurement.

Equation (3.20) defines the process state x_k as the sum of the process state at the previous time-step x_{k-1} multiplied by the transition matrix A which describes how the previous state changes in time, the control input u_k times the control matrix B which models external inputs to the system, and a white noise variable w_{k-1} describing the possible random noise of the system state. The measurement z_k is defined as the transition H of the state to produce a new measurement and white noise v_k.

Both the process and measurement noise are assumed to be independent, white and with Normal probability distributions. In the specific case of blobs in a sequence of images without knowledge of the force that makes them move, the optional control input u_k can be ignored. The complete set of equations used for the Kalman Filter prediction and correction can be found in the Appendix B.

3.4.2 Object State Update

What's really important to be noticed is the difference made about the object or system state, and the measurement state. Using a Kalman filter it is possible to obtain the state of internal variables of a system by using only the external measurements if the model of the system is well defined. This model, stored in the matrix A, explicitely defines the transition of each component of the state vector, using the values of the system at the previous time-step.

In the simplest case, where the parameters in the state vector are direct measurements, such as the size of the object, and are considered to remain more or less the same, an indentity matrix I is sufficient to represent the transition matrix. In the case of linear motion ignoring acceleration, the transition matrix should describe the relation between position and velocity using the standard constant velocity formula. As that the position is dependent on the time the object has been moving in a frame step, the transition has to include this variable parameter. Given that the velocity is not directly measured, it is not defined in the measurement vector. The equations for the state vector x_k, the state measuement z_k, the state transition A and the measurement transition H are shown below of the first order motion escenario are shown below.

$$x_k = \begin{pmatrix} x & y & v_x & v_y \end{pmatrix}^T \tag{3.22}$$

$$z_k = \begin{pmatrix} x & y \end{pmatrix}^T \tag{3.23}$$

$$A = \begin{bmatrix} 1 & 0 & t & 0 \\ 0 & 1 & 0 & t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.24}$$

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{3.25}$$

where t is the time elapsed since the last system update. Notice how the measurement update matrix H doesn't use the velocity to update the measurement, as the two las columns are zero vectors. With these equations it would be possible to estimate the velocity of the blob as it moves from only it's position in each frame and the time between frames, with the correction from noise.

3.4.3 Tracking Correspondance Test

Besides approximating the measured values to the correct values, the Kalman filter can also be used to estimate the best possible match from a set of several measurements. Using the Mahalanobis distance

$$M = z_k^T (H P_k^- H^T + R)^{-1} z_k \tag{3.26}$$

a new measurement is compared to the predicted measurement, where the matrix P_k^- is the predicted error covariance and R is the measurement noise. The best match will be the one with the smallest distance. It is also possible to use a validation gate

$$M < \rho \tag{3.27}$$

to filter measurements. This best match is then used to correct the state prediction, and complete the tracking step for one iteration.

3.4.4 Motion Detection and Recording

Once the object is ready to be tracked, it is then necesary to know when a person started to move it and when it is back to a resting position. A simple yet effective way to do this is to define triggering parameters that activate and deactive the recording of motions. When the object is resting, and a parameters of the object's state vector fulfills the starting requirements, the systems starts recording the movements of the object. When it is moving, the system checks for fulfillment of the stopping triggers. The entire set of observations in the trajectory are stored as a single motion.

3.5 Stereo Triangulation

There are typically three processes that must be done to obtain a 3d point from two 2d points in two images. First the physical charateristics of each camera and the point of them relative to each other are estimated, usually known as the calibration step. Next, the pixels correspondances of interest between images are found and finally the actual estimation of the 3d point is done. Each of these steps will be described below.

3.5.1 Pinhole Camera Model

The simplest model for a camera is defined by three components: an *image or retinal plane* R, the *optical center* C or *focal point* which doesn't belong to R and the distance f between R and C called the *focal length*. An illustration of this model is seen in Figure 3.6.

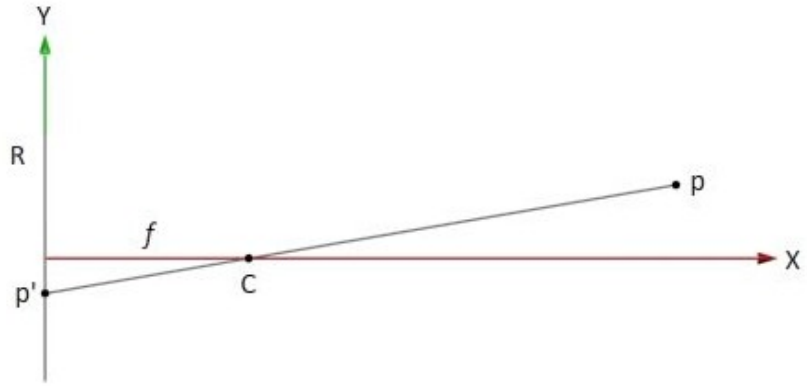


Figure 3.6: The pinhole camera model in 2d.

A point $p\prime$ in the image plane R is the intersection of the optical ray from the point p in space passing through the optical center. The line perpendicular to R that goes through the optical centre C is called the *optical axis*. The point where it intersects the plane R is called the *principal point*. There are two coordinate frames of interest in the pinhole camera model. The first one is the orthonormal 2d plane that is in the image plane. The second one is the *camera coordinate system* centered at the optical center with the plane defined by two of it's axis being parallel to the image plane. Figure 3.7 shows the pinhole camera model with the image plane in front of the optical center.

The relationship by similarity triangles of a point $p = (X, Y, Z)^T$ in the camera coordinate frame to and it's projection $p\prime = (x_c, y_c, f)$ in the image plane is given by

$$x_c = f\frac{X}{Z}, \quad y_c = f\frac{Y}{Z}. \tag{3.28}$$

This can be written with homogeneous coordinates as a matrix multiplication operation

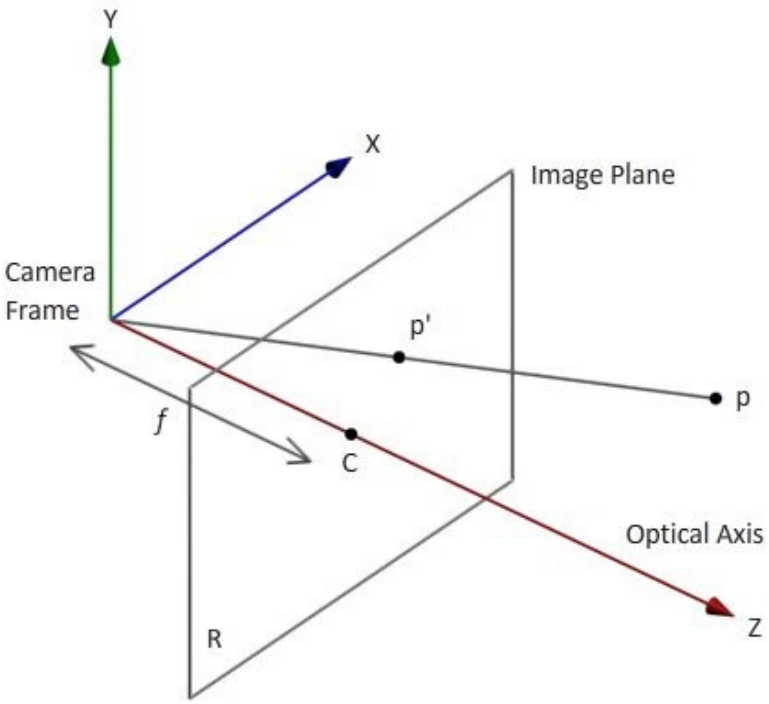


Figure 3.7: The perspective camera model.

$$\begin{pmatrix} Zx_c \\ Zy_c \\ Z \end{pmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = P \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{3.29}$$

where P is the projection matrix. If $p\prime$ is normalized dividing by the third coordinate the point is said to be in the *normalized image plane*.

3.5.2 Intrinsic Camera Parameters

Besides the camera projection, other physical and digitalization factors are modeled to more accurately define the transformation of points from the 3d space to a 2d image plane. These come from the transformation of the 3d point in the camera coordinate frame, to a projection on the 3d normalized image plane and finally a 2d mapping to the image in pixel coordinates. Many texts [**?**, **?**, **?**, **?**] explain in detail the origins of the intrinsics parameters, here it will be limited

to the definition of the intrinsic parameters as a matrix

$$K = \begin{bmatrix} \frac{f}{s_x} & s & c_x \\ 0 & \frac{f}{s_y} & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.30}$$

where s is the skew factor and c_x and c_y are the coordinates of the principal point. The s_x and s_y values represent the scaling applied to the points in the image plane so they are transformed to the pixel coordinates. The values are defined separately as the sensors in the physical cameras might not be square. The unit of the scale parameters in this form depend on the units used for the focal length, but are usually meters/pixels or mm/pixels.

In homogenuous coordinates the instrinsic matrix is

$$M = \begin{bmatrix} K & 0 \end{bmatrix} = \begin{bmatrix} \frac{f}{s_x} & s & c_x & 0 \\ 0 & \frac{f}{s_y} & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{3.31}$$

3.5.3 Undistortion

The pinhole camera model is only an approximation of the real camera projection. Because of the anormalities inherent to the physical construction of the camera, and the deformation of the image as a result of the curvature of the lens the final image is distorted in comparison to the ideal projection model.

Figure 3.8: Examples of radial distorsion.

The two most common distortions treated in computer vision literature are the radial distorsion and the tangential distortion. They are both stored as two sets of coefficients which can distort an undistorted image, result of the projection in the normalized image plane. Using an inverse transformation it is possible to undistort images which is an almost necesary step for performing an accurate estimation of the 3d structure of a scene.

The radial distortion can be approximated using the following expression:

$$\triangle x_r = x_c(1 + k_1 r^2 + k_2 r^4 + k_3 r^6 + \ldots) \tag{3.32}$$

$$\triangle y_r = y_c(1 + k_1 r^2 + k_2 r^4 + k_3 r^6 + \ldots) \tag{3.33}$$

where r is the distance from the point in the image plane to the optical center, $< k_1, k_2, k_3, \ldots >$ are the coefficients of the polynomial series. This model found in [**?**, **?**, **?**] and others, assumes the radial distorsion is symmetrical, equally affecting both x and y. Usually the model is sufficient to use only two or three degrees of the polynomial. A figure with the radial distortion model is shown in Figure 3.9.

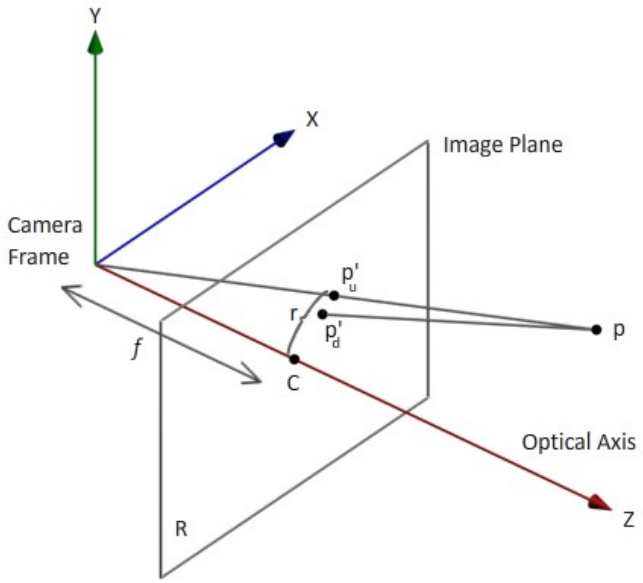


Figure 3.9: Perspective model with radial distortion.

Centers of curvature of the lens surfaces are not strictitly colinear, in other words, the lens may not be completely aligned to the sensor in the camera, but has a slight inclination and possibly offset from the center of the image. This distortion is often modelled as

$$\triangle x_t = p_1 y_c + p_2(r^2 + 2x_c^2) \tag{3.34}$$

$$\triangle y_t = p_2(r^2 + 2x_c^2) + 2p_2 x_c \tag{3.35}$$

where $p1$ and $p2$ are the coefficients for the tangential distortion. Both radial and tangential distortions are added to the projection to obtain the final distorted point in the normalized image plane

$$x_d = x_c + \triangle x_r + \triangle x_t \tag{3.36}$$

$$y_d = y_c + \triangle y_r + \triangle y_t \tag{3.37}$$

that is then afterwards expressed as pixels after scaling and offsetting with the image center.

The inverse problem of obtaining the normalized image projection p' from the distorted pixel coordinate $p'_d = (x_d, y_d)$ is not a simple one because of the high degree distortion model. Fortunately OpenCV [**?**] provides the tools for obtaining this value either by undistorting and entire image, or an array of 2d points. Figure 3.10 shows the result of undistorting an image after calibration using OpenCV.

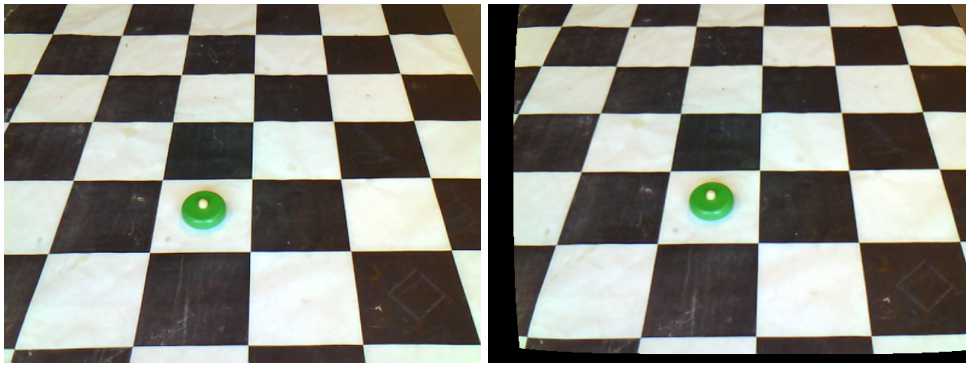


Figure 3.10: Original image and result after undistortion.

3.5.4 Extrinsic Camera Parameters

In the previous definition of the projection matrix in eq 3.29 the 3d point being projected was located in the coordinate frame of the camera. If a point is in another coordinate frame, i. e. the world coordinate frame, it must be transformed to the camera's coordinate frame to be used in the projection matrix. The relationship between the coordinate frame of the camera an another coordinate frame can be represented by a translation vector T and a rotation matrix R. A point p_w in the reference coordinate frame, is located at the point

$$p_c = R^{-1}(p_w - T) = R^{\tau}(p_w - T) \tag{3.38}$$

where R is the rotation of the camera with respect to the other coordinate frame and T it's translation. The rotation matrix has a property that its inverse is its transpose; hence $R^T R = R^T R = I$ where I is the identity matrix. Figure 3.11 illustrates the transformation of a point between coordinate frames.

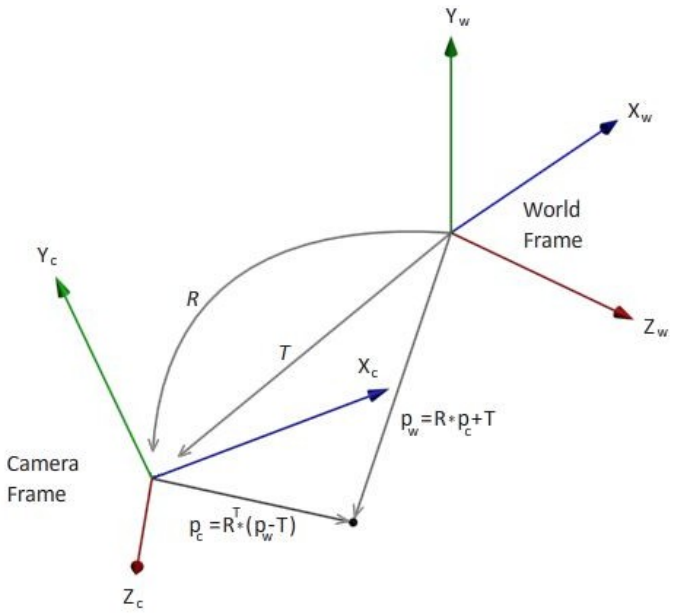


Figure 3.11: Transformation between world and camera coordinates.

