Simplified Computer Interaction Using Mixed Reality

Jean-Paul Balabanian

December 22, 2004

Master Degree Thesis
University of Bergen,
Department of Informatics

Abstract

This thesis describes a system for mixing reality, as captured by a camera, with a virtual 3-dimensional world. A system to recognize and track a square pattern of markers is created in order to obtain the extrinsic parameters of the camera. The parameters are used for rotating and translating the virtual world to align it with the pattern of markers in the image. The result is a system where a user can interact with a virtual world. A camera can be move freely around the pattern or the user can manipulate the marker square while the virtual world is blended with the real world and viewed on a screen or with a HMD (Head Mounted Display).

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

In this thesis we want to create an application that enables a user to interact with a virtual world in a natural and intuitive way. By using a camera to capture images of markers we want to project a virtual object into the image so that it looks like the virtual object belongs in the scene. This field of Computer Science is called Mixed Reality (MR).

1.1 Mixed Reality, a Definition

A definition of MR was found in [19]. That work starts by defining a *Reality-Virtuality Continuum*, where one end is reality, where nothing is computer generated, and the other end is virtuality, where everything is modelled. Moving from reality towards virtuality we come into a field called *augmented reality* (AR). AR is described as:

> "any case in which an otherwise real environment is *augmented* by means of virtual (computer graphic) objects"

Moving from virtuality towards reality, another field is reached called *augmented virtuality* (AV). AV is the opposite of AR where a virtual world is augmented with real objects.

The area on the continuum that starts with AR and ends with AV describes a field called *mixed reality*. This means that all AR and AV applications are MR applications.

The reason for not explicitly labeling this thesis as an AR work is that we don't feel that the scene is enhanced with the virtual addition. For example an AR application could be x-ray or Magneto Resonance Imaging (MRI) data superimposed on the body of a patient. Another example is an excavator operator being able to see where cables and pipes lie underground. These examples illustrate an AR application giving the user useful information about the current environment. In our case the virtual addition does not necessarily give the user any useful information.

1.2 Motivation

The motivation to create a MR based application originates from the way human-machine interaction is presently done. A human being that wants to interact with a computer, must learn to use a keyboard, a mouse, and the intricacies of the software. We want to create a complete system to view and interact with 3D objects in a more natural way. By natural, we mean basic human ways to interact with real-world objects, like hand gestures, speech, head movements, and physically walking around an object. These interaction methods are more intuitive than learning to use a keyboard, a mouse, and a graphical user interface on a computer.

One of the benefits with the MR techniques is that a user is able to visually see the 3D object in the same environment as the user occupies. It is also possible to superimpose extra information on top of real objects so that multiple physical information sources are unnecessary.

Another benefit is that collaboration is easier, because everyone can see the same information in the same space, and all users are able to communicate about the same information while seeing each other. This keeps the focus on the information and the other people, and not on the equipment necessary to create the information.

1.3 Objectives

The goal is to create an application that enables the user to view and interact with a 3D model.

- The interaction should be accomplished by moving the head, the hands, or to physically change position. The model should be integrated into the scene, this means that size, position, and perspective of the virtual object should correspond with the real world as seen from the user.
- The system should track the user by using the same device as for capturing the real world. The reason for this is that we want to use low cost equipment and avoid the need for additional tracking devices. The tracking should work in real time.
- The software should be built with Java, because of cross platform independence and easier porting to smaller devices like mobile phones with camera.
- The system should support both a computer screen and a HMD for presentation of the results.
- Get an in-depth understanding of how to compose a complete MR system.

1.4 Previous and Current Work

Many mixed reality applications are available and currently developed. The following are examples of this:

- One of the applications is a development library called ARToolKit [1]. This is a tool for creating AR applications. ARToolKit is primarily being developed by Dr. Hirokazu Kato of Osaka University, Japan, and is being supported by the Human Interface Technology Laboratory (HIT Lab.) at the University of Washington, and the HIT Lab NZ at the University of Canterbury, New Zealand. This tool uses something similar to what we develop in this thesis. The ToolKit finds a simple black square and uses this square to find position and pose. The square also contains a marker that indicates to the software what virtual representation that marker should have. The final result is similar to what we have developed but the differences are in the implementation of marker recognition and camera calibration, and in the marker design.
- Another project that is similar to the ARToolKit is [22]. This work uses a cube with different colored sides to estimate pose and lighting.
- Other MR applications are real world scenes with rendered objects using image based lighting. Captured footage of people is rendered in a virtual world where shadows and reflections of the real people are created so that it looks like the people are in the virtual world. An example of this type of work can be found at [4].
- The Playstation 2 game console has an accessory called Eyetoy which is a camera connected to the console. This camera records the user and the images are used as a background images. The images are frame by frame checked for movement that the user is providing. If the movement is in the correct areas the user interacts with game, for example in a fighting game you can punch the virtual enemy off the screen by slapping him with your hand [25].
- Heads Up Displays (HUD) are available for pilots, motorcycle helmets, and cars. This gives the user important information without having to look down at the instrument panels of the vehicle. For instance motorcycle helmet can give the user information about speed and rounds per minute on the view screen of the helmet. An important point is that the information is focused at a distance so that the user can read the information with as little effort as possible.

1.5 Proposed Architecture

By analyzing the objectives of the mixed reality application, we have found that the system needs to be composed of four main parts. These parts are:

1. **Capturing** Acquiring images of the real world.
2. **Recognizing markers** Finding markers in the image.
3. **Camera calibration** Finding the relative position and orientation of the camera based on the location of the markers and their relative positions.
4. **Visualization** Using conventional visualization techniques; pose the virtual object and blend the computer generated object with the image of the real world.

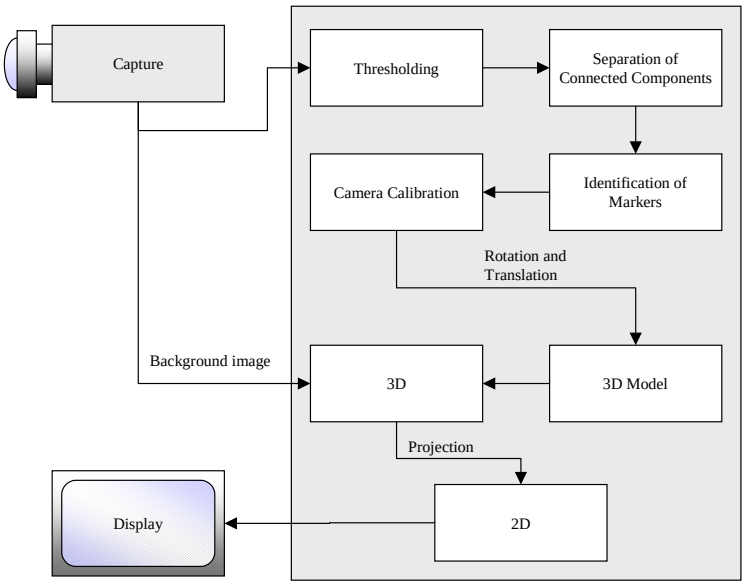


Figure 1.1: System setup and software flow.

Investigating further, we find that each part is composed of smaller parts. The following details are not implementation details, but a schematic overview of the necessary steps for each part. Figure 1.1 is an illustration of all of the necessary steps.

1. **Capturing** images of the real world will in this case be accomplished with a camera and transferred to the software using a suitable library

2. **Recognizing markers** is a three step process that consists of:

   (a) **Thresholding** Removing data from an image that does not have the correct color while highlighting the data with the correct color.

   (b) **Separation of Connected Components** Finding and grouping together data, from the image, that belong together by proximity and color.

   (c) **Identification of Markers** Markers are identified by color and localization inside other markers.

3. **Camera calibration** takes the position of the markers and finds a camera's relative position and orientation.

4. **Visualization** is done by blending a 3D model with the captured image. Using the camera's position and orientation a virtual object is posed and positioned so that the object appears to be in the scene. The resulting image is displayed on a computer screen or a HMD.

1.6 Chapter Overview

The following is an overview of the chapters in this thesis. This list describes what each chapter entails, as well as references to the different processes depicted in Figure 1.1.

Chapter 2 describes the process of capturing an image, thresholding the image, finding connected components, identifying the components as different markers, and sorting the markers. This is the Thresholding, Separation of Connected Components, and Identification of Markers steps in Figure 1.1.

Chapter 3 walks through some of the background theory needed for the next chapter.

Chapter 4 describes each step needed to find the rotation matrix and the translation matrix necessary for setting up a correct camera position for the visualization process. This is the Camera Calibration step in Figure 1.1.

Chapter 5 shows how OpenGL is used to merge the real world image and the computer generated image. This is the part where a 3D model is rotated and translated and merged with the background image through projection in Figure 1.1.

Chapter 6 demonstrates the complete setup and shows some good results, but also points out problems that occur.

1.7 Hardware and Software Development Tools Decisions

The hardware was chosen based on availability and cost. The software libraries were chosen based on performance and compatibility with Java. As we will see later, these choices have limitations that impact the final developed system.

Hardware list

- AthlonXP3000+ 2.17GHz, 1 gigabyte RAM
- ADSTech USB Turbo 2.0 WebCam
- Logitech QuickCam Pro 4000
- i-visor HMD DH-4400VPD (Personal Display Systems, Inc.)

Software libraries

- Capture capability is achieved with Java Media Framework (JMF) [26]. This library provides real time access to a capturing device
- Graphics library. The industry standard OpenGL is provided through Lightweight Java Gaming Library (LWJGL) [3].

Figure 1.2: The cameras. On the left: Logitech. On the right: ADSTech

Figure 1.3: HMD.

1.8 Notation and Definitions

This section contains notations, definitions, a mathematical proof for intersecting lines is given, and general knowledge about pixels and color models.

The knowledge of how a pixel stores and represents color information will later be useful for thresholding a color image. The intersection of lines will be used for sorting markers and finding vanishing points.

1.8.1 Notation

The following notation is used in this thesis.

Points A point in 3D is written as:

$$P = \{x, y, z\}$$

A point in 2D is written as:

$$p = \{x, y\}$$

Vector A vector named u is written as $\vec{u}$.

Parallel Vectors A vector $\vec{u}$ is parallel to a vector $\vec{v}$ is written as $\vec{u} \parallel \vec{v}$.

Lines A line L can in 2D and 3D be described with a point P, a parameter t, and a direction vector $\vec{d}$ as:

$$L = P + t\vec{d}$$

A line can also be described with two points $\{p_1, p_2\}$ and a parameter t as:

$$L = p_1(1 - t) + p_2 t$$

1.8.2 General Math

Vectors A vector $\vec{u}$ in two dimensions is defined as:

$$\vec{u} = (u_x, u_y) = u_x\vec{i} + u_y\vec{j}$$

A vector $\vec{u}$ in three dimensions is defined as:

$$\vec{u} = (u_x, u_y, u_z) = u_x\vec{i} + u_y\vec{j} + u_z\vec{k}$$

A vector $\vec{u}$ in n-dimensions is defined as:

$$\vec{u} = (u_1, u_2, \ldots, u_n)$$

Dot Product The dot product of two vectors $\vec{u}$ and $\vec{v}$ in n-dimensions is defined as:

$$(\vec{u} \cdot \vec{v}) = u_1v_1 + u_2v_2 + \cdots + u_nv_n$$

Also known as inner product or scalar product.

Cross Product The cross product of two vectors $\vec{u}$ and $\vec{v}$ in three dimensions is defined as:

$$\vec{w} = (\vec{u} \times \vec{v}) = (u_yv_z - v_yu_z)\vec{i} - (u_xv_z - v_xu_z)\vec{j} + (u_xv_y - v_xu_y)\vec{k}$$

The analogy in two dimensions is a scalar and is defined as:

$$w = (\vec{u} \times \vec{v}) = (u_xv_y - v_xu_y)$$

The value of w can be used to find out what side $\vec{v}$ is on $\vec{u}$:

If $w < 0$	$\vec{v}$ is on the right side of $\vec{u}$
If $w > 0$	$\vec{v}$ is on the left side of $\vec{u}$
If $w = 0$	$\vec{v}$ is on $\vec{u}$

Orthogonal A vector $\vec{u}$ is orthogonal to another vector $\vec{v}$ if the dot product of the two vectors equals 0.

$$(\vec{u} \cdot \vec{v}) = 0$$

Orthonormal A set of vectors are orthonormal if they are orthogonal to each other and are of unit length.

1.8.3 Image Format and Pixels

A digital image obtained from a camera is a 2-dimensional (2D) representation of a 3D world from a certain point of view using discrete values. An image is composed of small square components called pixels. A pixel is the smallest fragment of an image. Each pixel represents a single color that can span the entire visual range of the human eye. A pixel can also represent data from x-ray, infrared, or other imaging techniques like MRI.

The amount of data contained in a pixel depends on the image type. A greyscale image typically contains 8 bits of information resulting in 256 different shades of gray from black to white, as seen in Figure 1.4.

Figure 1.4: 8-bit greyscale.

Figure 1.5: 8-bit scales of red, green and blue.

A color image is often divided into 3 different channels. Each channel represents the intensity of one primary color with 8 bits of information. The number of color combinations is $256 \times 256 \times 256 = 16\,777\,216$. Figure 1.5 illustrates the different colors available for each channel.

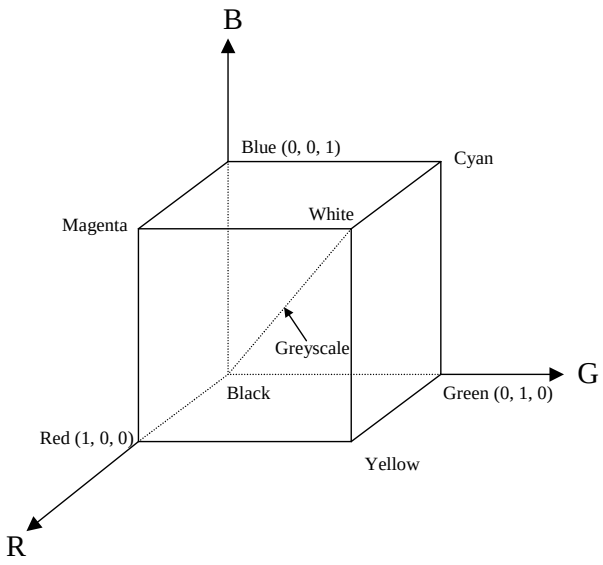
1.8.4 The RGB Color Space.

A cathode ray tube (CRT) monitor uses three rays of colored light that are blended to color each pixel on the monitor. The higher the intensity of a ray the brighter the color, as shown in Figure 1.5. When all three rays are at full intensity, the color is white.

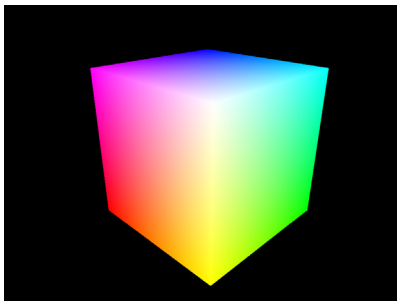
There are different light models available in computer imaging for blending colors. The one used in this thesis is called the RGB-model. The RGB model is the one that most closely represents the way light operates. The model-name reflects the primary colors used[1].

Figure 1.6 illustrates the blending of light. The RGB color cube in Figure 1.6(a) demonstrates the different hues of color available when mixing colors using the three axes; red, green and blue. When equal amounts of red, green, and blue are blended together, the result is the greyscale, which is represented as the diagonal in the color cube. Figure 1.6(b) and Figure 1.6(c) shows the colors generated on the sides of the cube.

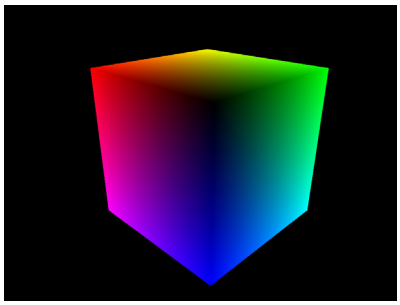
[1] RGB stands for Red, Green, and Blue



(a) Model



(b) RGB 24-bit color cube



(c) RGB 24-bit color cube backside

Figure 1.6: RGB Color model.

1.8.5 Intersecting Lines in a Plane

This section presents equations to find an intersection between two lines on the same plane. Later, we will use these equations to find intersections inside squares and to calculate the vanishing point for the perspective of an image.

Two lines that are not parallel will intersect at a point. The following equations locate an intersection if it exists. The line L_1 is defined by the two points $\{p_1, p_2\}$ and the line L_2 is defined by $\{p_3, p_4\}$. These two lines are described by the following two equations where s and t are real-number parameters:

$$L_1 = p_1(1-t) + p_2 t \tag{1.1}$$

$$L_2 = p_3(1-s) + p_4 s \tag{1.2}$$

To find the intersection of these two lines we set $L_1 = L_2$ and get

$$p_1(1-t) + p_2 t = p_3(1-s) + p_4 s$$

Then by splitting it into its separate components x and y, we get

$$x_1(1-t) + x_2 t = x_3(1-s) + x_4 s \tag{1.3}$$

$$y_1(1-t) + y_2 t = y_3(1-s) + y_4 s \tag{1.4}$$

solving Equation 1.3 for t gives

$$t = \frac{x_3(1-s) + x_4 s - x_1}{(x_2 - x_1)} \tag{1.5}$$

inserting t from Equation 1.5 into Equation 1.4 and solving gives

$$s = \frac{(y_3 - y_1)(x_2 - x_1) - (x_3 - x_1)(y_2 - y_1)}{(x_4 - x_3)(y_2 - y_1) - (y_4 - y_3)(x_2 - x_1)} \tag{1.6}$$

inserting the value of s into Equation 1.2 returns the intersection coordinate of the two lines. If the lines are parallel, Equation 1.6 will result in a division by zero. This can be avoided by first checking the denominator of Equation 1.6 before dividing in the algorithm.