The relationship between the camera coordinate frame and another one is defined as the extrinsic parameters, and it is represented as a transformation matrix in homogeneous coordinates

$$H = \begin{bmatrix} R^T & -R^T t \\ 0 & 1 \end{bmatrix} \tag{3.39}$$

which when multiplied with a point in the external coordinate frame will give the corresponding point in the camera's coordinate frame. The projection matrix taking into account intrinsic and extrinsic parameters becomes:

$$P = MH = \begin{bmatrix} \frac{f}{s_x} & s & c_x & 0 \\ 0 & \frac{f}{s_y} & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R^T & -R^T t \\ 0 & 1 \end{bmatrix}. \tag{3.40}$$

3.5.5 Calibration

The objective of the calibration process is to estimate both the intrinsic and extrinsic parameters of the camera. In the case of a stereo setup it is common to select one of the cameras as the world coordinate frame, and the other camera positioned relative to this frame. Other times a third coordinate frame, usually at the same distance from both cameras is used as the world. Here the former is the method of choice.

The process of calibrating a camera usually involves minimizing the error between the projection of some known 3d points using the intrinsic and extrinsic parameters of two cameras and the actual measured points in the images. Camera calibration has been treated in detail to solve many computer vision problems and it is out of the scope of this thesis to write the entire theory behind it, but good introductions to stereo calibration can be read from [**?**, **?**, **?**].

The standard calibration algorithm for a stereo setup uses a set of corresponding image points obtained by taking pictures of a chessboard calibration pattern. The pattern has uniform sized blocks of alternating white and black filling, which result in easy to identify corners. By knowing the geometry of the corners, it is possible in each snapshot to identify the corners that are in the pattern and discard erroneous ones. Figure 3.12 shows a sample of the calibration pattern with the corresponding detected points. Using this corresponding points in the images and with the mathematical model of intrinsic and extrinsic camera parameters, the unknowns can be obtained with sufficient number of samples.

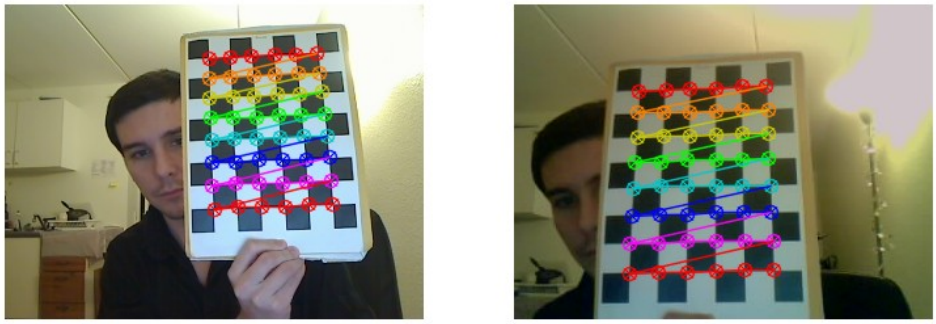


Figure 3.12: Two images from the left and right camera showing the calibration procedure.

Once the parameters have been found it is now possible to put all the points in the image planes into a common coordinate frame to work with them. This is needed for the triangulation step in section 3.5.8.

3.5.6 Epipolar Constraint

In a typical stereo vision setup two cameras are placed next to each other sharing a common viewing volume. The Figure 3.13 is describing such system.

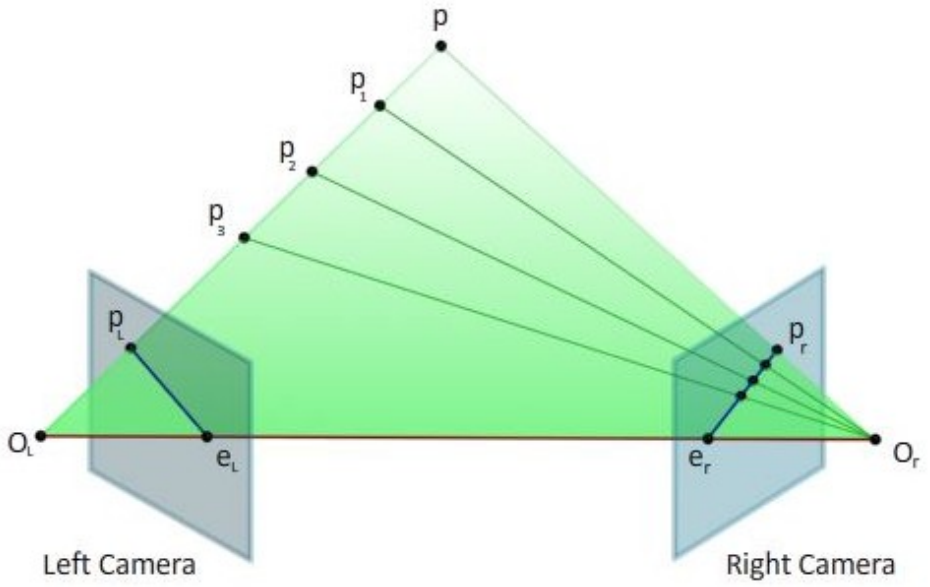


Figure 3.13: Epipolar geometry.

A point p inside the shared volume is visible in the image planes of both cameras. This point p forms a 3d plane called the *epipolar plane* with the two points p_l and p_r seen by the cameras in their respective image planes, also with the two camera focal points O_l and O_r. The line that goes from the focal point O_l of the left camera to the focal point O_r of the right camera is called an *epipolar line*, and their intersections with their respective image planes, e_l and e_r are called the $epipoles$. The line $p - O_l$ is seen by the left image as a point p_l but in the right image it is seen as an *epipolar line* on the epipolar plane, defined by the projection p_r on the right image plane and the epipole e_r. All points p_r lying on this epipolar line on the right image are possible projections of p. It is a matter of finding the corresponding p_r in the epipolar line for a given p_l on the left image, and using the transformation between both image planes to estimate the position of p.

The cameras' local coordinate systems are related to one another by a rotation R and translation T in the world coordinate system. If R and T are the rotation and translation of the right camera with respect to the left camera, a 3d point

p_l in the left camera's image plane is related to the observation of the same 3d point seen in the right camera plane in the coordinate frame of the left camera as the point p_r, by the equation

$$p_r = R * p_l + T. \tag{3.41}$$

If R is defined as the rotation of the left camera with respect to the right one, the same 3d point seen in the right camera plane in the coordinate frame of the left camera as the point p_r is

$$p_r = R^T(p_l - T). \tag{3.42}$$

Given that both equations have the same purpose but depend on the input parameters R and T it is important to know what they mean exactly as they are provided by some calibration system. In OpenCV, the calibration routine returns a rotation and translation for the second case.

With this equations it is possible to obtain two sets of poins points $(O_l,\, p_l)$ and $(O_r,\, p_r)$ in a single 3d coordinate frame, each pair defining a line that can be used for triangulation of the 3d point.

3.5.7 Object Correspondances

The most common approach in computer vision is two have a rectification step to find correspondances using a baseline. This is not the procedure done in this thesis. As the correspondances are found directly using the tracking module the interest is to find the position the object in each of the normalized image planes. This is easily obtainable applying the inverse transformation of the intrinsics matrix. For a given point p_{img} in the 2d image in pixel coordinates, the corresponding point p_c in the normalized image plane is

$$p_c = \begin{pmatrix} x_c \\ y_c \\ 1 \end{pmatrix} = K^{-1} p_{img} \tag{3.43}$$

where x_c and y_c are the coordinates of the point in the normalized image plane. Using this previous equation 3.43, the line from the camera focal point to the image plane can be defined in parametric form as

$$r(t) = C + t(p_c - C). \tag{3.44}$$

as can be seen in the Figure 3.14. Notice that the focal point previously represented by O has been replaced with C so it's easier to identify it as belonging to the camera in the next sections.

3.5.8 Triangulation

The triangulation step is in charge of estimating the 3d position of a given observation visible in two or more cameras. For the interest of this thesis only a configuration with two cameras is considered.

The points C_l and C_r correspond the the focal points of the left and right cameras respectively, and p_l and p_r correspond to the projections of the point p in the left and right image planes. The two lines l_l and l_r originating from the focal points and each passing through the corresponding projection p_l and p_r respectively, intersect in space at point p. The the points $p_1$and p_2 that lay in the each one of the lines can be defined as

$$p_1(t) = C_l + t(p_l - C_l), \qquad t \in \mathbb{R} \tag{3.45}$$

$$p_2(t) = C_r + t(p_r - C_r), \qquad t \in \mathbb{R} \tag{3.46}$$

The easiest solution for finding the point p would be to solve the system of two equations to find the parameters $t_1$and t_2 for which $p_1(t_1)$ and $p_2(t_2)$ are equal. Unfortunately with real measurements, given the noise in the images, the calibration errors and the discretization of the projections into pixels, the lines will most likely not intersect.

It may be shown that two skew (non parallel and non colinear) lines have a common perpendicular line to both. The length of this line is also the shortest distance between the two lines. Therefore one solution is to find the shortest segment between the two lines and choose the midpoint of this segment as the 3d reconstruction of the two observations in the images, as seen in Figure 3.14.

Let the points $P_1$and P_2 be the endpoints of the segment joining the two lines, for some values t_1 and $t_2$of the parameter t of each the corresponding line equations 3.45, 3.46. Then using the orthogonal restriction, the dot products

$$(P_1 - P_2) \cdot (p_l - C_l) = 0 \tag{3.47}$$

$$(P_1 - P_2) \cdot (p_r - C_r) = 0 \tag{3.48}$$

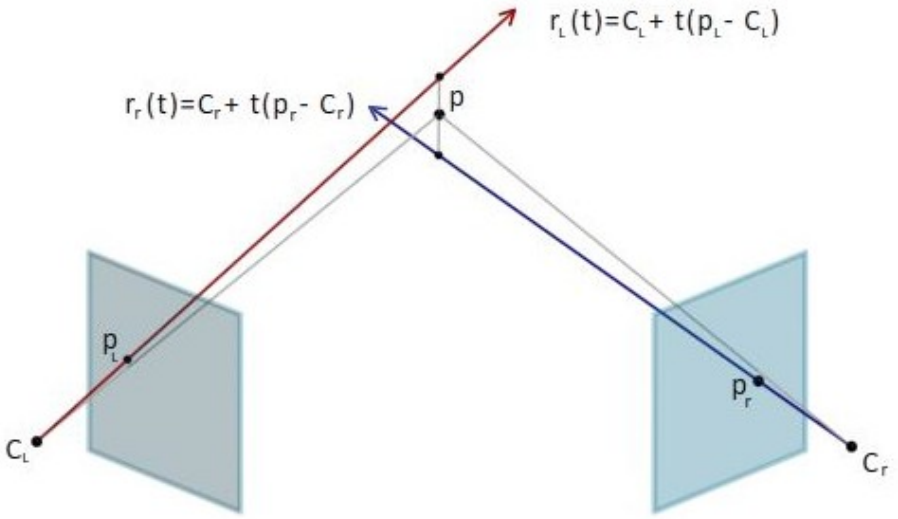


Figure 3.14: Triangulation of a point using the middle point of the shortest segment between two skew lines.

are zero. Expressing this in terms of the line equation we obtain

$$[(C_l - C_r) + t_1(p_l - C_l) - t_2(p_r - C_r)] \cdot (p_l - C_l) = 0 \tag{3.49}$$

$$[(C_l - C_r) + t_1(p_l - C_l) - t_2(p_r - C_r)] \cdot (p_r - C_r) = 0 \tag{3.50}$$

After expanding and simplifying the equations, is possible to obtain t_1 and t_2 for a solution in the three-dimentional space.

Replacing t_1 and t_2 in the line equations 3.45 and 3.46, the points P_1 and P_2 defining the segment can be obtained. The 3d reconstructed point is estimated as the midpoint of the segment by averaging:

$$p\prime = \frac{P_1 + P_2}{2}. \tag{3.51}$$

An example showing the triangulation of an object is shown below in Figure 3.15.

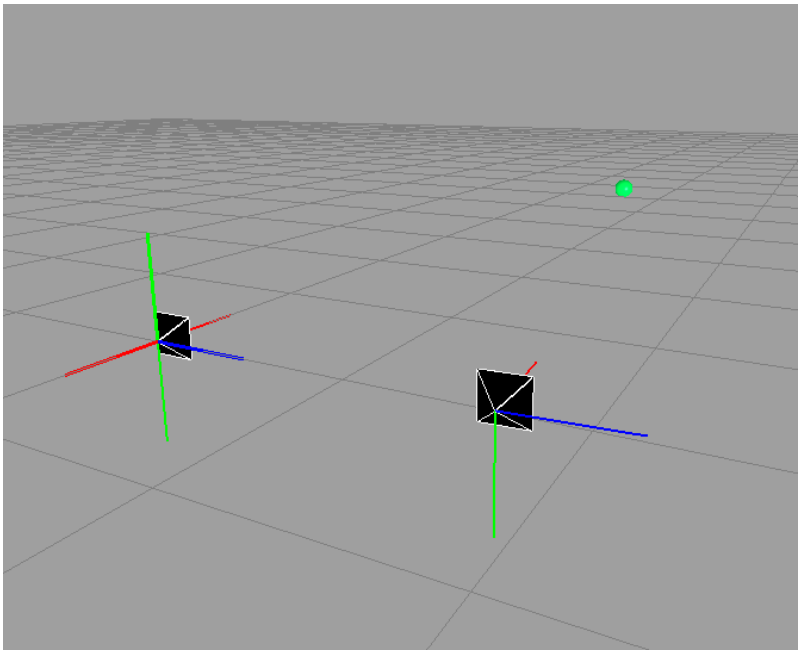


Figure 3.15: Stereo setup and triangulation of an object's position.

3.6 Learning Action Primitives

The final goal is to represent human actions as a set of sequential action primitives performed on an object. Many actions will be efficiently described by a single model reusing common action primitives between the object motions. In the traditional hidden Markov model approach, each set of samples from a specific motion would have a separate model. Here, the idea is to build one single HMM from all the observations and all the sample motions performed on the object.

As the main objective of this thesis is to extend the work done in [**?**] by adding a real-time stereo tracking and a learning system, most of this section will expand from the relevant theory and work already contained there.

3.6.1 Model Building

The model building process starts by parametrizing each motion trajectory as a HMM λ_i for the i-th trajectory. Each state in the HMM is defined by a multivariate Gaussian distribution. The aim is to find a HMM λ_F that represents all these HMMs from the motion trajectories and in which all the observation sequences can be expressed as a sequence of states in λ_F.

To create this joint model a distance measurement is used to compare the Gaussian in each state of λ_i for all i. Those which are similar are joined creating a new Gaussian. The final hidden Markov λ_F model is built with the Gaussians that are unique in each obsevations' HMM and the joint Gaussians. The state transitions are defined by the motion trajectories in the observations.