Chapter 2

Marker Acquisition

In this chapter, techniques for capturing and analyzing images are investigated. The road from acquiring images with a camera, thresholding, finding connected components and then separating markers from non-markers is thoroughly described. Some pitfalls and some cases where the algorithm fails are pointed out.

2.1 The Physical Setup

In this section, the setup and appearance of the markers are described, and the different sets of markers tested for this thesis are listed. The marker pattern is required to have a set of important properties:

- all markers lie in the same plane.
- the markers are placed in a square, this results in two pairs of parallel lines which are perpendicular to each other.
- one marker has a different color so that the orientation of the square can be found.
- the size of the markers must be proportional with the intended operating distance from the markers, so that each marker is captured by a reasonable number of pixels.

The last property ensures that the pattern should be easy to recognize in the captured image. Some of these properties are the requirements of the camera calibration presented later. Based on these required properties, different types of markers has been tested. Two criteria used for developing the markers are visibility under different lighting conditions and the colors used must be easy to identify.

A simple setup using a sheet of paper printed by a color printer was the first trial, and it worked as long as any possible markers in the background were not present to confuse the recognition. This setup is depicted in Figure 2.1.

Another trial was done using light emitting diodes (LED) inside styrofoam hemispheres. The intention was to make the markers more visible but this was unsuccessful due to the lights being too intense for the camera. The light-intensity was so high that the sensors in the camera became oversaturated.

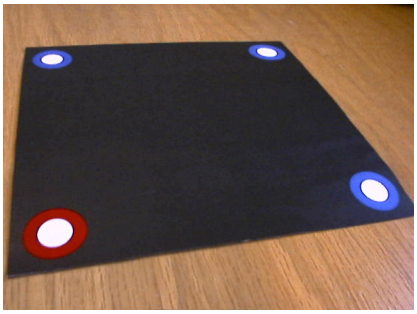
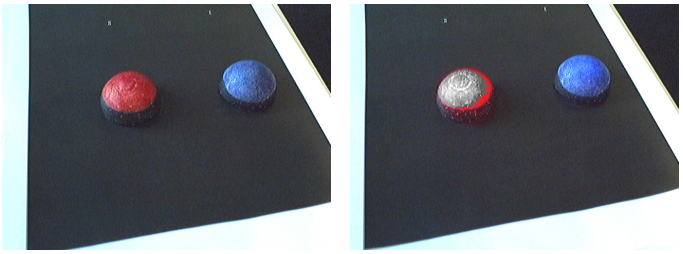


Figure 2.1: The first markersquare pattern.

This resulted in grey pixels. Figure 2.2(a) shows the styrofoam hemispheres and Figure 2.2(b) shows the effects of the high intensity of the LEDs.



(a) Styrofoam hemispheres with LEDs turned off.

(b) Styrofoam hemispheres with LEDs turned on.

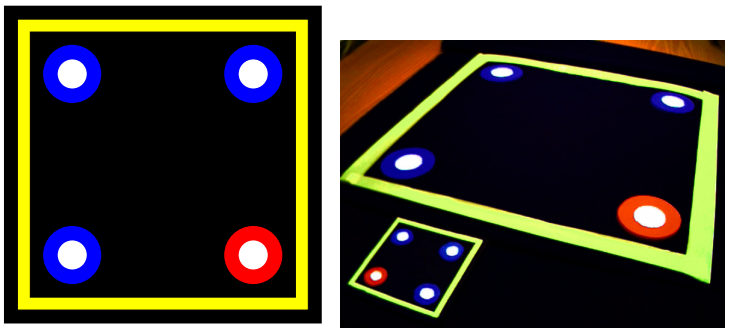
Figure 2.2: Styrofoam hemispheres and LEDs.

Finally, a setup was done using painted wood discs and a yellow border on cloth. Another smaller setup was done using a board of tree-fiber painted with acrylic paint to resemble the marker square pattern. The yellow border made it easier to group and identify a set of potential markers. The pattern is illustrated in Figure 2.3(a). The large pattern has a useable range of up to 2 meters while the small works at approximately 70 centimeters.

2.2 Camera Capture

Capturing images from a camera is the first step in acquiring the location of the markers. In this thesis, the Java package Java Media Framework has been used to retrieve images from a webcamera [26]. The webcameras used are a Logitech Quickcam 4000 PRO connected with USB 1.1 and a ADSTech USB Turbo 2.0 Webcam connected with USB 2.0. The resolution used during capturing is 640x480 and the frame rate is set to 15 frames per second. The image size of 640x480 is the maximum resolution of the cameras. The human eye is insensitive to changes faster than about 24 frames per second, so the choice of a frame rate of 15 is a good compromise between quality and computer resource usage.

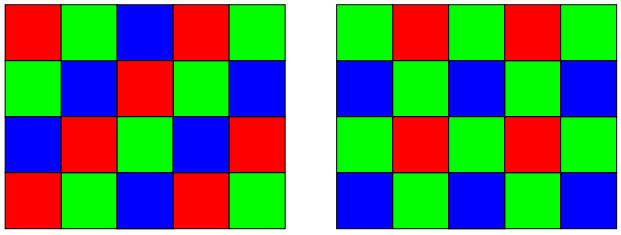
The sensor array of the cameras chosen does not have three separate 640x480 red, green, and blue sensor chips. Neither is the array composed of 640x480 sen-



(a) The final marker pattern. (b) Image of two different sets of markers.

Figure 2.3: The final marker pattern and an image of the markers.

sors all capable of simultaneously sensing red, green, and blue light intensities. Usually there is a single chip with a little more than 640x480 sensors in total. For example, the ADSTech camera has an effective sensor array of 644x484. Each sensor detects the intensity of one of the primary colors (red, green, and blue), and the sensors are placed in a mosaic pattern. Examples of sensor configurations can be seen in Figure 2.4. The image is then reconstructed from the mosaic pattern in the camera hardware. A discussion of the technique can be found in [9]. The reconstruction phase explains some of the reduced quality of images from webcamers. After reconstruction, the image is transferred to the computer via the USB channel as a 640x480x3 bytes image.



(a) Diagonal pattern (b) Baye pattern

Figure 2.4: Different sensor patterns.

2.2.1 Image Quality and USB Speed

USB stands for Universal Serial Bus and is maintained by USB Implementers Forum Inc. [16]. USB is a communication port used to transmit information between computers and devices. Two different standards exist for this kind of communication, namely USB 1.1 and USB 2.0. As mentioned in Section 2.2, both cameras use USB, but different standards. Both specifications are briefly explained here to show the difference.

The USB 1.1 bus is capable of transmitting a maximum of 12 megabits/second which equals 1.5 megabytes/second. The camera generates 640x480x3x15 =

13 824 000 bytes/second. So, before the images are sent over the USB-bus, each image must be compressed to reduce bandwidth usage and make it possible to send at least 15 images per second. The type of compression used is JPEG. The compression reduces the image quality. Artifacts from high JPEG compression can be read about in [14].

The USB 2.0 bus is capable of transmitting a maximum of 480 megabits/second which equals 60 megabytes/second. The camera generates 13 824 000 bytes/second. Due to the high speed of the USB, there is no need to compress the data before it is transferred to the computer. [1]

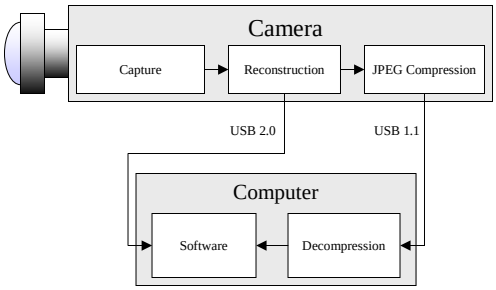


Figure 2.5: The image capture process.

Figure 2.5 illustrates where the compression and decompression of captured data happens. Due to the high bandwidth of USB 2.0, the compression and decompression step is unnecessary.

2.2.2 Lighting and Capture Quality

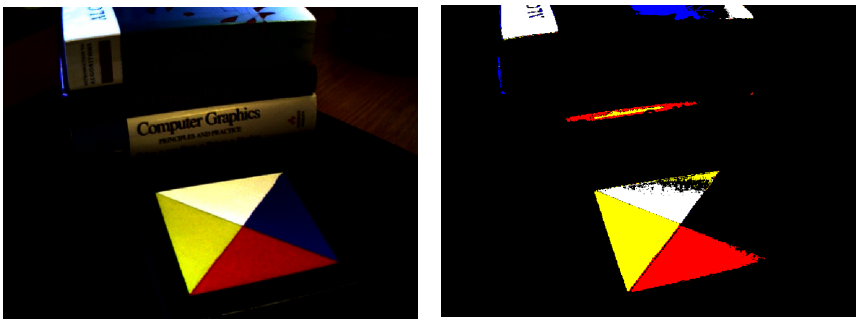
The cameras representation of color under different lighting conditions may also have an impact on the image quality. It is important that the light-conditions in the room is sufficiently bright so that colors are presented clearly. The best results are achieved when the light-source is diffuse, as on overcast days.