After λ_F has been created the Viterbi algorithm is used to estimate the most likely state sequences $< s_1 s_2 s_3 s_4 ... s_k >$ defining each of the objects' motion trajectories used for training. These sequences are then expressed as state changes by removing multiple continuos ocurrences. Using these state changes the action primitives are identified as the longest common substring of states in all the sequences. The actions model is finally represented as an acyclic directed graph, having the first state in the HMM as the root action, and adding nodes and edges according to the state sequences, represented as actions. This process is described in detail in the following subsections.

3.6.2 Hidden Markov Models

Hidden Markov Models (HMM) have been successfully used in speech recognition [**?**], hand gesture recognition [**?**] and many other applications [**?**, **?**, **?**]. Following is the description of HMM using the same notation as in [**?**].

A HMM is a statistical model composed of N states, M observation symbols, the state transition probability distribution A, where

$$a_{ij} = P[q_{t+1} = S_j | q_t = S_i], \qquad 1 \leqslant i, j \leqslant N \tag{3.52}$$

is the probability of transitioning from state S_i to S_j in one time-step, the observation symbol probability distribution B in state j, where

$$b_j(k) = P[v_k \, at \, t | q_t = S_j], \qquad \begin{matrix} 1 \leqslant j \leqslant N \\ 1 \leqslant k \leqslant M \end{matrix} \tag{3.53}$$

is the probability of observing the symbol v_k when transitioning to the state S_j and finally the initial state distribution π, where π_i is the probability of starting in state S_i. It is common to denote the HMM in a compact notation as

$$\lambda = (A, \, B, \, \pi). \tag{3.54}$$

Instead of using discrete symbols for the observations, the specific HMM model used for the motion trajectories will use states defined as continuous variables and the symbol probability distribution B is replaced by Gaussian distribution

$$b_j(O) = \Phi(O, \mu_j, \Sigma_j), \qquad 1 \leqslant j \leqslant N \tag{3.55}$$

where O is the observation vector, μ_j and Σ_j are the mean and covariance of the Gaussian probability density function $\phi$for the j-th state. The observation vector in theory can contain any of the features of an object, such as position, velocity, size, etc, and be defined by any type of PDF. In practice, only the position, velocity and acceleration are obtained after triangulation and filtering with the Kalman filter and a Gaussian distribution is the optimal distribution.

3.6.3 Preparing Data for Model Building

Each trajectory T_i in the set of motion trajectories $< T_1, T_2, ..., T_k >$ is divided into segments using the arc length of the trajectory. Each segment is defined as as a multivariate Gaussian distribution using a subset of continuous observations $< O_{ij}, O_{i(j+1)}, ..., O_{i(j+n)} >$ from a motion trajectory T_i whose combined arc lenght is larger than a minimum value. The general arc length equation

$$L = \int \sqrt{\sum_{feature\, X} (\frac{dX}{dt})^2 dt} \tag{3.56}$$

measures the variation of the features in time between observations. The segments extraction process starts by measuring the arc length of the first consecutive pair of observations $< O_{j1}, O_{j2} >$ and continues adding up the length of the next consecutive observation pairs until the length is larger than a minimum arc length or it reaches the end of the trajectory. When the feature space on the observations is the position in the 3-dimentinal space, the arc length for n observations can be approximated to

$$l = \int \sqrt{(\frac{dx}{dt})^2 + (\frac{dy}{dt})^2 + (\frac{dz}{dt})^2} dt \sim \sum_{i=1}^{N} \triangle x_i^2 + \triangle y_i^2 + \triangle z_i^2 \tag{3.57}$$

using the Pythagorean Theorem. When the distance l is larger than the minimum, the *i-th* segment is identified as a state s_{ji} in the HMM λ_j for the trajectory T_j and parametrized as a multivariate Gaussian using the mean μ and covariance Σ equations

$$\mu = \frac{1}{N} \sum_{i=1}^{N} x_i \tag{3.58}$$

$$\Sigma = \frac{1}{N-1}\sum_{i=1}^{N}\sum_{j=1}^{N}(x_i - x_j)^2 \tag{3.59}$$

where N is the number of states in the sequence.

If the length of the last segment in the observation is less than a percentage (2/3 was the choice) of the minimum length, it is joined with the previous segment, otherwise it is considered a new segment.

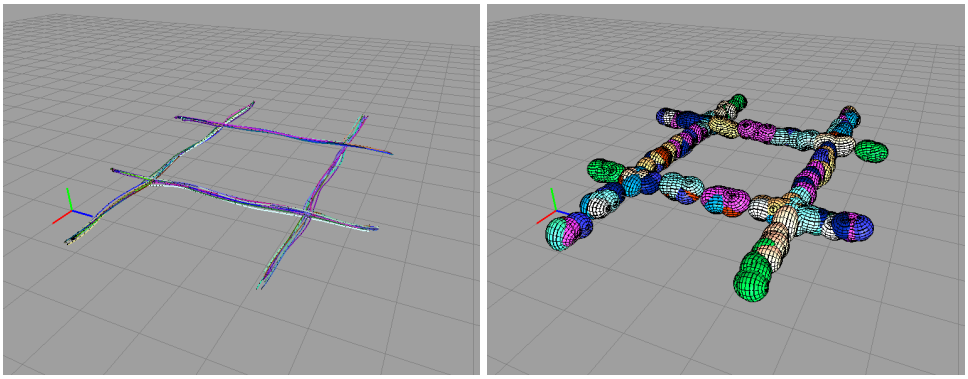


Figure 3.16: State Gaussians obtained from a set of motion trajectories. The ellipsoids show the contour of the Gaussians. The color differentiates the index in the array of states for each trajectory.

Once the observations in the trajectory have been covered by Gaussians, the creation of the HMM λ_j for the trajectoy T_j is completed by defining it as a bounded Left-Right model as in Figure 3.17, with each state being able to transition with the same probability to itself and the next state in the sequence only, as seen in eq 3.61.

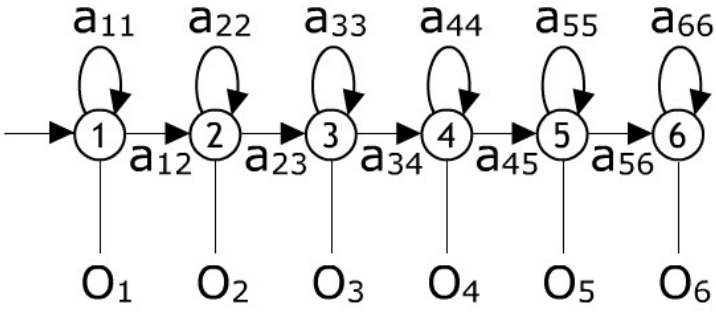


Figure 3.17: Example of a Left-Right HMM

The initial state probabilities π and the transition probability matrix A for the bounded Left-Right HMM have the form:

$$\pi_i = \begin{cases} 1, & i = 1 \\ 0, & i \neq 1 \end{cases}, \qquad 1 \leqslant i \leqslant N \tag{3.60}$$

$$A = \begin{bmatrix} 0.5 & 0.5 & 0 & 0 & . & . & 0 \\ 0 & 0.5 & 0.5 & 0 & . & . & 0 \\ 0 & 0 & 0.5 & 0.5 & . & . & 0 \\ 0 & 0 & 0 & 0.5 & . & . & 0 \\ . & . & . & . & . & . & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.61}$$

3.6.4 Localizing observations

In some escenarios an action is defined by the interaction with an object in the object's local coordinate frame. We can imagine that the object starts to move from the origin in each of the sequences. As the different observations collected can happen in different places in world space, it is therefore needed to localize the observations to a single frame of reference, the object space. This is done by subtracting the first observation O_{j1} from all the observations O_{ji} in a trajectory T_j. This transformation removes the location dependency. Figure 3.18 shows the states as Gaussians obtained from a set of recordings with offsetting aplied.

In Figure 3.18 it can be noticed that only the position of the Gaussians is offset by the localization but the rotation of the entire motion remains the same.

In the application developed a separate helper coordinate frame can be used to transform the world space observations O_{ij} to this this helper coordinate frame. This allows for the observations to be independent of the camera position, and instead, relative to a coordinate frame which can be chosen during the system setup. When the HMM of the motion trajectories is already built this coordinate frame will allow to use it even if the stereo camera setup is positioned at a different place and orientation in relation to the training observations, assuming the calibration of the cameras is doesn't change the scaling of the data.

3.6.5 Divergence Test

The next step in building the model is to identify the states in the HMMs created for each trajectory which are common between two or more trajectories. This requires for a similarity measurement and the definition of a threshold that can classify two states as similar or different. The measurement function used is the Kullback-Leibler divergence. In the case of two Gaussians $\Phi_{ik} = \Phi(x, \mu_{ik}, \Sigma_{ik})$

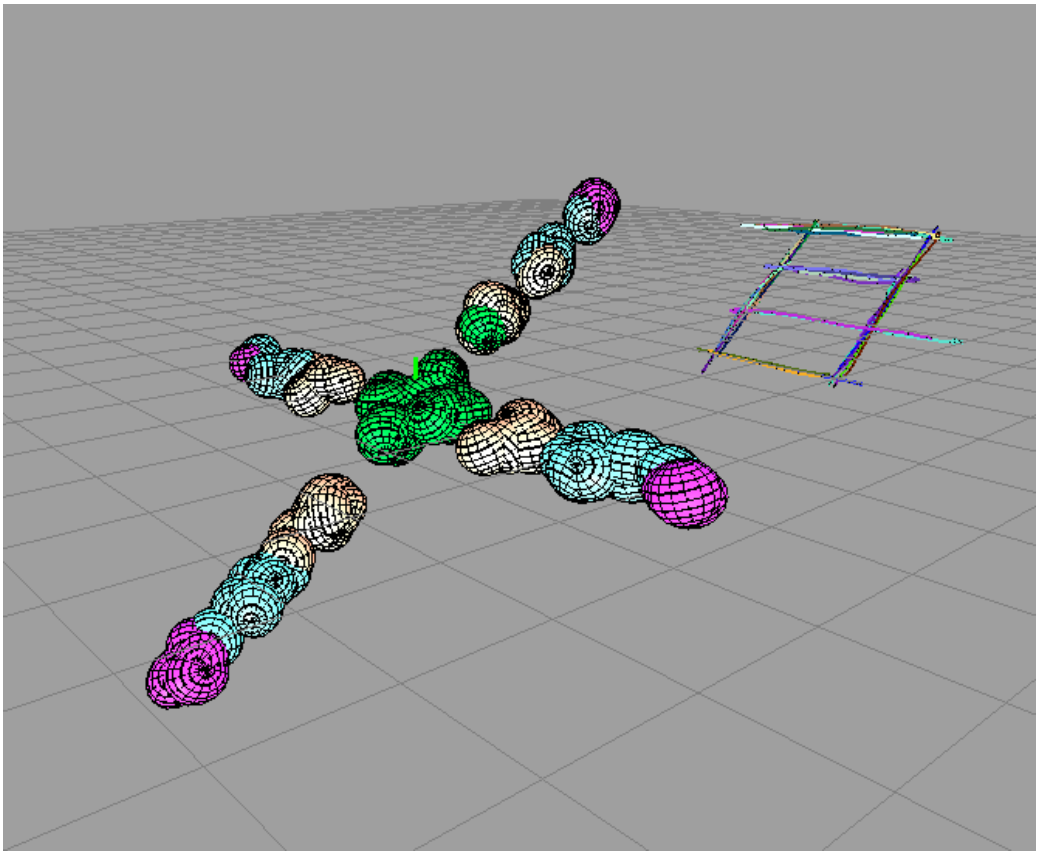


Figure 3.18: Observations localized by offsetting with the first observations of each trajectory

and $\Phi_{jl} = \Phi(x, \mu_{jl}, \Sigma_{jl})$ of two states (s_{ik}, s_{jl}) from λ_i and λ_j respectively, the divergence with a close form solution as

$$D_{KL}(\Phi_{ik}||\Phi_{jl}) = \frac{1}{2}\left(log\frac{|\Sigma_{jl}|}{|\Sigma_{ik}|} + Tr(\Sigma_{jl}^{-1}\Sigma_{ik}) + (\mu_{jl} - \mu_{ik})^T\Sigma_{jl}^{-1}(\mu_{jl} - \mu_{ik}) - n\right) \tag{3.62}$$

where n is the dimension of the space spanned by the random variable x, μ and Σ represent the mean and covariance and Tr is the trace of the matrix resulting from multiplying Σ_{jl}^{-1} and Σ_{ik}.

The Kullback-Leibler divergence is an assymetrical metric, meaning that the order of the arguments gives different resuts. A symmetric measurement can be obtained with

$$D'_{KL}(\Phi_{ik}||\Phi_{jl}) = \frac{D_{KL}(\Phi_{ik}||\Phi_{jl}) + D_{KL}(\Phi_{jl}||\Phi_{ik})}{2}. \tag{3.63}$$

3.6.6 Joining Gaussians

If the distance $D'_{KL}(\Phi_{ik}||\Phi_{jl})$ is less than a threshold θ for two Gaussians in λ_i and λ_j, then these Gaussians are considered as overlapping or in other words common to both HMMs. As such they will be joined into a single Gaussian built from both Gaussians in the states where

$$\Sigma^{-1} = \omega\Sigma_{ik}^{-1} + (1-\omega)\Sigma_{jl}^{-1} \tag{3.64}$$

and

$$\mu = \omega\Sigma_{ik}^{-1} + (1-\omega)\Sigma_{jl}^{-1} \tag{3.65}$$

are the inverse of the covariance and the mean of the new Gaussian. The parameter ω is a weighting factor defining the contribution from each of the states to the new Gaussian. As there is no reason to prefer one Gaussian or the other, the weighting factor is chosen as $\omega = 0.5$ to have equal preference for both Gaussian distributions.

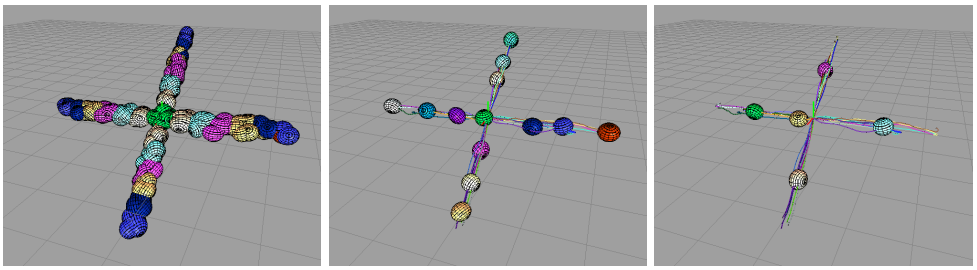


Figure 3.19: Two HMMs (middle, right) created from the Gaussians in the left image using different divergence thresholds.

Joining the Gaussians can have the possibly undesired effect that the starting location of the trajectory may be shifted towards some point far from the original position. This can be seen in the right image in Figure 3.19. One solution proposed for solving this is to join the initial Gaussians only with other initial Gaussians. The effect of doing this can be seen in Figure 3.20.