When the lighting is insufficient, the colors gets too dark to be recognized. This can be observed in Figure 2.6(a) where a calibration pattern, presented later, is captured under poor light-conditions.

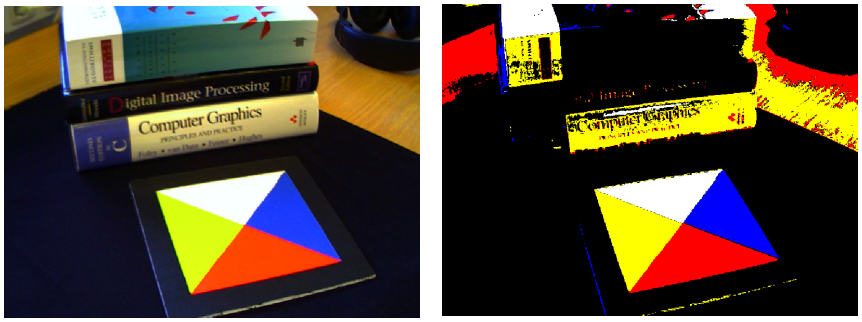
Too much light is also a problem: many colors turn too bright or too white to be recognized as their real color, and instead they are recognized as white or grey. This behaviour can be observed in Figure 2.6(e) and is called highlighting. As we will see later, white areas are important to the recognition phase, so it is important to avoid highlighting as much as possible.

Ideally the colors should look like in Figure 2.6(c) from any viewpoint that the camera is exposed to. The lighting problem is solved for most light-conditions by doing a manual color calibration of the camera before using the system.

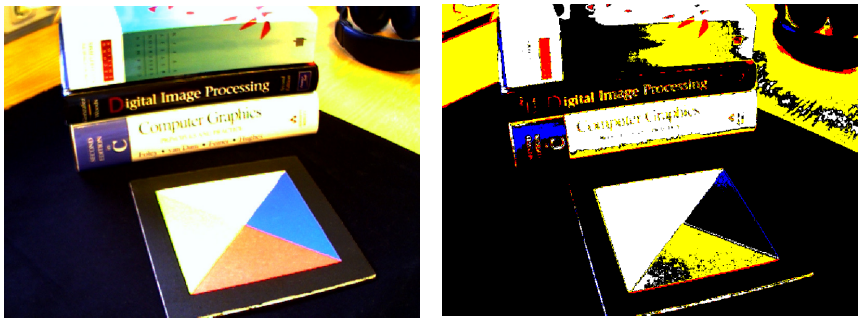
[1] Because of poor implementation of camera drivers for the ADSTech camera the computer is using up to 60% of its resources while capturing, this significantly decreases the performance of the final application.



(a) Too dark (b) After Thresholding



(c) Optimal (d) After Thresholding

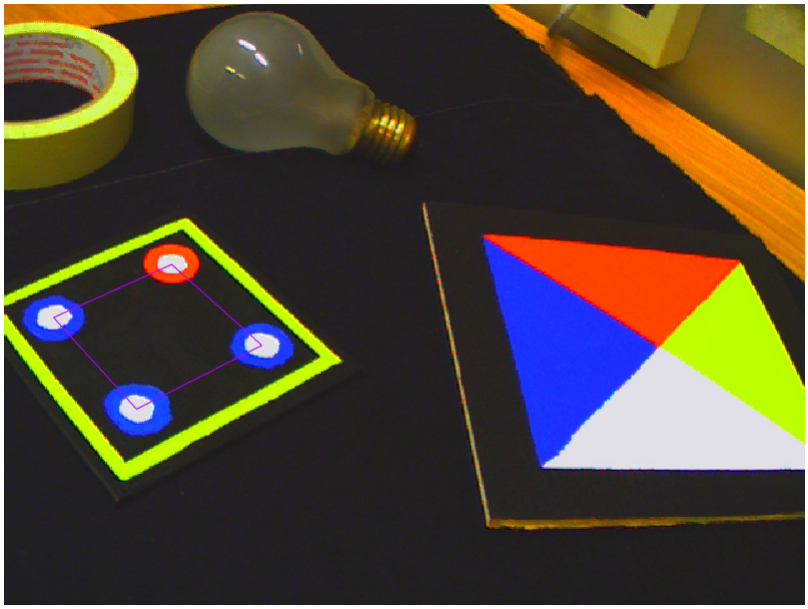


(e) Too bright (f) After Thresholding

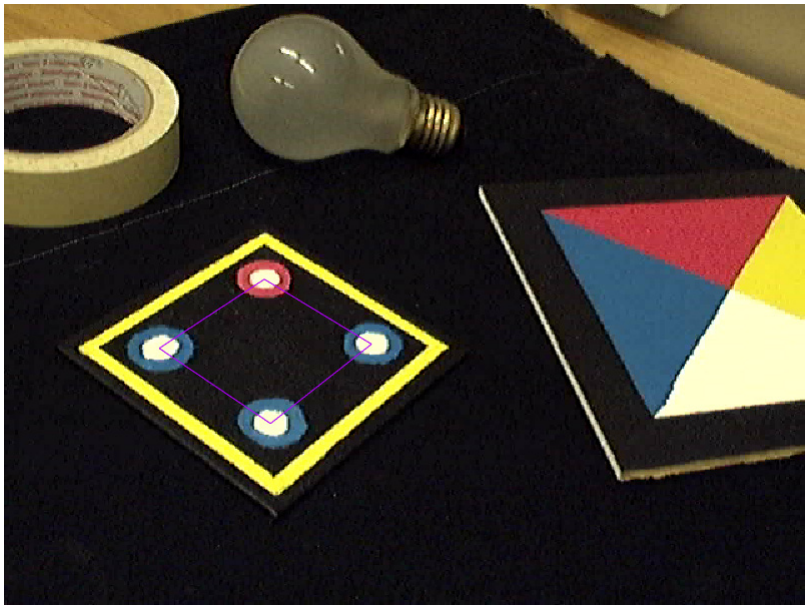
Figure 2.6: Lighting conditions and results of thresholding.

2.2.3 Camera Quality Difference

The two cameras used has huge differences in image quality. Figure 2.7(a) shows an image captured with the ADSTech camera. Figure 2.7(b) shows an image captured with the Logitech camera from an almost identical position. The images were capture using the same lighting conditions and with camera properties settings set to values that enables the application to function properly. The purple square indicates success in locating the markers.



(a) ADSTech



(b) Logitech

Figure 2.7: The difference in quality of the two cameras used.

2.3 Thresholding Captured Images

For this thesis it is necessary to acquire the location of markers in an image. In order to achieve this a technique called *thresholding* is used. Thresholding splits a wide range input signal into two or more distinct values. Thresholding of an

image results in highlighting of areas that have a particular color and removal of uninteresting parts of an image. A more thorough explanation of the topic can be found in [14]. The following function shows the general case where the resulting data are binary values:

$$f(x) = \begin{cases} 0 & \text{if } x < threshold \\ 1 & otherwise \end{cases}$$

A color image contains three channels of data, one for each color component. So to be able to threshold a color image, we have to separate the channels and check for each channel if the value is above or below a predetermined value.

When thresholding an image, good threshold values are necessary. The values in this thesis have been found empirically. Values used in the functions 2.1, 2.2, 2.3 and 2.4 have been used to find white, red, blue and yellow components in an image.

The empirically found values have been selected so that a range of hues represent that color. To find a white pixel the values have been adjusted so that the upper 37.5% of the greyscale represents white.

Due to quality differences between cameras these values are not static and must in some cases be adjusted. The values presented here are adapted to the Logitech camera. For example the ADSTech camera has better contrast and more vivid colors so all the thresholds can be increased.

$$isWhite(r,g,b) = \begin{cases} true & \text{if } r > 160 \text{ and } g > 160 \text{ and } b > 160 \\ false & otherwise \end{cases} \tag{2.1}$$

$$isRed(r,g,b) = \begin{cases} true & \text{if } r > 127 \text{ and } g < 127 \text{ and } b < 127 \\ false & otherwise \end{cases} \tag{2.2}$$

$$isBlue(r,g,b) = \begin{cases} true & \text{if } r < 127 \text{ and } g < 127 \text{ and } b > 127 \\ false & otherwise \end{cases} \tag{2.3}$$

$$isYellow(r,g,b) = \begin{cases} true & \text{if } r > 127 \text{ and } g > 127 \text{ and } b < 127 \\ false & otherwise \end{cases} \tag{2.4}$$

The colors that are not caught with these functions are set as black. To threshold an entire image Algorithm 1, see Appendix A, separates a full color image into five different values; white, red, blue, yellow, and black.

Figure 2.6(d) shows the result of thresholding Figure 2.6(c) using Algorithm 1.

2.3.1 Color Calibrating the Camera

To increase the quality of the thresholding, it is necessary to color calibrate the camera. A color plate with the colors needed for thresholding is used for calibrating, the design of the color plate used in this thesis can be viewed in Figure 2.8. Special care with the lighting conditions during the calibration is necessary, and

should represent the lighting conditions when in use. Figures 2.6(b), 2.6(d) and 2.6(f) show the thresholding under different lighting conditions. Figure 2.6(d) shows the result when the camera has been correctly calibrated. The calibration itself is done by manually adjusting camera properties such as brightness, contrast, hue, gamma and exposure. The properties available differ between cameras.

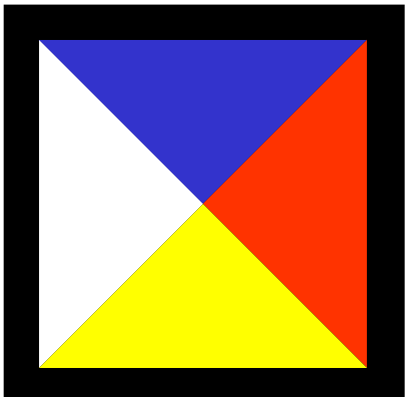


Figure 2.8: Calibration pattern.

2.4 Connected Components

After thresholding, each pixel is now marked as either red, white, blue, yellow or black. We need to group together pixels of the same color that are neighbors. These groups are called connected components. The connected components will be used in order to find potential markers.

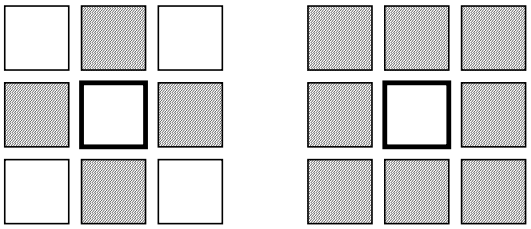
2.4.1 Connected Components Definition

A connected component is a group of pixels G such that for every pair of pixels g_i and g_j in G there exists a chain of pixels $\{g_i, \ldots, g_j\}$ that are all of the same color, and every 2 pixels adjacent in the sequence are neighbors. A connected component is also maximal, meaning that there does not exist a chain of pixels that includes pixels which are not in the group.

2.4.2 Neighborhood of a Pixel

The definition of the neighborhood of a pixel results in different connected components. A pixel has eight possible neighbors. Four of the neighbors are placed at the sides and the last four are placed diagonally at the corners. The neighborhood of a pixel is a mask that describes what neighboring pixels are included in the neighborhood. Two different neighborhoods are illustrated in Figure 2.9. As can be seen in the illustration a 4-way neighborhood contains only the adjacent pixels on the edges of a pixel, while an 8-way neighborhood also include the diagonals. The neighborhood type used in this thesis to find connected

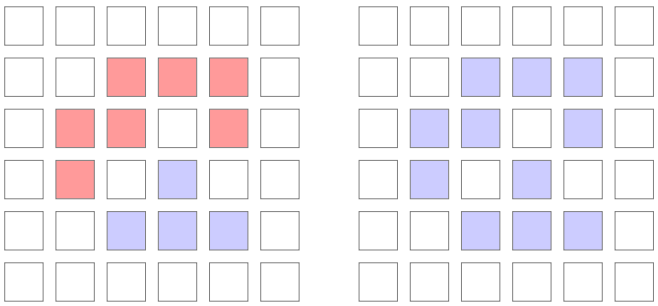
components is an 8-way neighborhood. Different neighborhood mask results in different connected components.



(a) 4-way neighborhood (b) 8-way neighborhood

Figure 2.9: The neighborhood of a pixel.

The result of using different neighborhoods can be viewed in Figure 2.10. In Figure 2.10(a) a 4-way neighborhood result in two different components because the neighborhood does not include diagonal neighbors. Using a 8-way neighborhood results in a single component as shown in Figure 2.10(b).



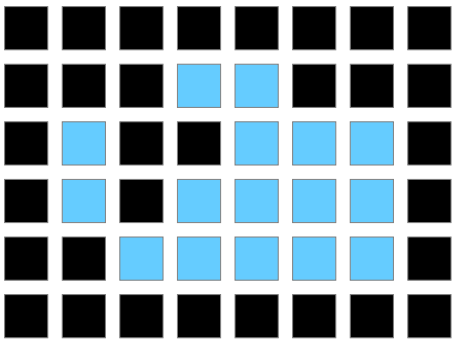
(a) Using a 4-way neighborhood results in 2 components. (b) Using a 8-way neighborhood results in a single component.

Figure 2.10: Connected Components with different neighborhoods.

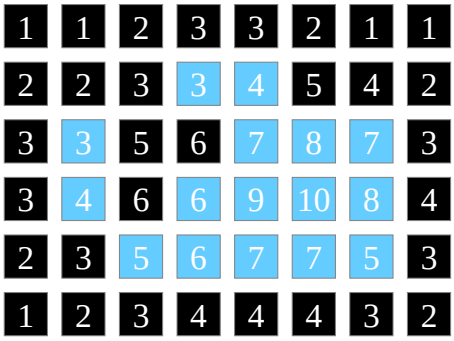
2.4.3 Finding Connected Components Using a Breadth First Search

The first algorithm used to find the connected components is based on a breadth-first-search (BFS) algorithm. The algorithm goes sequentially through each pixel of the image. When a pixel has a color different than black the BFS algorithm maps out the complete area of pixels of the encountered color. This means that for each pixel in a connected component, the algorithm has to visit each of its neighbors to check if the neighbor has been visited, or should be included in the search. After the component has been mapped out the algorithm still needs to visit all the pixels of the component once more. As can be observed in Figure 2.11(b), the maximum number of visits the algorithm needs to perform is 10. If the area was larger, the number of pixels that would need 10 visits would

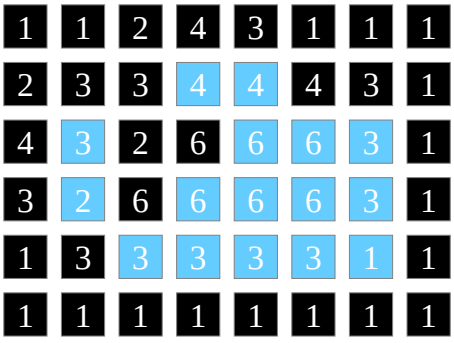
increase. This leads to a big constant in the algorithm, and runs too slow for a realtime application, so an improved algorithm is needed.



(a) Area to search for Connected Components



(b) Number of visits performed by BFS



(c) Number of visits performed by the Disjoint Sets method

Figure 2.11: BFS vs. Disjoint sets.

2.4.4 Finding Connected Components Using Disjoint Sets

An algorithm to label the pixels is presented in [24]. This algorithm describes a mask, see Figure 2.12, and a 2-pass method to label a thresholded image. The first pass labels all pixels and the second pass cleans up multiple labeled components. The algorithm presented here uses only the first pass and instead of a second pass, utilizes a data structure to keep track of components with multiple labels.

The algorithm uses disjoint sets and the neighborhood mask illustrated in Figure 2.12 This algorithm reduces the number of times a pixel is visited. An example of the results of this algorithm can be observed in Figure 2.11(c). The maximum number of visits has been reduced to 6, since this is a realtime application the improvement is significant. A more thorough description of the disjoint sets data structure can be found in [10].

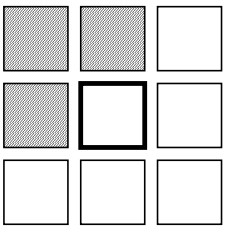


Figure 2.12: Algorithm 2's neighborhood mask.

The algorithms depends on a thresholded image, a label list of the same dimension as the image, a class called Component and an array that tracks all created Components. The Component class has a color field. This field is the same as the pixel color the Component represents. The class also has an index field which is the location in the Component array. There are also fields to track the minimum and maximum values of x and y of pixels that are added to the Component.

The Component class will be used to track the boundaries of a component and its center. The boundaries are used for identifying whether a Component is inside another Component, and the center of a marker will later be used for tracking calculations. The class contains methods as well, these are:

getRepresentative() recursively checks to see if this Component is the top level Component or if it is the child of another Component. If it is a child, it will ask its parent who the representative is. While checking for the representative a tree compression is performed so that subsequent calls return faster.

setRepresentative(Component) assigns a new representative for a Component.

addPixel(x, y) adds a pixel to a Component. For each call, a size variable is increased. The method also tracks the minimum and maximum x and y values for all calls. This results in a square that includes all pixels of a Component. Also, all x values are added together and all y values are

added together so that the values can be averaged for making a rough estimate of the center of the Component.

setColor(Color) sets the color of a Component. Neighboring pixels with the same color are part of the same Component.

setIndex(Index) every Component has a unique number associated with it. The number represents the index the Component has in the array of Components but also the number the labeling algorithm has given a pixel.

calculateCenter() calculates the center of a pixel by averaging the accumulated x and y values from addPixel(x,y). The computed center coordinate is returned.

The Problem With the getRepresentative() Method

This method is the most time-consuming part of this algorithm. An example of the problems that might occur can be observed in Figure 2.13. In the second row, 5 Components are created. In the third row, Component 1 is united with Component 2, and Component 1's representative is set to Component 2. Then Component 2 is united with Component 3, and Component 2's representative is set to Component 3. This continues throughout row three. On row 4, a new component is created. At the next pixel, Component 6 is about to be united with Component 1 when Component 1 checks its representative, which is Component 2. However Component 2 must check its representative all the way back to Component 5.