3.6.7 Updating the HMM

The final HMM λ_F starts as an exact copy of the first HMM, λ_1, of the trajectories. As this HMM can have many states and possibly some of them overlapping,

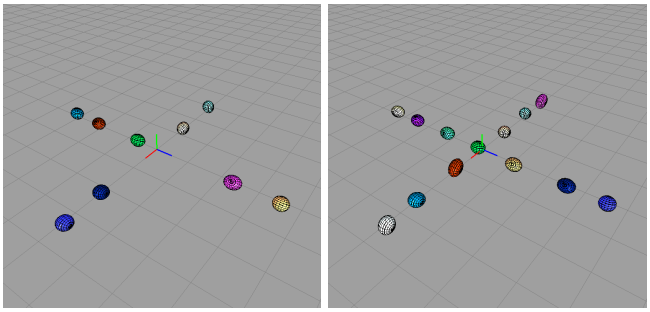
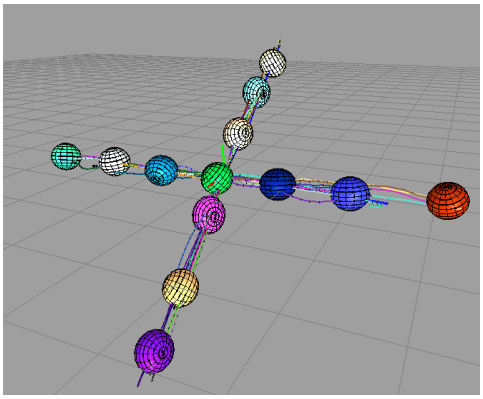


Figure 3.20: The left image shows a HMM trained with the first Gaussian without restriction. The right image shows a HMM trained equally but restricting the way the first Gaussian is joined. .

a pre-processing step is performed to join the Gaussians in the initial HMM using the divergence test. The initial transition matrix and state probabilities of λ_F are defined as a bounded Left-Right model as described in section 3.6.3.

The algorithm for building the model's HMM λ_F iterates through each trajectory's HMM λ_j to join and add new states. For a single HMM λ_i each state's s_{ik} Gaussian is compared to the states Gaussians s_{Fj} in λ_F by means of the divergence test defined in Equation 3.63. If the Gaussian of s_{ik} is found to be overlapping with a Gaussian from s_{Fj}, the pair is added to a list of states to be joined. If not, s_{ik} is added to a list of non-overlapping states. After all the states have been compared and classified, the model is updated by joining the Gaussians and adding the new ones.



	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0.2	0.2	0	0.2	0	0.2	0	0	0.2	0	0	0	0
2	0	0.5	0.5	0	0	0	0	0	0	0	0	0	0
3	0	0	0.5	0	0	0	0	0	0	0	0	0	0.5
4	0	0	0	0.5	0.5	0	0	0	0	0	0	0	0
5	0	0	0	0	0.5	0	0	0	0	0	0	0.5	0
6	0	0	0	0	0	0.5	0.5	0	0	0	0	0	0
7	0	0	0	0	0	0	0.5	0.5	0	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0.5	0.5	0	0	0
10	0	0	0	0	0	0	0	0	0	0.5	0.5	0	0
11	0	0	0	0	0	0	0	0	0	0	1	0	0
12	0	0	0	0	0	0	0	0	0	0	0	1	0
13	0	0	0	0	0	0	0	0	0	0	0	0	1

Figure 3.21: Model and transition matrix obtained from joining many HMMs from different trajectories.

In either case the transition matrix also has to be updated. In the case of two states being joined, when a state s_{Fj} from λ_F is joined with a state s_{ik} from a motion's HMM λ_i, the transition matrix of λ_F must be updated by defining a transition to the state $s_{i(k+1)}$ in λ_i. Similarly, the state transition to s_{ik} from the

state $s_{i(k-1)}$ preceding s_{ik} has to be updated to transition to the s_{Fj}. After all the states have been joined, the remaining states are appended to the end of the HMM. For a correct model to be built, the first state in every observation's HMM must always overlap with the first state in λ_F. A posterior step normalizes all the transition probabilities.

3.6.8 Identifying the Motions in the Model's HMM

After the hidden Markov model λ_F for the set of actions performed on the object has been trained and possibly having joined some of the states, the constructing observation's HMMs $< \lambda_{1,}\lambda_2, ..., \lambda_n >$ can be discarded. The next step is to identify the sequence of steps in the new HMM λ_F that best represents each trajectory T_j. This can be obtained using the Viterbi algorithm [**?**] which is defined in the Appendix C. The algorithm receives a vector with the trajectory observations $< O_{j1}, O_{j2}, ..., O_{jn} >$ for each motion trajectory λ_j and finds the path as a sequence of states $< s_{F1}, s_{Fk}, ..., s_{Fy} >$ in λ_F that outputs the highest likelihood for an observation.

3.6.9 Finding Action Primitives

The general definition of an action primitive here is of a sequence of states defining a unique path in the trajectories. Ideally, we want to maximize the amount of information each action contains by minimizing redundancy while still being able to represent the trajectories correctly. Once all the trajectories have been identified in the HMM as state sequences the next step is to eliminate some of this redundancy by expressing the sequences as states changes, as shown below

$$\underbrace{s_1, s_1, \ldots s_1}, \underbrace{s_2, s_2, \ldots s_2}, \underbrace{\ldots}, \underbrace{s_k, s_k, \ldots s_k}$$
$$\Downarrow$$
$$s_1, s_2, \ldots s_k$$

The new sequences defined as state-changes remove the multiple occurrences of the same state in an observation. The information retained now is the path taken by the object in the trajectory, without caring about how long it stayed in which state. If each state in the sequences is considered a letter, and each sequence a string, the common sub sequences between all the sequences of state-changes can be defined as finding the *longest common sub strings*. The Longest Common Substring problem is a well know problem in strings manipulation. The algorithm implemented is shown in Appendix D can be found in [**?**].

Each of the longest common substrings or subsequences found in the state-change sequences of all the trajectories is an action primitive a_i. The set of

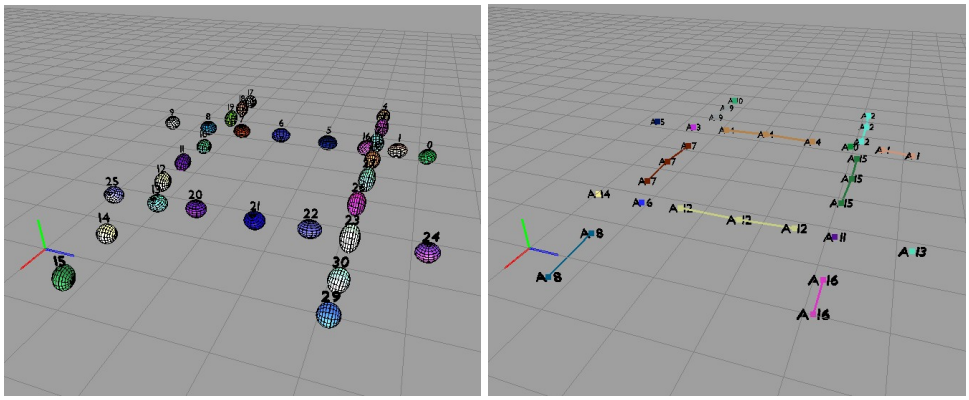


Figure 3.22: Model and action primitives obtained from the model.

action primitives A contains the individual subactions performed in the motion trajectories, when executing an action on an object. This higher level abstraction of the trajectories requires a higher level model instead of the HMM, to represent the different actions performed with the different trajectories. A suitable model for this is a *directed graph*.

3.6.10 Actions Primitives Graph

A graph [**?**] is a an abstract representation of a set of objects where some of the objects are connected by links called edges. Each object is represented by a vertex or also commonly known as nodes. The mathematical notation for a graph is $G = (V, E)$ where V is a set of vertices and E is a set of edges which is a set of pairs of vertices from V.

A directed graph is a type of graph in which the edges between two objects have a specific direction, starting at one vertex and ending at the connected vertex. It is represented by an ordered pair $D = (V, A)$ where V is the same set of vertices as in a graph and A is a set of *arcs* which is has ordered pairs of vertices from V.

The motion trajectories as action primitives can be represented as a directed graph. A root vertex will exist and be defined by the common action primitive obtained from the starting state of sequences. Nevertheless it is also possible to have more than one starting action primitive. Each node or vertex in the graph is an action primitive. Two nodes in the graph are connected if they appear together in some sequence, with the direction defined by the order in which they appear. Traversing the graph from a starting node to another node defines a trajectory (not necessarily a correct one).

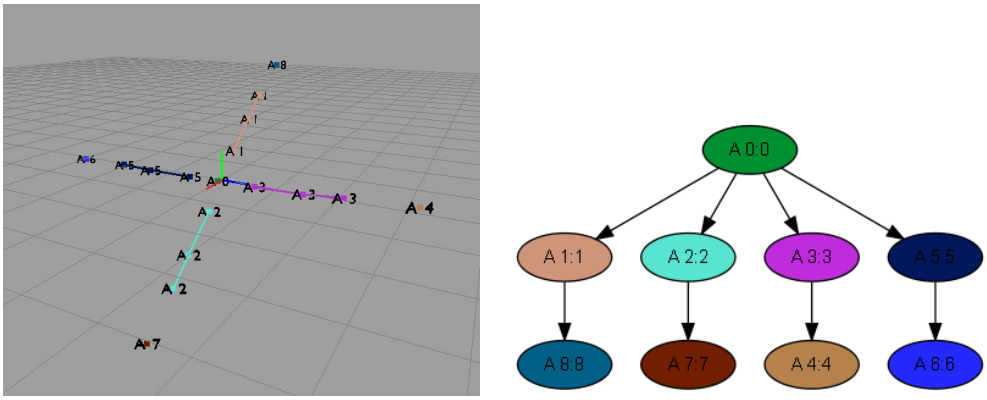


Figure 3.23: Action primitives and action primitives graph obtained from the model.

3.6.11 Trajectory Validation

Given a directed graph of action primitives and it's corresponding HMM where each node is assigned to one action, the sequence of steps to validate a trajectory a new trajectory T_j is the following:

1. Estimate the most likely sequence of states $< s_{F1}, s_{Fk}, ..., s_{Fy} >$ in λ_F that could produced such observation using the Viterbi algorithm, as described in section 3.6.8.
2. Find the action primitives sequence in the sequence of states 3.6.9. If a sequence of states doesn't have a corresponding action, the observation is not valid.
3. Parse the action primitives sequence using the action primitives graph. If the entire sequence is parsed completely, the sequence is valid and so is the observation. If the parser arrives at a dead end, where two actions identified in the states sequence are not connected, the observation is invalid.

This sequence of steps are used in the tests in Section 5 to validate the observations.

Chapter 4

Implementation

THIS section discusses the different modules designed and implemented for the system as well as a depiction of the hardware used.

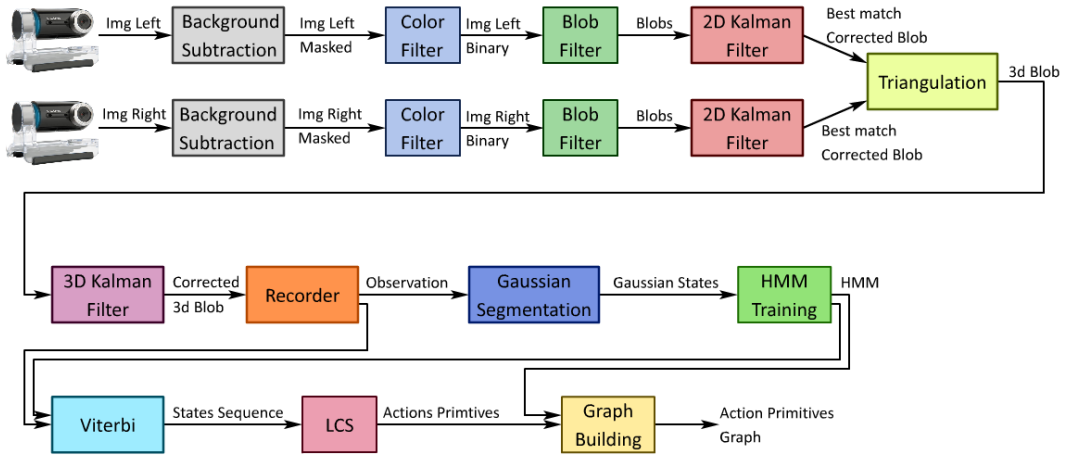


Figure 4.1: Pipeline of the entire Action Primitives Learning system defined in this thesis.

Two separate pieces of software were developed for this project. The first one is a set of classes exposed as a library which are the basis for all the algorithms and techniques described in the analysis section of this thesis. The name of the entire package of classes is named GToolkit. It was designed from the ground up as a series of separate modules which can be stacked one on top of the other to perform the steps necessary for tracking the objects and record their motion. The modules are separated as classes by functionality: camera, color filter, blob filter, Kalman filter, motion recorder, HMM trainer, HMM class, and a serializer.

The library makes use of OpenCV for matrix storage and operations, as well as other utilities. The serializer uses TinyXML to save and load the different objects as XML human-readable files. Some of the classes and structures define the data structures used in the library.

The second part is a graphical tool that allows all the functionality of the library to be exposed with a friendly interface. The editor uses the Qt library [**?**], which is a very robust and well established cross-platform application programming library. Both of this pieces of software will be explained in detail in the following sections.

4.1 Hardware

To track objects the system can use any type of camera recognized by OpenCV. It is currently restricted to use only one or two cameras. In should be easy to allow for more cameras if an algorithm for stereo reconstruction using more than two cameras is implemented in the GToolkit library.

The setup used for the project can be seen in Figure 4.2. Two cameras are connected to a computer via USB, feeding live images to the editor. The cameras used in this case were two Creative Optia AF USB Cameras, that can run at 30fps under normal lighting conditions.

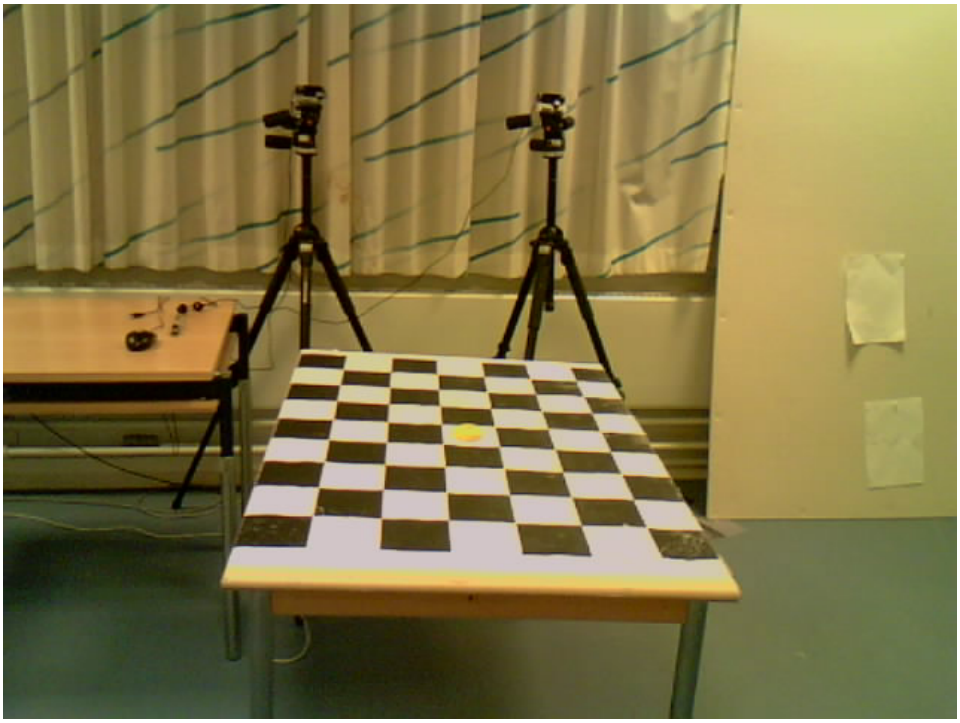


Figure 4.2: Screenshot of the stereo camera with the two USB Creative Optia AF USB cameras used in the thesis.