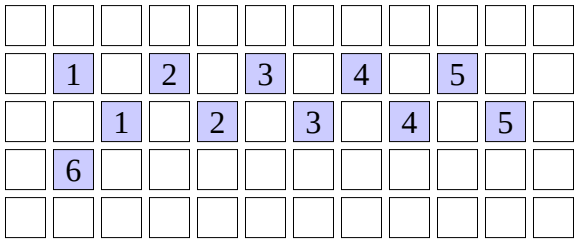


Figure 2.13: Deep recursion.

2.4.5 Component Size

Before a Component is added to the result-list, the size of the Component is checked. The maximum size of a Component is found by moving the camera as close as possible, while still being able to see all four markers and the yellow border. The minimum size is found by moving the camera as far away as possible and still be able to identify a group of pixels as a marker. This technique resulted in that the largest component was about 15 % of the total number of the pixels in an image. The minimum ended up as $\frac{1}{1000}$ of the 15 %.

2.5 The Algorithms

Figure 2.14 illustrates the complete process from thresholding an acquired image to returning a sorted set of markers. All of the algorithms are presented in Appendix A with pseudocode.

Algorithm 1 thresholds an image into 5 different colors; red, blue, yellow, white, and black.

Algorithm 2 takes a thresholded image as input and finds all connected components. The labeling algorithm, Algorithm 3, is used as a part of this step. The result of this algorithm is a set of Components of different colors. Before returning the result, the size of the Components are checked; the Components that are either too small or too big are removed.

Algorithm 3 checks if a pixel belongs to a neighboring Component or if the pixel is the start of a new Component.

Algorithm 4 uses the colored Components returned from Algorithm 2, and separates these into three groups: white, yellow, and a composite group of red and blue.

Algorithm 5 takes the white Components and the red and blue Components as input and checks all white Components if they are inside red or blue Components. If a white Component is inside a red or blue one a marker is created that has the merged characteristics of the white and the surrounding Component. The algorithm returns a list of markers.

Algorithm 6 compares the yellow Components from Algorithm 4 and the markers from Algorithm 5 to see if any of the markers are inside a yellow component. If more than four markers are found, the markers are checked for redundancy. Any redundant markers are removed. If three blue markers and one red marker are found inside a yellow Component, the four markers are returned, otherwise *null* is returned.

Algorithm 7 The markers returned from Algorithm 6 are here sorted in a counter-clockwise order, starting with the red marker.

2.6 Markers and Components

In this thesis, a marker is a white Component surrounded by a red or blue Component. Because of all the background noise it is difficult to know whether Components identified as a marker, truly is a marker, or just a random pair of Components. We will however try to reduce the chance of erroneously identified markers as much as possible.

With all the connected components identified and located, the process of separating markers from background noise can be done. This process involves separating the different colored Components from each other and then finding markers. When a set of one red and three blue markers are found inside a yellow frame, the marker square has been found. With all of the markers identified, they need to be sorted and the center of each of them must be calculated.

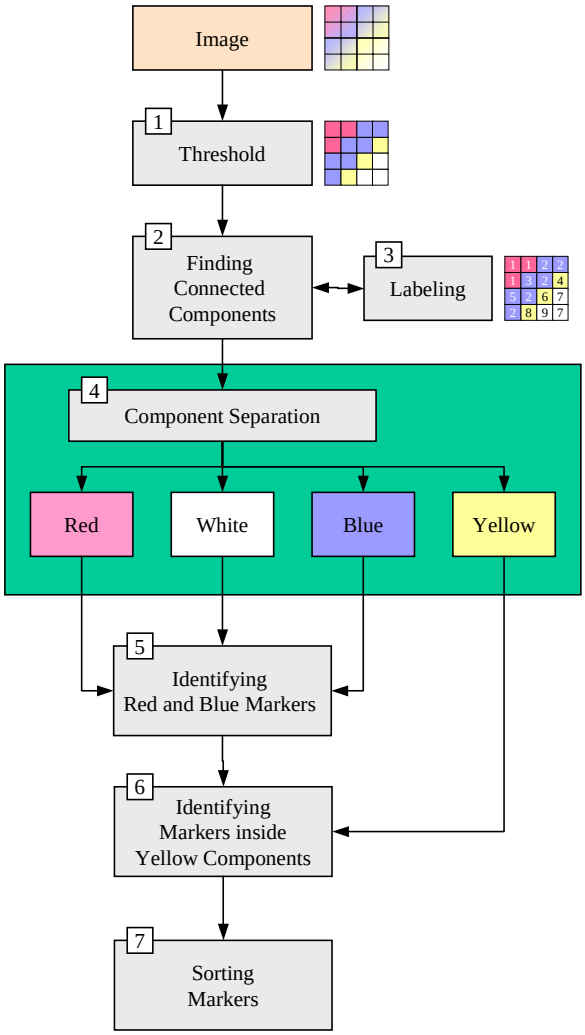


Figure 2.14: Overview of algorithms. Numbers represent numbering of algorithms in the appendix.

2.6.1 Separating the Components

After the connected components have been identified and localized, we want to separate the components into the different colors; white, red, blue and yellow. Algorithm 4 does this and returns three lists: w, rb and y. List w returns white components, list rb returns red and blue components while y returns the yellow components. The black components are of no interest so they are ignored.

2.6.2 Identifying the Markers

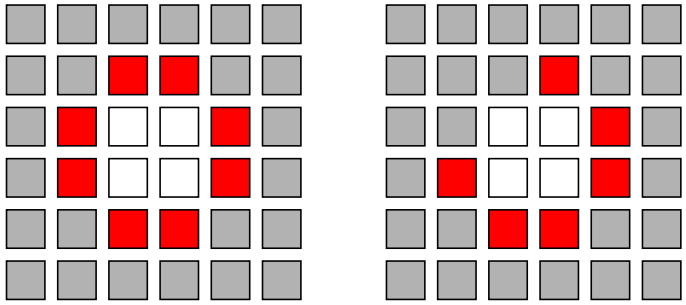
After separating the components, we must locate markers. Markers are created by finding white Components surrounded by a red or blue Component. All markers are then checked for localization inside a yellow border. If we end up

with three blue markers and one red, the marker plate has most likely been identified.

When identifying markers, it is necessary to check if a Component is inside another Component. Equation 2.5 describes when a Component is inside another.

$$inside(c1, c2) = \begin{cases} true & \text{if } c1.minX \geq c2.minX \\ & \& \quad c1.maxX \leq c2.maxX \\ & \& \quad c1.minY \geq c2.minY \\ & \& \quad c1.maxY \leq c2.maxY \\ false & otherwise \end{cases} \tag{2.5}$$

Algorithm 5 identifies red and blue markers by checking that a white component is surrounded by a red or blue component. This is illustrated in Figure 2.15(a). Due to the nature of Component boundaries, incorrect identification as shown in Figure 2.15(b) is possible.



(a) Correctly identified marker. (b) Incorrectly identified marker.

Figure 2.15: Red Component surrounding a white Component.

Algorithm 6 checks which of the markers, identified in Algorithm 5, that are inside a yellow component. There are potentially many yellow components, but if the scene has few yellow areas and the removal of too small and too large yellow components was successful, there should be only one area large enough to incorporate the red and blue markers.

In the case when two or more large yellow components exists and partly overlaps, one or more of the markers can be identified as being inside both of the components. The result is that the algorithm creates a list of components that has repeated members. With a total number of markers of more than four, the algorithm fails. To fix this, a check for repeated members is implemented, and all redundant markers are removed.

Due to the way a component's boundary is calculated, a different type of failure may arise. A Components boundary is the minimum and maximum pixel in the x and y direction of the Component. This method of obtaining the boundaries leaves gaps between the boundary and the actual pixels of the Component. In Figure 2.16, the black square is the boundary of the yellow Component, and the area inside the black square and outside of the yellow Component is the gap. The problem is that markers can be identified in this area, but these markers will be identified as being inside the yellow Component.

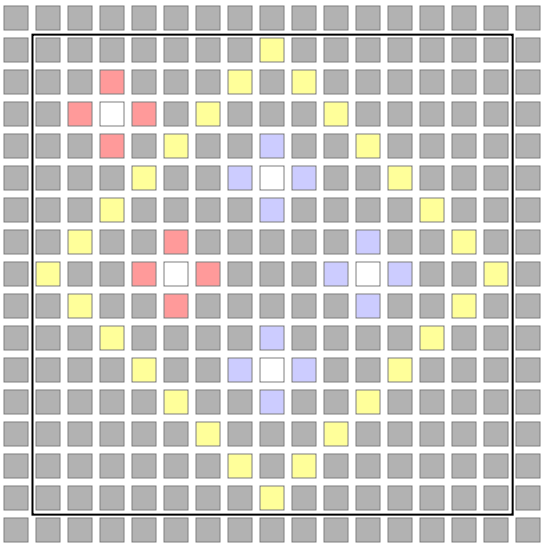


Figure 2.16: Failure during marker recognition. The upper left red marker is falsely identified to be inside the yellow square.

This is the case in Figure 2.16, where the red marker is outside the yellow Component.

2.6.3 Sorting the Markers

At this stage, all the markers have been identified and there are only three blue markers and one red marker inside the yellow border. The sorting of the markers is the next step. The sorting process results in the red marker being first followed by the blue markers sorted in counter-clockwise order.

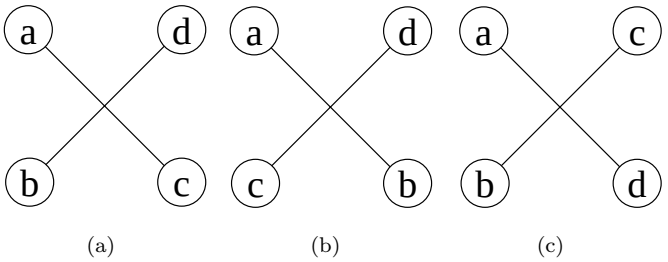


Figure 2.17: Different Marker Configurations.

The input data to this algorithm consists of four arbitrarily ordered markers; labeled a, b, c, and d based on the input order. It is assumed that the markers span a square. The sorting algorithm then tries to find which pair of markers that creates an intersection inside the square. Figure 2.17 illustrates the three possible marker configurations. For example, Figure 2.17(b) shows that marker a must pair up with marker b to create an intersection with the two other markers. For each of the three configurations there is an additional possibility,

for example in Figure 2.17(a), where the markers labeled b and d may swap positions. This totals to six configurations. The algorithm finds which of the six configurations the markers has by doing the following:

1. Finding the configuration that creates an intersection inside the spanned square.

2. Checking if the configuration is the one illustrated or the one with the swapped markers.

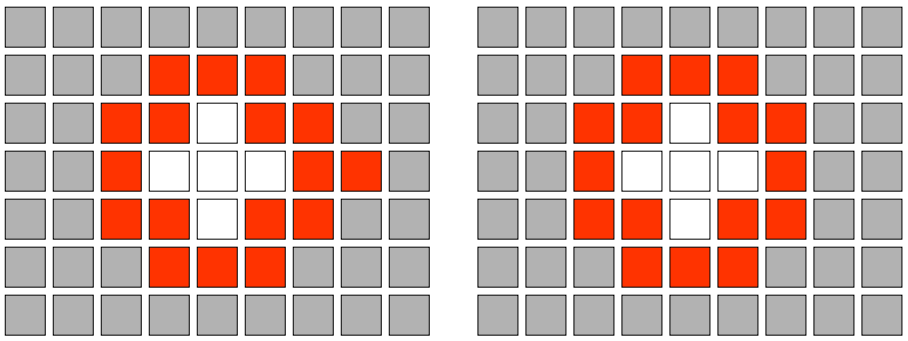
Algorithm 7 checks for all configurations and returns the sorted list. The algorithm uses a method called **isIntersetionInsideSquare**$(\mathbf{m_1}, \mathbf{m_2}, \mathbf{m_3}, \mathbf{m_4})$, where m_1, m_2, m_3, and m_4 are markers. The method checks the line created by the marker pair m_1 and m_2 against the line created by m_3 and m_4 for intersections. The test uses the intersection equations developed in Section 1.8.5, specifically Equation 1.5 and 1.6. The equations return two parameters: s and t. If $s \in [0, 1]$ and $t \in [0, 1]$ then the intersection is inside the square defined by the four markers.

Another method **isOnRight**$(\mathbf{u}, \mathbf{v}, \mathbf{w})$ checks wether the marker w is on the right or the left side of the line from u to v. It does this by using the 2D cross product described in Chapter 1.8.2. This method is used to check if the markers are swapped according to the previous example.

After sorting the markers, they must be shifted so that the red marker is the first marker in the list. The result of sorting and rearranging is a square of markers counter-clockwise order, which starts with a red marker and is followed by the blue. The list is returned as four points:

$$p_i = \{x_i, y_i\}, i = 1, 2, 3, 4$$

2.6.4 Center of a Marker

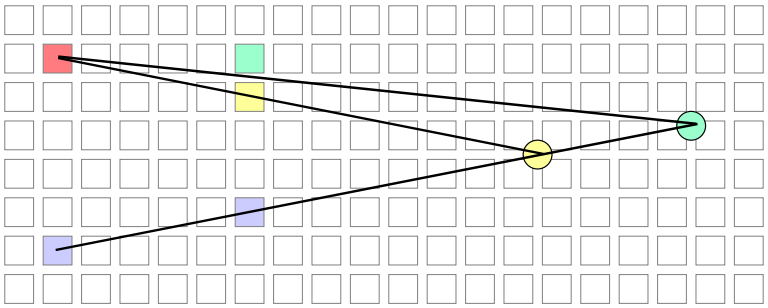


(a) Marker in frame 1 (b) Marker in frame 2

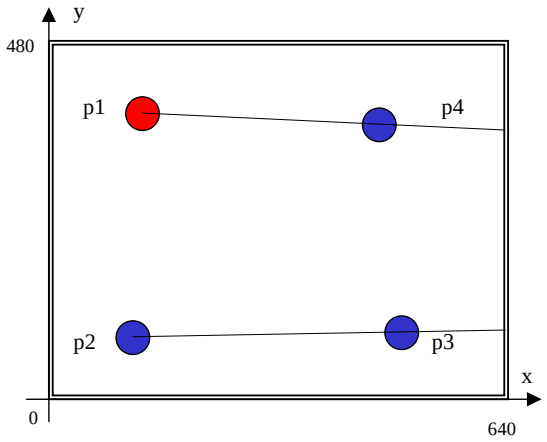
Figure 2.18: Marker shape change.

When a marker is created, all the pixels that belong to the marker are counted and the coordinates are summed separately for x and y directions, and the average is calculated. The problem with this is that the center of a marker

is unstable. In one frame the marker may look like Figure 2.18(a), but in the next frame, like Figure 2.18(b), the marker has lost a pixel on the right side. The marker center has in this example moved by 0.137 in the x direction. The unpredictable nature of the marker center results in intersection calculations being very unstable. This is illustrated in Figure 2.19(a), where the yellow circle indicates the intersection of a line that uses the yellow pixel as its marker center and the green circle indicates the new intersection when the marker center has moved half a pixel towards the green pixel. The intersection-point has moved 4 pixels in x direction and 1 pixel in the y direction. This causes significant errors in matrix calculations later on.



(a) Small scale illustration.



(b) Towards parallel lines.

Figure 2.19: Marker intersection instability.

The illustration in Figure 2.19(a) shows in a small scale, what happens when one marker moves its center by half a pixel. In a real scale, the distance between the markers can be in the hundreds of pixels range, and more than one marker can move its center. This makes the calculations even more unstable. For example consider the four points $p1$, $p2$, $p3$, and $p4$ shown in Figure 2.19(b) with coordinates $(70, 400)$, $(50, 50)$, $(450, 55)$, and $(420, 390)$. Calculating the intersection between the lines, we get something close to $(8585, 156)$. If $p4$ moves by 0.15 of a pixel in the y direction, the resulting intersection moves to about $(8675, 157)$, which is a difference of 90 pixels in the x direction. This fluctuation

increases when the intersecting lines moves towards being parallel.

To counteract this problem, the center of a marker is calculated by averaging the position over many frames. However, this causes the tracking to be delayed. It takes some frames for the tracked position of the marker to catch up with the actual position. For example: a marker center is located at $(x1, y1)$, the camera is moved and in the next frame the center is located at $(x2, y2)$. Due to averaging the center will use a certain amount of time before it represents the new location. This delay is experienced as a slow transition from position $(x1, y1)$ to $(x2, y2)$ over several frames. This is not a desired behavior.

A good compromise between the two solutions has been developed. The new method calculates the average marker position frame by frame and checks if the current position of a marker is within a 3 pixel radius of the average. If the current marker position is inside of the 3 pixel radius the averaged position is used, otherwise the current position is used. This results in a tracking that does not have slow transitions between marker positions, but is unstable for as long as it takes to build up a new stable average point. This solution works well in practice, especially when the framerate is reasonably high.

Chapter 3

Camera Calibration Theory

The theory presented in this chapter builds a foundation for the techniques presented in the next chapter. This chapter describes how a 3D point is projected onto an image plane and how parallel lines in 3D space are projected as lines converging to a vanishing point. Lens distortion is covered as a source of error. Theory for changing coordinate system and finding the point where the distance between two 3D lines is at a minimum is described.

3.1 Perspective Projection

A 3D point (x, y, z) is projected onto the image plane z' as (x', y') by the following equations:

$$x' = \frac{xf}{z}$$

$$y' = \frac{yf}{z}$$

$$z' = f$$

where the focal length f is the distance from the center of projection (COP) to the image plane. The COP is the point located distance f in front of the image plane and perpendicular to the image plane center. A point is projected to the image plane on a line that spans from the point itself to the COP. Figure 3.1 illustrates this with only the upper half of the image plane.