It is possible in Windows XP/Vista to configure the camera to disable all the automatic exposure and illumination compensation that it has active by default using a free utility called AMCap. By overriding these settings the cameras can be tweaked to optimize the performance and enhance the color in the images used for color tracking. A screenshot of the AMCap utility configuration is shown in Fig 4.3.

To track the objects, it is best if their color is homogeneous and ideally unique in the image. Even though the Mahalanobis distance with the Kalman filter can be used to avoid erroneous candidates during tracking as seen in section 3.4.3, it is best to limit the environment to have unique colors to improve the tracking result.

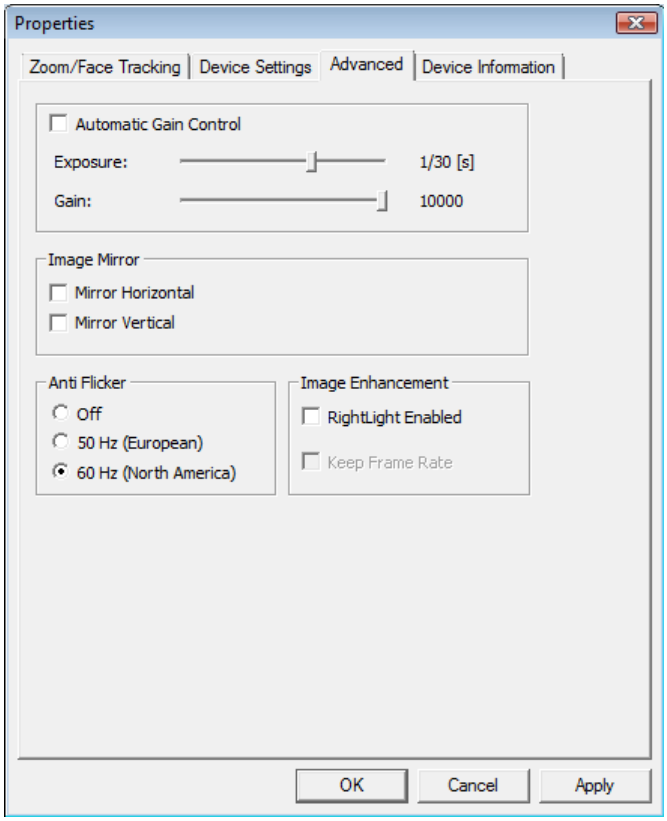


Figure 4.3: Screenshot of the camera device configuration dialog available in the AMCap utility for windows.

4.2 Software

4.2.1 GToolkit

Several classes were defined in the library to perform each of the steps required by the thesis. As stated previously one of the objectives with the design of the software was to be able to use each class as a separate module, to increase cohesion and reduce coupling. To accomplish this each functional module was defined as a class. The most relevant classes created are described below.

Camera

The camera class performs several functions. The first and most important is to define an interface for the library to easily make use of the camera devices

available in the system. It allows the acquisition of the camera device from the system and grab frames when requested. The frame grabbed is stored for later use in the application. Another function of the Camera class is to apply post processing filters if desired. The filters implemented are Median, Gaussian, Blur(Sobel), Dilation and Erosion. Dilation and erosion are useful for blob filtering as they remove a lot of noise from the images, at some performance cost.

The Camera class can also apply undistortion to the images as they are grabbed from the device using the radial and tangential distortion coefficients. This is implemented using the cvRemap function available in OpenCV. It can also store the internal and external parameters of the camera. Each instance of a Camera object defines one physical camera.

Finally an important feature of the Camera class is the transformation of the original grabbed image in the RGB space to one of the color spaces defined in section 3.2.1, an essential part of the color tracking step.

BackgroundSubtraction

This class is used to identify moving objects in images, by separating the foreground from the background. It implements the running Gaussian average as described in section 3.1.2. After passing an image format for initialization, each pixel holds the mean and variance describing the distribution for each color channel. When a new image arrives, each channel for each pixel is tested. If all the channels are classified as background the pixel is considered part of the background, otherwise it is classified as a foreground pixel. A selectivity flag filters the background update to be applied only after a pixel has been classified as background.

TrackableBlob

A TrackableBlob in the library is the implementation of the blob representation of an object in an image. This class holds all the properties to perform the color filtering, blob filtering, tracking and recording of trajectories. It also holds instances of each of the modules used for the previous steps. Every object being tracked has one TrackableBlob object per camera. This allows the object to be defined differently in each camera as may be required by the possibly different color, size, and other properties device dependent due to the positioning, environment and intrinsic characteristics of each device.

This class also holds the features measured by the blob filter at a given time, such as 2d position, bounding box, area and size. Other parameters obtained using the Kalman Filter are 2d velocity and acceleration. This data is stored in a class member struct of type BlobData.

TrackableObject

In the GToolkit a TrackableObject is the main class representing an object that is to be tracked in a video sequence. A TrackableObject inherits all the properties of a TrackableBlob, and also has an array containing a TrackableBlob for each camera. On every new frame, each one of the TrackableBlobs in the array should be updated. The data in these TrackableBlobs is used to update the TrackableObject they define. If only one TrackableBlob is used, the values of the single TrackableBlob area copied directly to the TrackableObject. If more than one camera is used, all the values in the different TrackableBlobs are combined to obtain the common values and also estimate the 3d information. For now only one or two cameras are supported in this estimation.

ColorFilter

This class uses color statistics previously defined to classify pixels in an image as belonging or not belonging to the distribution. It implements the four classifiers described in section 3.2.2: Color Thresholds, Mahalanobis Distance, Back Projection and Mixture of Gaussians. The ColorClassifier outputs a single channel 8-bit black and white image with black (0) meaning the pixel was classified as not belonging to the object and white (255) that it belongs.

To define the color properties of an object, a vector of pixel samples is stored in the ColorFilter object. Once the user is satisfied with the number of pixels obtained the method calcSamplesStatistics estimates all the parameters for each one of the different classifiers. To improve the performance of the filter, it is possible to set the number of pyramid levels to down-scale the image before the pixel classification. After it is done the image is up-scaled the same number of pyramid levels to have the same size as the original input image.

BlobFilter

The BlobFilter is in charge of finding up to a maximum number of blobs of a minimum size from a gray scale image, usually the image output of the ColorClassifier. To find the blobs it implements two different methods. The first one uses OpenCV contours related functions to find the blobs and obtain their properties. The second one is the implementation of the algorithm proposed for this objective in section 3.3.1. In practice it has been seen that the proposed algorithm is faster than OpenCV's, most likely because it has been specifically targeted at the requirements of the project.

After the blobs have been identified a posterior step compares the blob's properties to a set of user-defined filters, discarding blobs that do not match the filters.

Any property of a blob can be used as a filtes, such as position, velocity, width, height, area, and so on. This values are always in the scale of the original input image. Similar to the ColorFilter, the BlobFilter also allows to use pyramids to speed up the process, and this doesn't affect the scale of the data, with only incurring in some loss of precision inherent to downscaling.

The last action performed by the filter is to store the blobs in descending order sorted by an internal id. The id is not consistent between frames and shouldn't be used to identify an object between frames. For performance reasons this array of blobs has a fixed size, doubled at runtime every time the number of blobs surpasses the size of the array. Because of this, the id of some blobs is set to -1 to identify them as inactive. This id is useful externally to iterate over the array of resulting blobs until it has a value of -1.

KalmanFilter

Every TrackableBlob has an instance of a KalmanFilter object. The KalmanFilter receives an array of potential candidates in the image as of BlobData (measurements) and the BlobData (current state) of the blob whose data will be updated. The KalmanFilter will find the most likely match for the blob it is tracking measuring the mahalanobis distance in the array of measurements, as described in section 3.4.3. If the match is within a threshold it is used to apply the Kalman correction for the data of the blob.

Before it can be used the KalmanFilter must be initialized, by defining the measurement and state vectors parameters. The library was written so that the initialization is as simple as possible, but allowing for direct manipulation of the KalmanFilter data if necessary. The data from the blob that is not corrected using the Kalman filter is directly copied from the best match, so there is no data loss.

MotionRecorder

This is a tool to obtain the motion trajectories of a TrackableObject. The MotionRecorder is fed with a TrackableObject's data on every camera frame. Two set of parameters define the blob parameters that activate and deactivate recording a motion. In practice the parameter chosen is velocity; i.e. setting a large velocity as a starting trigger and a low velocity as a stopping trigger. When idle and the conditions to start recording are met, the MotionRecorder is set in recording mode. It stores every BlobData fed to it. When recording and the conditions to stop are met, the MotionRecorder stops storing data, and awaits for the next trajectory.

HMM

A HMM defines a continuous hidden Markov model and the action primitives graph. It stores all the data necessary to work with the action primitives graph and HMM. For the HMM is stores the initial state probabilities, the transition probabilities and the array of states. Each state of the HMM is defined by a HMMState which holds all the information for a specific state of the HMM, that is, the mean and covariance of the distribution. It also defines what data is stored in each state of the HMM with an array of parameters. The observation probabilities are not stored but calculated using the probability distribution function for the Gaussian parameters defined in the state. The Forward, Backward, Viterbi, Baum-Welch and custom HMM training for the action primitives were implemented for this class.

The action primitives graph is defined as described in section 3.6.10 as a set of nodes and arcs. The HMM class provides amongst other functionality methods for: 1) Finding the optimal sequence of HMM states for an observation, 2) Using a sequence of HMM states identify the action primitives and 3) With a sequence of actions, validate a path in the graph of action primitives.

HMMTrainer

The HMMTrainer uses a set of observations to train a HMM and create the action primitives graph. After training is done, the HMM defined can evaluate an observation and return the likelihood. It implements all the steps required for finding the action primitives and building the actions graph as defined in Section 3.6.

GtoolkitSerializer

The purpose of this class is to provide the possibility of reusing the objects one they have been created, avoiding retraining and configuration. It saves each object type in a separate file, as an XML human-readable text file. The classes that can be saved and loaded are: Camera, TrackableObject, HMM and recordings as an array of BlobMotionData (BlobMotionData is a structure that has the original BlobData and a normalized copy of the data).

4.2.2 Action Primitives Editor

The second part of the software developed and which became the final tool to perform all the tests of the thesis is the Action Primitives Editor. This editor is

a user friendly application meant to provide all the functionality of the GToolkit library and add the easiness of a state of the art graphical user interface. The editor was developed using the commercial quality and cross-platform library Qt. In this section the most important features of the editor will be described.

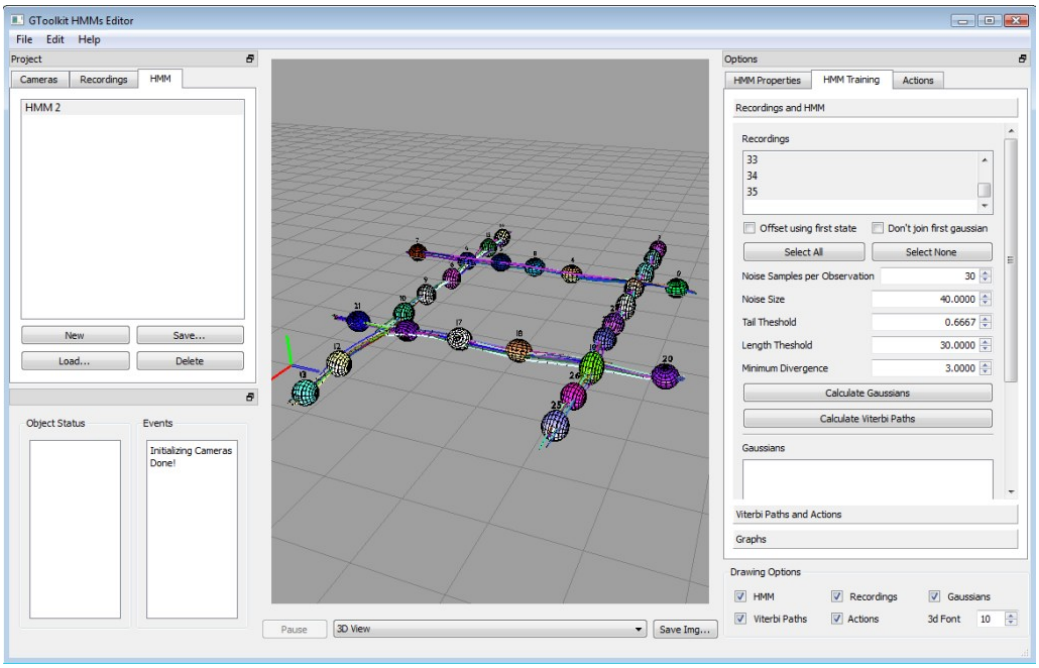


Figure 4.4: Screenshot of the Action Primitives Editor.

Camera Tools

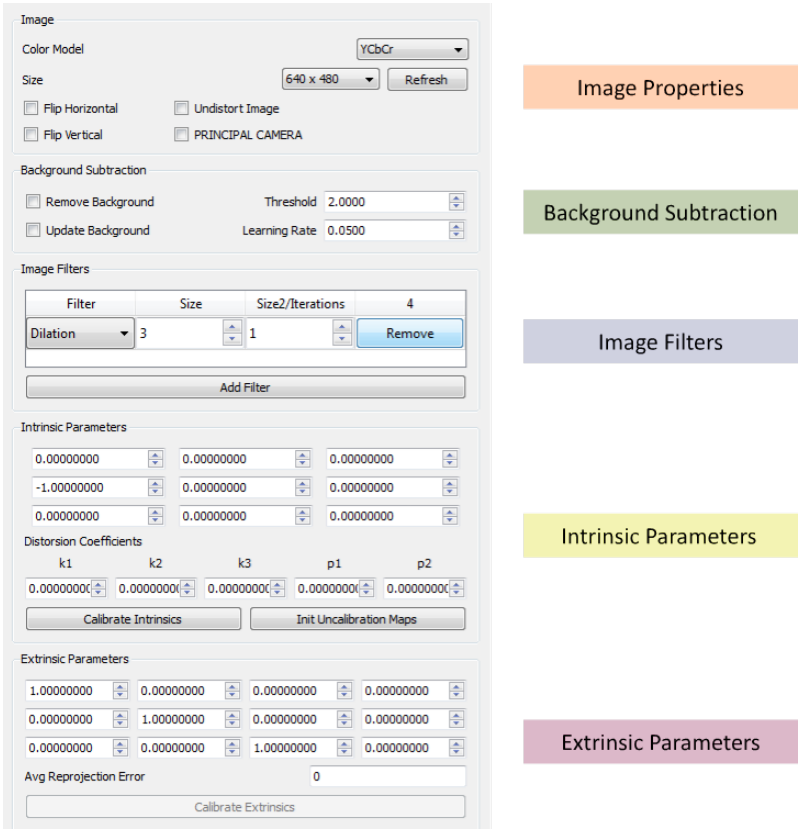


Figure 4.5: Camera tools in the Action Primitives Editor labeled for illustration purposes.

The figure shows the tools for configuring the cameras. The are five group boxes used to organize the tools by functionality: Image Properties, Image Filters, Background Subtraction, Intrinsic Parameters and Extrinsic parameters. Intrinsic and extrinsic camera calibration are in-built in the editor.

Trackable Object Tools

Figure 4.6: Different tabs of the trackable objects tools from the editor are shown in this figure.

In Figure 4.6 three tabs (one is excluded) with the tools for defining the properties of the trackable objects are shown. The tools include widgets for setting the color properties and the pixel classification technique to be applied on the image, the blob filtering methods and filters, and the Kalman filter configuration. To define the color properties, a sampling button allows to grab pixels from the live camera image, and it will calculate all the parameters for the different pixel classification techniques, as defined in Section 3.2.2, by means of the GToolkit library. As stated previously the configuration of a trackable object is done per-camera, allowing the flexibility required by the difference color temperature and quality of the images.

A last tab not shown in the figure allows to enable and define an auxiliary coordinate frame to localize the observations in real time. This is useful, as described in Section 3.6.4, to be able to obtain observations in the same frame of reference regardless of the camera positions. It also allows to obtain observations in a

more friendly orientation, as in the case of a planar surface being parallel to the world coordinates.

Recording Tools

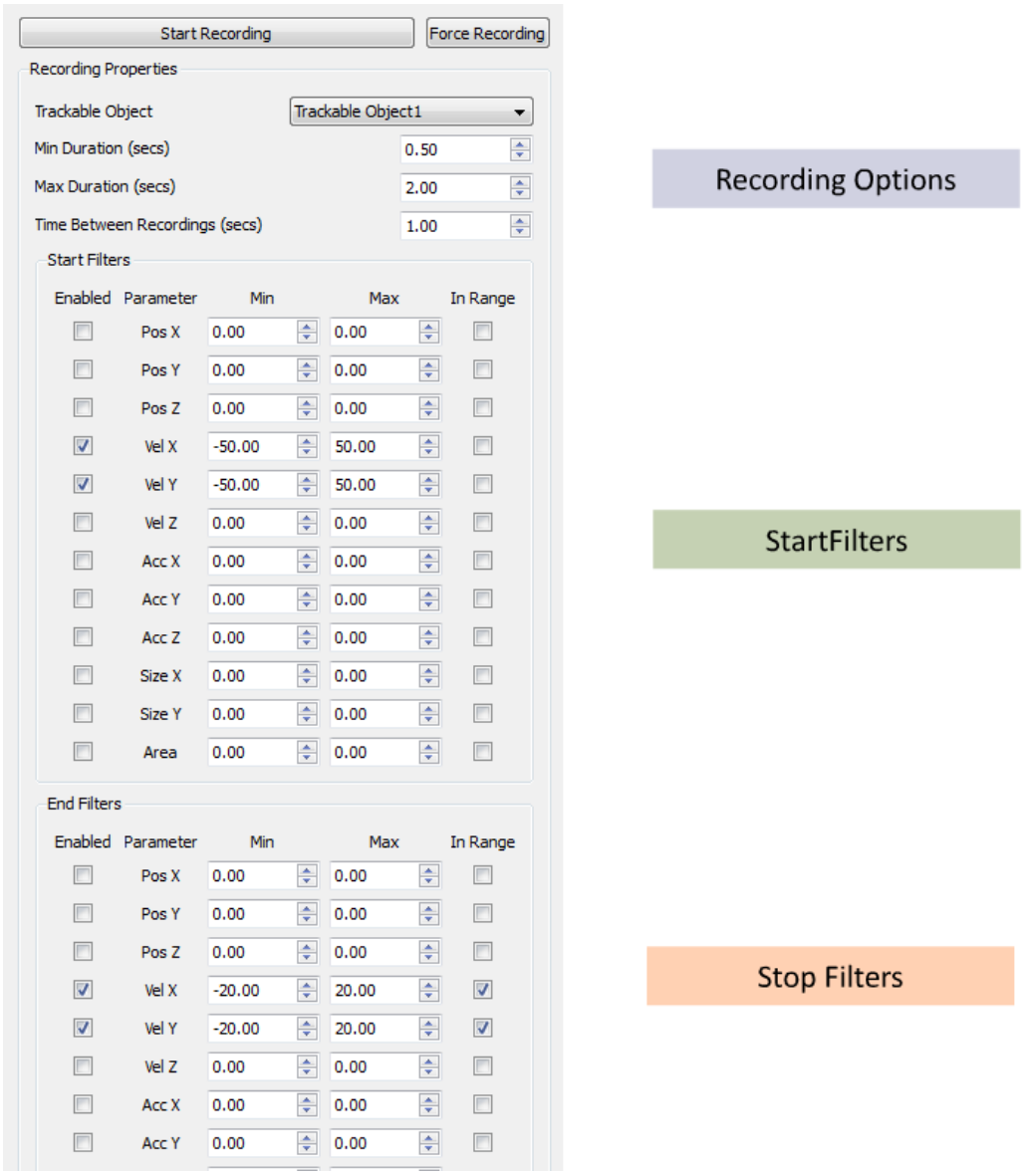


Figure 4.7: The image shows the UI for defining the properties of the recorder.

The recording tab show the filters defining the triggering parameters for recording trajectories. Any parameters of a trackable object can be used to start/stop the recording. Other options include the minimum and maximum time a recording can last to be considered valid and the time to wait between recordings, to allow the object being tracked to be placed in a desired position before starting a new recording. Figure 4.7 show this tool in the editor.

HMM and Actions Tools

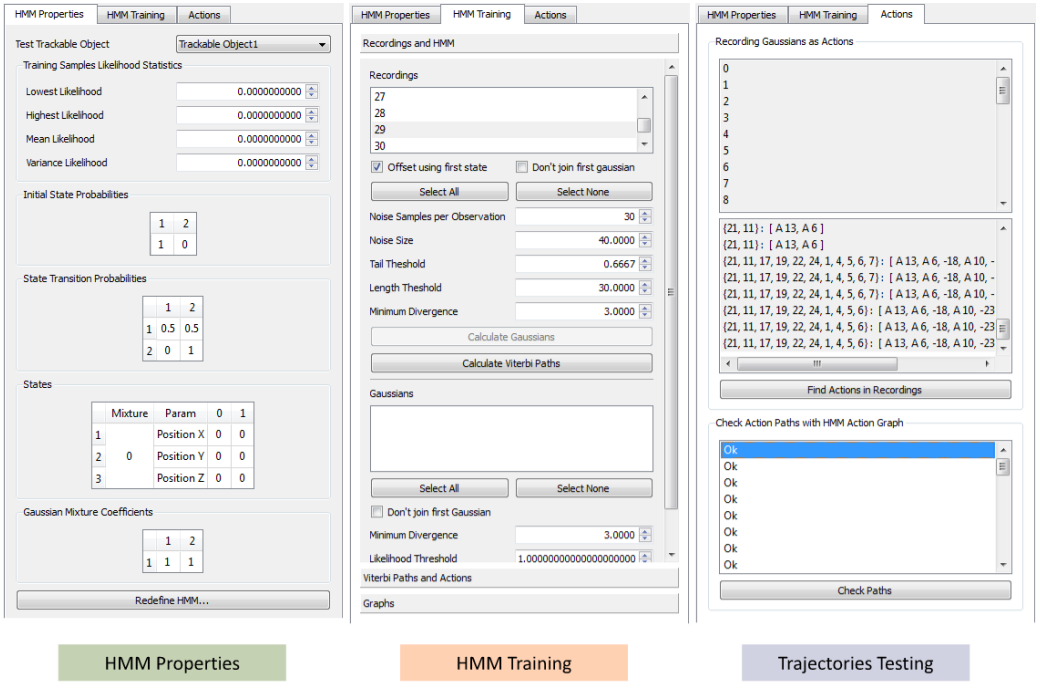


Figure 4.8: Tools for building and visualizing the HMM and action primitives graph.

The last tool briefly described here is the one used to create and train action primitives models from observations. In Figure 4.8 a screenshot of the tools is seen. In short, the tools allow to define the properties of the HMM as a combination of blob parameters, segment the trajectories as Gaussians, and train a HMM from these trajectories. Using the HMM and a set of trajectories the state-changes as described in Section are identified, which leads to the building of the action primitives graph. With the aid of the Dot tool of the Graphviz [**?**] package, the editor is capable of drawing the HMMs and action primitives graphs as images.

Chapter 5

Tests

This chapter describes the tests used to evaluate the performance of the system for the action primitives model building. Given the complexity of the system a component-wise testing is not possible in the given time. Therefore, the object tracking modules of the system are implicitly tested as they are preconditions to find the action primitives and define the action primitives graph.

5.1 Testing Scenario

To elaborate the tests a table was placed to perform the motions of an object whose trajectories were recorded. The object chosen was a homogeneously colored object of high contrast easily identifiable in the scene. Specifically round colored flat disks were used. A configuration of two USB cameras placed on a tripod each looking at the table with the object were used to track the position of the object in the three dimensional space using the functionality provided by the GToolkit library.

Figure 5.1: A subject performing the motion of an object over the testing table during a recording session.

All the data from the tests was collected using the Action Primitives Editor. Various subjects moved an object after receiving instructions and it's motions were recorded. In Figure 5.1 an image of a subject performing the motions is shown.

The steps for setting up the system were the following:

1. Start the two cameras in the editor. Select the color space to use, train the background model and add any image filters desired. The intrinsic and extrinsic parameters can be obtained using the dialogs for this in the editor and a calibration pattern.

2. Create a trackable object to be followed. The color filtering, blob filtering and tracking filtering parameters have to be defined using the tools in the editor.

3. Set the recorder properties. There are two modes for recording that can be set with the options in the recording tools. The first one is using the filters that define the starting and ending triggers of a recording, usually a starting and ending velocity zones. The second mode is set by not defining any filters, which triggers recording immediately and finally forcing the

creation of a recording pushing the “Force Recording” button. This last mode is good for rapid prototyping.

4. Create a HMM and action primitives model using the editor. Using a dialog the state parameters for the HMM can be defined. Using a set of recordings and by setting the parameters in the trainer, the HMM and action primitives graph are built. It is possible to draw an image of both the HMM and the action primitives graph by supplying the path to the Graphviz dot.exe application. In practice the parameters chosen are the positions in the X, Y and Z coordinates.

5. Test the data. Using some recordings it is possible to test them offline with the “Actions Evaluation” tab of the editor. It is also possible to evaluate observations online in the “Evaluation” tab of the editor, if the cameras are running.

Three tests were defined for the system. The first test is interested in motions that are defined in object space, as discussed in Section3.6.4. The second test is concerned with motions defined in world space, that is, the starting point is not always the same. The third test is focused on the grammar representation of the observations.

5.2 Object Space Test

5.2.1 Purpose of Test

This test was performed to evaluate the HMM and action primitives graph generated for a set of observations that are known to be defined in object space. A practical scenario of such case is moving a chess piece on a chessboard. For this scenario the starting point is not important, only the trajectory.

5.2.2 Test Method

The motions in the test consisted of movements in the four cardinal directions: north, south, east, west, as performed over the table, with three different trajectory lengths. The lengths are defined by the squares on the table covering either one, two or three squares of distance. An auxiliary coordinate frame was used to locate all the observations during the recording phase on the XZ plane of the world space. Figure 5.2 shows the different actions performed in different colors. Each motion trajectory was shifted to the origin of the world, in other words localized, by subtracting the first observation in the trajectory to all the other observations in the trajectory.

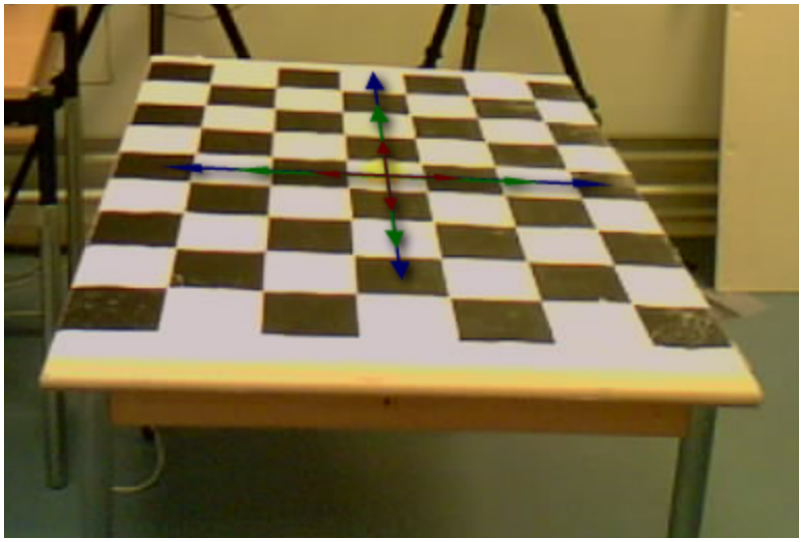


Figure 5.2: The motions in object space, in three different lengths. Each motion starts at the center point.

In this scenario the expected result is to obtain a model with one common starting Gaussian state for all the observations. It is also expected that most of the Gaussians and action primitives will be common for most of the observations. The training and testing data consists of three sets of observations, each one with four repetitions of each motion. That is a total of 48 observations per set, with a total of 144 observations for the entire test. Three different subjects participated in the recording of the motions. One set of motions from one subject was used for training, and the other two were used to test the model. Table 5.1 shows the number of samples taken.

Motion	Samples #1	Samples #2	Samples #3	Total
	4	4	4	12
	4	4	4	12
	4	4	4	12
	4	4	4	12
	4	4	4	12
	4	4	4	12
	4	4	4	12
	4	4	4	12
	4	4	4	12
	4	4	4	12
	4	4	4	12
	4	4	4	12
All	48	48	48	144

Table 5.1: Number of samples taken for each trajectory type.

5.2.3 Test Results

Trajectories As Gaussians

Using the editor the trajectories for the observations were covered with Gaussians as can be seen in Figure 5.3. The parameters used for this step were the following:

- Number of Noise Samples: 150.
- Noise size: 30.
- Tail threshold: 0.6667.
- Length Threshold: 15.
- Minimum divergence: 4.
- Starting Gaussians were prevented from joining.

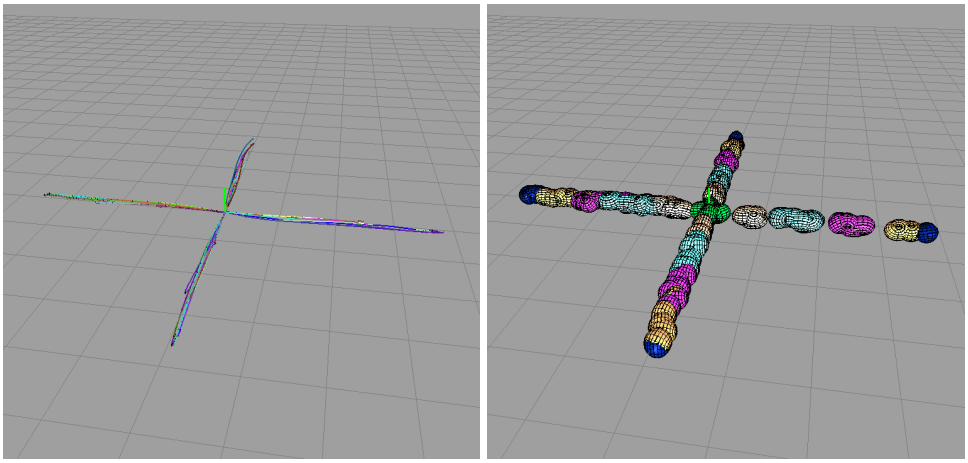


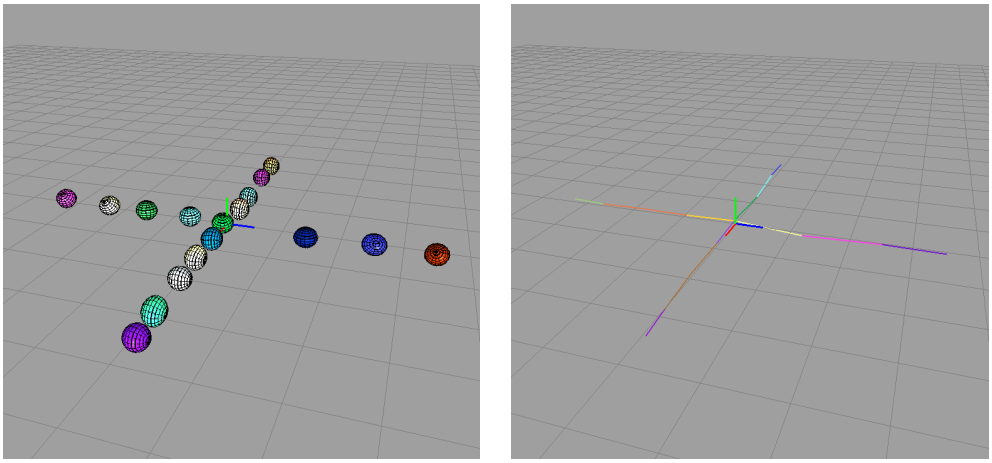
Figure 5.3: Gaussians (right) obtained for the recordings (left) in object space.

A total of 48 trajectories from the training sample set were covered by Gaussians. Each one is kept separate.

HMM Training and Action Primitives Graph Building

With the Gaussians estimated the next steps are to build the joined HMM, find the action primitives and construct the action primitives graph. For this it is necessary to use both the raw observations and the Gaussians estimated for each observation. The parameters used in the editor for building the model were:

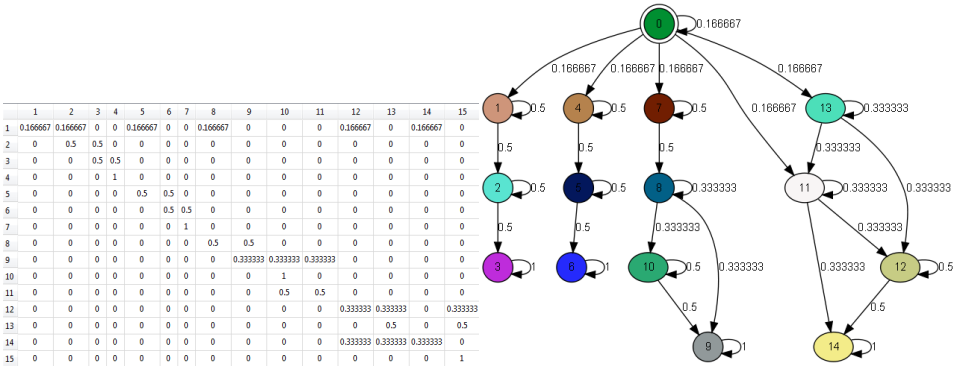
- Minimum divergence: 4
- Likelihood Threshold: 1



(a) (b)

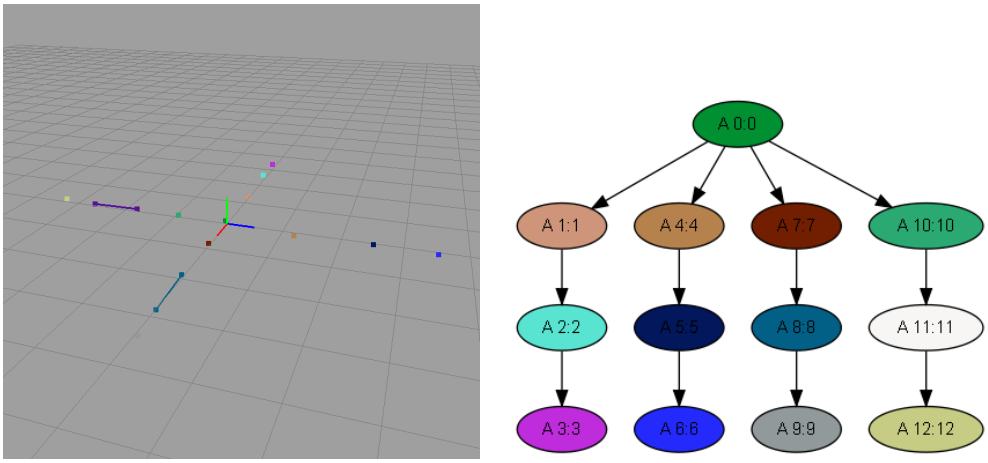
Figure 5.4: (a) Gaussian states of the HMM for the object space motions scenario. (b) Viterbi paths found for the observations used for training. Each colored line is a different path.

Figure 5.4 shows the HMM built using the Gaussians covering the observations. Using the Viterbi algorithm the most likely paths in the HMM for the training observations were identified. From the image it is possible at this step to identify the three distinct lengths of the observations. The HMM has a total of 15 states. The transition matrix of the trained HMM can be seen in figure 5.5. It can be seen that the first state can transition to the adjacent four states and to itself as expected.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0.166667	0.166667	0	0	0.166667	0	0	0.166667	0	0	0	0.166667	0	0.166667	0
2	0	0.5	0.5	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0.5	0.5	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0.5	0.5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0.5	0.5	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0.5	0.5	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0.333333	0.333333	0.333333	0	0	0	0
10	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0.5	0.5	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0.333333	0.333333	0	0.333333
13	0	0	0	0	0	0	0	0	0	0	0	0	0.5	0	0.5
14	0	0	0	0	0	0	0	0	0	0	0	0.333333	0.333333	0.333333	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Figure 5.5: Transition matrix and graph representation of the trained HMM in object space.



(a) (b)

Figure 5.6: (a) Actions primitives as seen in the editor. (b) Actions primitives graph of the resulting model. Each colored node in image (b) is a primitive action, similar to the points and lines in image (a).

The graph in Figure 5.6 shows the possible action primitives extracted from the state-change sequences. Each node in the graph represents in this specific case a different length trajectory. It is noticeable that after going through the first action primitive, a trajectory as a sequence of Gaussian states can only follow a single path ending at one of three possible actions. This ending action defines the type of observation that was tested. This model is coherent with the three different lengths used in training.

Model Evaluation

The previous model with the action primitives and grammar as a directed graph was trained using one of the three sets of trajectories. The two other trajectories were used to evaluate how well the model performs when classifying them as correct or incorrect trajectories. In Table 5.2 the results of the evaluation validating the input trajectories is shown.

Motion	#Samples	#Correct	#Incorrect	Validation Rate %
→	8	8	0	100%
→	8	8	0	100%
→	8	8	0	100%
↑	8	8	0	100%
↑	8	8	0	100%
↑	8	8	0	100%
←	8	8	0	100%
←	8	8	0	100%
←	8	8	0	100%
↓	8	8	0	100%
↓	8	4	4	50%
↓	8	8	0	100%
Total	96	92	4	95.83%

Table 5.2: Results of evaluating two sets of training data with the trained model for the object space scenario.

5.2.4 Conclusion

The results shows that the system is capable of building a HMM and action primitives model that correctly validates and elaborates an abstract representation of the observation trajectories defined in object space. The validation rate was satisfactory with an average of 95.83% as shown in Table 5.2.

Only for one type of motion in one of the two sets of observations the validation was wrong. After retracing the steps in validation it was found that the action primitive at the end of the graph consisted of many Gaussians. The Gaussian sequences identified in the trajectories tested were valid but the observation was not long enough to include all the necessary Gaussians to completely define the last action primitive. Therefore these last segments couldn't be identified as an action and the trajectories were classified as invalid. In the practical scenario this shows the case when the motion of the object didn't finish close to the endings of the training data, but it did follow the correct path. It is therefore a matter of application to choose whether this can be defined as invalid or valid. In some cases, it can be enough to arrive at the action, in others it may be chosen

to restrict the validation so the identified sequence has all the Gaussians that define the last action.

In summary, it can be concluded that the system can efficiently define and validate motion trajectories defined in object space.

5.3 World Space Test

5.3.1 Purpose of Test

The purpose of this test is to evaluate the functionality of the framework for a set of observations known to define a system with actions in world space. In practical terms, the validation of motion trajectories in world space is dependent on the starting place of the observation.

5.3.2 Test Method

This test is a simulation of the intersection of two streets, each one with two lanes in opposing directions. In the simulation a car starts from one of the streets and moves forward transitioning to another street which doesn't violate the street's direction, and stopping at the end of such street. The following figure describes the scene:

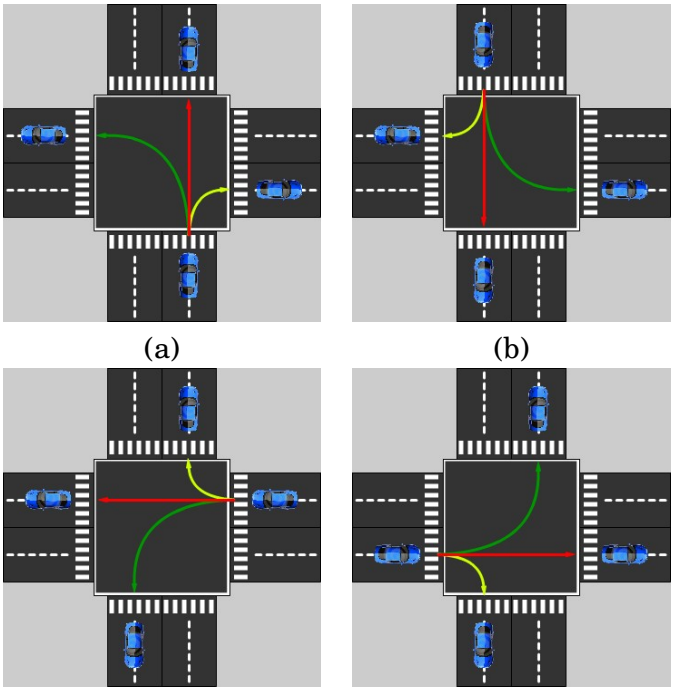


Figure 5.7: Illustrations (a)- (d) show the street intersection scenario with the possible transitions of a car from one street to another shown with arrows.

In this scenario it is expected that the model will contain action primitives for the streets, re-used by many trajectories. The HMM and action primitives graph should display the possible connections between streets in the intersection.

Three sets of observations of each trajectory were performed by three subjects moving a colored disk on the test table. A total of 36 observations per-set were done by each individual, with a total number of samples of 108. One set of observations was used for training and the other two were used to test the generated model.

	Samples #1	Samples #2	Samples #3	Total
	3	3	3	9
	3	3	3	9
	3	3	3	9
	3	3	3	9
	3	3	3	9
	3	3	3	9
	3	3	3	9
	3	3	3	9
	3	3	3	9
	3	3	3	9
	3	3	3	9
	3	3	3	9
All	36	36	36	108

Table 5.3: Number of samples taken for each trajectory type.

5.3.3 Test Results

Trajectories As Gaussians

Using the editor the trajectories for the observations were covered by Gaussians, as can be seen in Figure 5.8. The parameters used for this step were the following:

- Number of Noise Samples: 150
- Noise size: 30
- Tail threshold: 0.6667
- Length Threshold: 15
- Minimum divergence: 4
- Initial Gaussians were not prevented from joining other Gaussians..

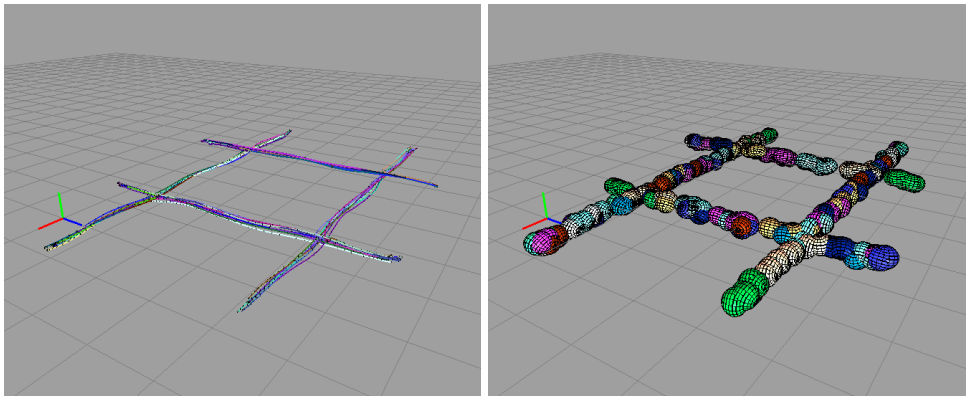
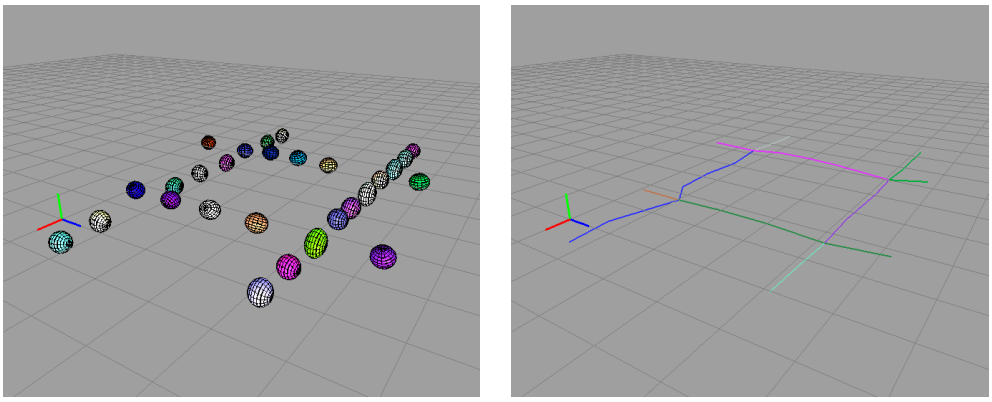


Figure 5.8: Gaussians (right) obtained for the recordings (left) in world space.

HMM Training and Action Primitives Graph Building

Similar to the previous test in Section 5.2, the HMM, action primitives and action primitives grammar are obtained with the estimated Gaussians. The parameters used in the editor for building the model were:

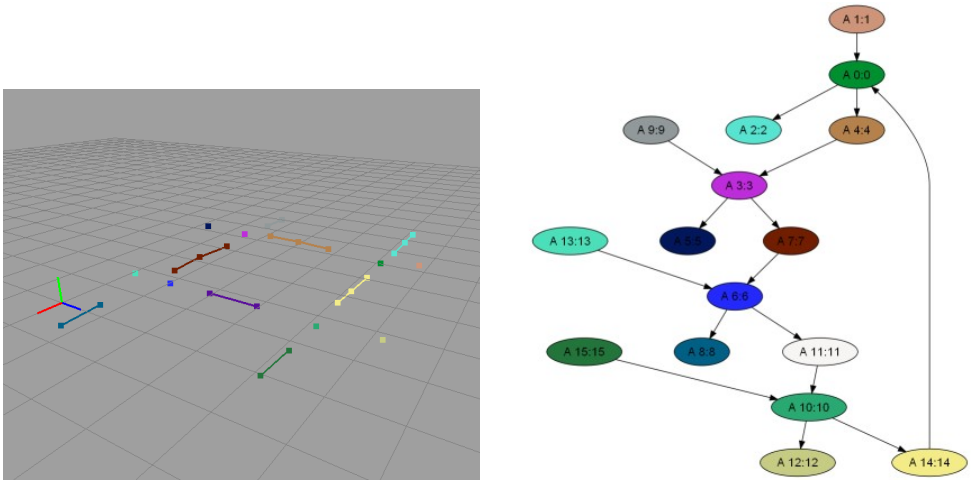
- Minimum divergence: 4
- Likelihood Threshold: 1



(a) (b)

Figure 5.9: (a) Gaussian states of the HMM for the streets scenario. (b) Viterbi paths found for the observations used in the training of the HMM.

A total of 28 states were identified in the final HMM from the trajectories as Gaussians. The transition matrix is not shown here because of it's large size, but it's included in Appendix E.



(a) (b)

Figure 5.10: a) Actions primitives as seen in the editor. b) Actions primitives graph of the street intersection scenario.

The graph in Figure 5.10 shows the connections between the streets defined as action primitives. It is to be noted that the graph is optimal as the number of action primitives is the same as the number of possible choices in every street, adding only one action primitive for the intersection itself.

Model Evaluation

The previous model with the action primitives and grammar as a directed graph was trained using only one of the three sets of trajectories. The two other trajectories were used to evaluate how well the model performs when classifying them as correct or incorrect trajectories. In Table 5.4 the validation of the trajectories is shown.

	#Samples	#Correct	#Incorrect	Validation Rate%
	6	6	0	100%
	6	6	0	100%
	6	6	0	100%
	6	6	0	100%
	6	6	0	100%
	6	6	0	100%
	6	6	0	100%
	6	6	0	100%
	6	6	0	100%
	6	6	0	100%
	6	6	0	100%
	6	6	0	100%
Total	72	72	0	100%

Table 5.4: Results of evaluating two sets of training data with the trained model for the world space scenario.

5.3.4 Conclusion

In this test it was shown that the system is capable of correctly defining a model from a set of observations located in world space. The model was tested with trajectories different from the ones used in training and achieved a validation rate of 100%.

5.4 Model Grammar Test

5.4.1 Purpose of Test

To check the consistency of the grammar as defined by one of the previous models using trajectories different from the ones used in the training data.

5.4.2 Test Method

The test will be divided in two parts. The first one is interested in showing that trajectories which use the same paths but don't follow the action primitives

grammar, will be classified as erroneous. The second part will use new trajectories which follow the grammar rules but were not defined in the training data, and test them as valid or not.

Incorrect Trajectories

This scenario uses the same test setup as in Section 5.3, but adding eight new trajectories using some of the same paths. The incorrect trajectories can be seen in Figure 5.11.

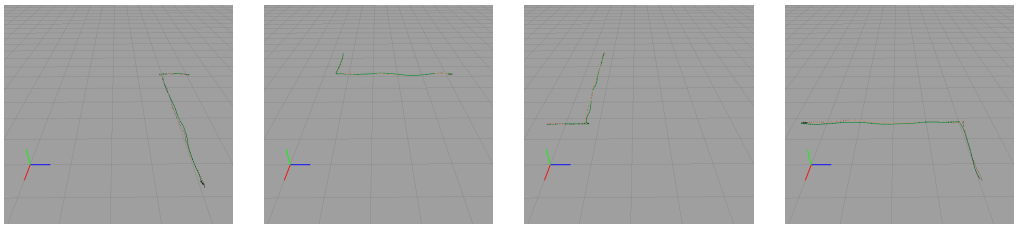


Figure 5.11: Incorrect trajectories passing through the portions of same paths with some parts being in the wrong direction of the streets.

New Correct Trajectories

As in the previous test eight new trajectories were defined, this time choosing action primitives sequences not defined in the initial training data but that should follow the grammar of the action primitives model. The new trajectories can be seen in Figure 5.12.

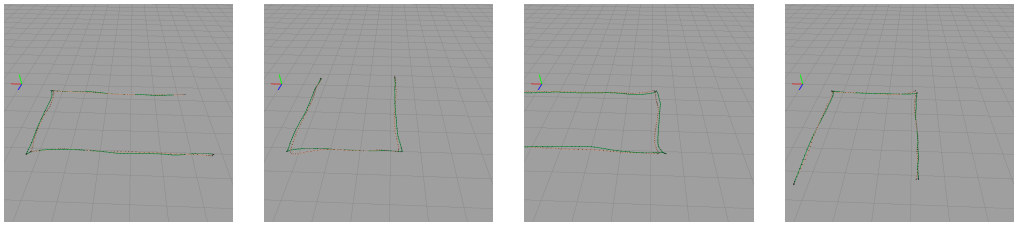


Figure 5.12: New trajectories passing thought valid action primitive sequences.

The trajectories were tested using the Action Primitives Editor in the same way as it was described in Section 3.6.11.

5.4.3 Test Results

The results of the validity of the trajectories can be seen in Tables 5.5 and 5.6.

Incorrect Trajectories Results

Sample#	Validity
1	Negative
2	Negative
3	Negative
4	Negative
5	Negative
6	Negative
7	Negative
8	Negative

Table 5.5: Table with the validation given by the model to the trajectories of incorrect motions.

New Correct Trajectories Results

Sample#	Validity
1	Positive
2	Positive
3	Positive
4	Positive
5	Positive
6	Positive
7	Positive
8	Positive

Table 5.6: Table with the validation given by the model of the new motions trajectories which follow the rules of the grammar.

5.4.4 Conclusion

With these tests it was shown that the system is able to correctly classify as invalid, trajectories that use the same physical path as the one defined by the Gaussians in the actions primitives of the model but whose estimated state sequences are in different order than the ones composing the action primitives.

On the other hand, trajectories which use the same action primitives and follow the grammar defined in the model, but which were initially not defined during training are tested as valid. These last motions test case is an interesting one as the paths validated can be used to learn or infer new ways to perform certain activities. An example could be to find an optimal path in a more complex scenario for completing an action by adding weights to each action primitive.

Chapter 6

Discussion

6.1 Contributions

The following sections describe the main contributions of this work.

6.1.1 Modeling Actions Primitives

Through the work in this thesis the theory and implementation of a model of action primitives for the representation of motion trajectories was presented. The framework developed is capable of representing many observations of different trajectories in a single model. The model allows for the identification, recognition and prediction of action primitives in observations. The recognition of trajectories is done in two levels, first in a low level using hidden Markov models, and second, using grammar defined in an action primitives graph. This provides more efficiency and better performance.

6.1.2 GToolkit

A programming library was developed exceeding the requirements of the objectives of the thesis. It can capture from several devices, apply background subtraction, image filtering, kalman filtering, build HMMs and action primitives models. It also provides a serializer for saving and loading data. The library is modular and cross-platform, programmed in C++. It is not limited to the action primitives problem, but can also be used as a tool in computer vision applications for other purposes.

6.1.3 Action Primitives Editor

A stand alone application providing a user friendly interface to do, in one single place all the steps required in this project. From image capturing and camera calibration to the creation and evaluation of action primitives models. All the requirements are presented in one single place. The application is also cross-platform.

6.2 Conclusions

In this thesis the objectives presented were: to detect and track objects in real-time while recording their features in 3d space, to define a stochastic model of a set of many actions from several samples, to identify sub-actions in the model as action primitives and to obtain the grammar of the model for validation and identification of new trajectories. The previous sections show that these objectives were accomplished with satisfactory results. A system for modeling a group of related motion trajectories as a single hidden Markov model and a graph of action primitives was described and developed. The results show that it is successful in defining models and that the models represent the motions described by the observations in two layers of abstraction. The defined action primitives learning model, the GToolkit library and the Action Primitives Editor constitute together a complete system for unsupervised learning of motion trajectories represented as HMMs and sequences of action primitives.

Appendix A

Color Conversion

A.1 RGB to Normalized RGB Conversion

For a color in the RGB space, with values ranging from 0 to 255, the conversion to the Normalized RGB color space is shown in the following C code:

Algorithm A.1 RGB to Normalize RGB Conversion

```
// r,g,b are the red, green, and blue channels of the color
// nr, ng, nb are the normalizes rgb colors
float sum = r+g+b;
if( sum = 0 )
nr =g =b =0;
else {
   nr = r/sum;
   ng = g/sum;
   nb = 1-nr-ng; // To avoid precision problems
}
```

A.2 RGB to YCbCr Conversion

For a color in the RGB space, with values ranging from 0 to 255, the conversion to the YCbCr color space is shown in the following C code:

Algorithm A.2 RGB to YCbCr Conversion

```
// r,g,b are the red, green, and blue channels of the color
Y = 16 + (1/256) * (65.738*r + 129.057*g + 25.064*b);
Cb = 128 + (1/256) * (-37.945*r - 74.494*g + 112.439*b);
Cr =128 + (1/256) * (112.439*r - 94.154*g - 18.285*b);
```

A.3 RGB to HSV Conversion

For a color in the RGB space, with values ranging from 0 to 1, the conversion to the Hue, Saturation, Value color space is shown in the following C code:

Algorithm A.3 RGB to HSV Conversion

```
// r,g,b are the red, green, and blue channels of the color
// h, s, v are the HSV colors
v = minc = r;
if( v < g )
   v = g;
if( v < b )
   v = b;
if( minc > g )
   minc = g;
if( minc > b )
   minc = b;
diff = v - minc;
s = diff/(float)(fabs(v) + FLT_EPSILON);
diff = (float)(60./(diff + FLT_EPSILON));
if( v == r )
   h = (g - b)*diff;
else
   if( v == g )
      h = (b - r)*diff + 120.f;
   else
      h = (r - g)*diff + 240.f;
if( h < 0 )
   h += 360.f;
```

A.4 RGB to HSL Conversion

For a color in the RGB space, with values ranging from 0 to 1, the conversion to the Hue, Saturation, Luminance color space is shown in the following C code:

Algorithm A.4 RGB to HSL Conversion

```
// r,g,b are the red, green, and blue channels of the color
// h, s, l are the HSV colors
float maxc = minc = r;
if( maxc < g )
   maxc = g;
if( maxc < b )
   maxc = b;
if( minc > g )
   minc = g;
if( minc > b )
   minc = b;
float diff = maxc - minc;
l = (maxc + minc)*0.5f;
if( diff > FLT_EPSILON ){
   s = l < 0.5f ?  diff/(maxc + minc) :  diff/(2 - maxc - minc);
   diff = 60.f/diff;
   if( maxc == r )
      h = (g - b)*diff;
   else
      if( maxc == g )
         h = (b - r)*diff + 120.f;
      else
         h = (r - g)*diff + 240.f;
   if( h < 0.f )
      h += 360.f;
}
```

A.5 RGB to Hunter LAB Conversion

For a color in the RGB space, with values ranging from 0 to 255, the conversion to the Hunter LAB color space is shown in the following C code:

Algorithm A.5 RGB to HSV Conversion

```
// r,g,b are the red, green, and blue channels of the color
// L, a, b are the Hunter Lab colors
if(r > 0.04045f){
   r = pow((r + 0.055f)*0.9478f, 2.4f);
}
else {
   r = r / 12.92f;
}
if( g > 0.04045){
   g = pow((g + 0.055f)*0.9478f, 2.4f);
}
else{
   g = g / 12.92f;
}
if(b > 0.04045f){
   b = pow((b + 0.055f)*0.9478f, 2.4f);
}
else{
   b = b / 12.92f;
}
//Observer.  = 2°, Illuminant = D65
float x = (r * 0.4124f + g * 0.3576f + b * 0.1805f) * 100.0f;
float y = (r * 0.2126f + g * 0.7152f + b * 0.0722f) * 100.0f;
float z = (r * 0.0193f + g * 0.1192f + b * 0.9505f) * 100.0f;
// Hunter Lab
sqrty = sqrt(y);
L = 10.0f * sqrty;
a = 17.5f * (((1.02f*x) - y) / sqrty);
b = 7.0f * ((y - (0.847f*z)) / sqrty);
```

Appendix B

Kalman Filter

The discrete Kalman Filter is a two-step process, prediction and correction. The notation and formulas here are found in [**?**]

$\hat{x}_k^-$ State prediction at time k

P_k^- Error Covariance prediction at time k

A System Model

H Measurement Model

Z_k Measurement

Q System Noise

R Measurement Noise

K_k Kalman Gain at time k

$\hat{x}_k$ Corrected State at time k

P_k Corrected Error Covariance at time k

Prediction

$$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_{k-1} \tag{B.1}$$

$$P_k^- = AP_{k-1}A^T + Q \tag{B.2}$$

Update

$$K_k = P_k^- H^T (H P_k^- H^T + R)^{-1} \tag{B.3}$$

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - H\hat{x}_k^-) \tag{B.4}$$

$$P_k = (I - K_k H) P_k^- \tag{B.5}$$

Appendix C

Viterbi Algorithm

The following is the Viterbi algorithm as defined in [**?**] used for finding the most likely state sequence which produced an observation, where

T Number of observations in observation O.

N Number of states in the HMM.

π_i Initial probability for state s_i

$b_i(O_j)$ Observation probability for state s_i and observation O_j.

δ, ψ Temporary arrays

q^* Array with the best sequence

1. Initialization

$$\delta_1(i) = \pi_i b_i(O_1), \quad 1 \leqslant i \leqslant N \tag{C.1}$$

$$\psi_1(i) = 0, \quad 1 \leqslant i \leqslant N \tag{C.2}$$

2. Recursion

$$\delta_t(j) = \max_{1 \leqslant i \leqslant N} \left[\delta_{t-1}(i) a_{ij}\right] b_j(O_t), \quad \begin{matrix} 2 \leqslant t \leqslant T \\ 1 \leqslant j \leqslant N \end{matrix} \tag{C.3}$$

$$\psi_t(j) = \arg \max_{1 \leqslant i \leqslant N} \left[\delta_{t-1}(i) a_{ij}\right], \quad \begin{matrix} 2 \leqslant t \leqslant T \\ 1 \leqslant j \leqslant N \end{matrix} \tag{C.4}$$

3. Termination

$$P^* = \max_{1 \leqslant i \leqslant N} \left[\delta_T(i)\right] \tag{C.5}$$

$$q_T^* = \arg \max_{1 \leqslant i \leqslant N} \left[\delta_T(i)\right] \tag{C.6}$$

4. Path recovery

$$q_t^* = \psi_{t+1}(q_{t+1}^*), \quad t = T-1, T-2, ..., 1. \tag{C.7}$$

Appendix D

Longest Common Substring

The following is the algorithm used for finding the longest common substring (LCS) between two strings.

Algorithm D.1 Longest Common Substring solution for two strings

```
function lcs(str1, str2)
   z = 0
   n = length(str1)
   m = length(str2)
   table = array[n,m]
   for i in range 0,...,n
      for j in range 0,...,m
         if i == 0 or j == 0
            table[i, j] = 0
         else
            if str1[i-1] == str2[j-1]
               table[i, j] = table[i-1, j-1] + 1
            else
               table[i, j] = max(table[i-1, j], table[i, j-1])
```

Appendix E

Transition Matrix for World Space Test

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
1	0.5	0.5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0.25	0.25	0	0.25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.25
3	0	0	0.5	0.5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0.5	0.5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0.5	0.5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0.333333	0.333333	0.333333	0	0	0	0	0	0	0.333333	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0.25	0.25	0.25	0	0	0	0	0	0.25	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0.5	0.5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0.25	0.25	0.25	0	0	0	0.25	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0.333333	0.333333	0.333333	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0.5	0.5	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.5	0.5	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0.333333	0	0.333333	0	0	0	0	0	0	0.333333	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.5	0.5	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.333333	0.333333	0.333333	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.333333	0.333333	0	0.333333	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0	0.5	0	0	0	0	0	0	0	0	0	0.5	0	0	0	0	0
22	0	0.333333	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.333333	0.333333	0	0	0
23	0	0.5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.5	0	0	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.333333	0	0	0	0	0.333333	0.333333	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.333333	0.333333	0	0	0	0	0.333333	0
26	0	0	0.333333	0.333333	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.333333