3.1.1 Projection of Parallel Lines

The projection of parallel lines onto the image plane, results in lines that converge on a vanishing point [13]. This is true as long as the lines are not parallel to the image plane. The following equations prove this claim:

A 3D line with t as a parameter can be written as:

$$L = P + t\vec{d}$$

In component form:

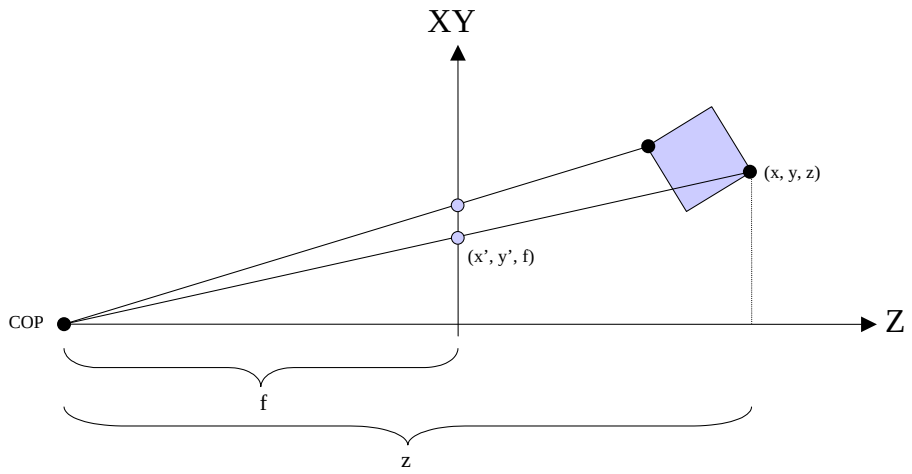


Figure 3.1: The xy-axis is the image plane. COP is the center of projection.

$$L_x = P_x + td_x$$

$$L_y = P_y + td_y$$

$$L_z = P_z + td_z$$

The projection of this line is:

$$X'(t) = f\frac{L_x}{L_z} = f\frac{P_x + td_x}{P_z + td_z}$$

$$Y'(t) = f\frac{L_y}{L_z} = f\frac{P_y + td_y}{P_z + td_z}$$

What we want is to see what happens to the projected line as the line goes towards infinity:

$$\lim_{t\to\infty} X'(t) = \lim_{t\to\infty} f\frac{P_x + td_x}{P_z + td_z} = f\frac{d_x}{d_z}$$

$$\lim_{t\to\infty} Y'(t) = \lim_{t\to\infty} f\frac{P_y + td_y}{P_z + td_z} = f\frac{d_y}{d_z}$$

This proves that the vanishing point of a line is independent of the localization of that line and only dependent of the direction of the line.

The fact that parallel lines converge on a vanishing point will be utilized with the markers identified previously, as the markers are a square with two sets of parallel lines.

3.2 Lens Distortion

Perspective projection in computer graphics relies on the pinhole camera model. The pinhole camera model, shown in Figure 3.2, describes the mapping between

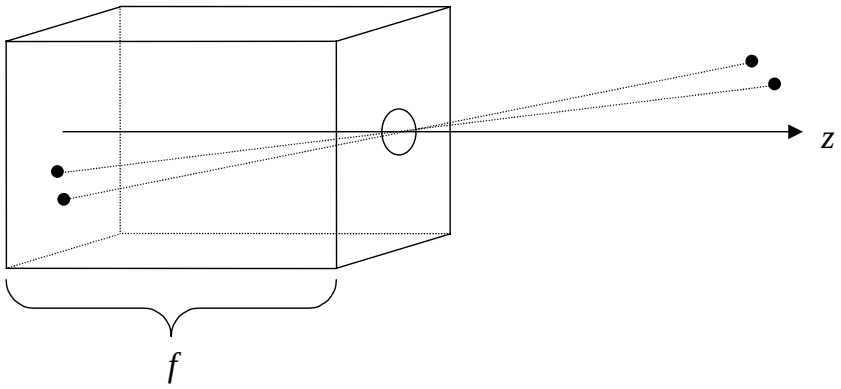
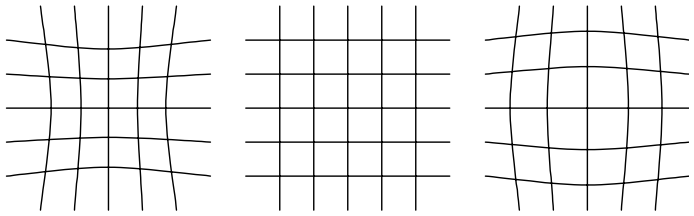


Figure 3.2: The Pinhole Camera Model.

3D world coordinates and 2D image coordinates, where a 3D point is projected onto the back wall of the pinhole camera.

When using real world camera systems, this camera model fails [21, 27, 28]. This is especially true with low-cost or wide-angle lens systems. When a lens is included in the model, radial distortion applies to the projected image. There are two different types of lens distortions: pincushion- and barrel distortion. Lens distortion is also the reason for the true image center not being at $\frac{width}{2}$ and $\frac{height}{2}$ where $width$ and $height$ are the dimensions of the image.



(a) Pincushion distortion. $\kappa_3 < 0$

(b) Undistorted.

(c) Barrel distortion. $\kappa_3 > 0$

Figure 3.3: Lens Distortion.

The inverse radial distortion functions can be written as:

$$x_u = x_d + x_d \sum_{i=0}^{\infty} \kappa_i r_d^{i-1}$$

$$y_u = y_d + y_d \sum_{i=0}^{\infty} \kappa_i r_d^{i-1}$$

where:

$$r_d = \sqrt{x_d^2 + y_d^2}$$

and x_u and y_u are the undistorted coordinates and x_d and y_d are the distorted coordinates.

Practical test have shown that it is sufficient to take only the parameters κ_3 and κ_5 into account [27]. Using this we can simplify the equations:

$$x_u = x_d + x_d(\kappa_3 r_d^2 + \kappa_5 r_d^4)$$

$$y_u = y_d + y_d(\kappa_3 r_d^2 + \kappa_5 r_d^4)$$

If we know the lens properties κ_3 and κ_5, these equations tell us how we can take a distorted point and find its undistorted position. When $\kappa_3 < 0$ the type of distortion is pincushion, while $\kappa_3 > 0$ gives barrel distortion. So to invert the effects of radial lens distortion, the parameters κ_3 and κ_5 must be found. Procedures for finding these parameters are described in [21, 27]. Both of these techniques take into account that straight lines should be straight after projection. So by using the curvature of lines and their distance from the center of the image plane the parameters can be estimated. Different ways of calculating the distortion parameters using calibration patterns are described in [18, 28, 30].

3.3 Changing Coordinate System

In the article [15] a method of obtaining the rotation transformation from three vectors is described. The technique will be explained in the next chapter, but some of the theory behind this technique is described here.

A rotation can be described as a change in coordinate system [6, 17], so by finding the transformation needed to go from one coordinate system to another we have found the rotation that describes this change. We assume that the coordinate systems have the same scale and that the corresponding axes of the coordinate systems have the same angles between the other axes in the same system.

A coordinate system in $\mathbf{R^3}$ contains three basis vectors that are linearly independent. For $\mathbf{R^3}$ the following set is called the standard basis [17] .

$$\vec{i} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \vec{j} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \vec{k} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{3.1}$$

These vectors can also be set up in a matrix:

$$\mathbf{I} = \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

This set is orthonormal, which means that all the vectors are orthogonal to each other and have unit length. When basis vectors are orthonormal, transformations preserve distances and angles, thus a transformation with an orthonormal matrix will be a rotation [13]. Another important property of an orthonormal matrix is that the inverse is the transpose of the matrix $\mathbf{M^{-1}} = \mathbf{M^T}$ [17].

Two points p and q in $\mathbf{R^3}$ can be written as:

$$p = \begin{bmatrix} p_x & p_y & p_z \end{bmatrix}$$

$$q = \begin{bmatrix} q_x & q_y & q_z \end{bmatrix}$$

Point p can, with respect to the standard basis, be written as:

$$P = p \begin{bmatrix} \vec{i} \\ \vec{j} \\ \vec{k} \end{bmatrix} = \begin{bmatrix} p_x & p_y & p_z \end{bmatrix} \begin{bmatrix} \vec{i} \\ \vec{j} \\ \vec{k} \end{bmatrix} = p_x\vec{i} + p_y\vec{j} + p_z\vec{k}$$

Point q can be represented with an arbitrary basis vectors as:

$$Q = q \begin{bmatrix} \vec{b_1} \\ \vec{b_2} \\ \vec{b_3} \end{bmatrix} = \begin{bmatrix} q_x & q_y & q_z \end{bmatrix} \begin{bmatrix} \vec{b_1} \\ \vec{b_2} \\ \vec{b_3} \end{bmatrix} = q_x\vec{b_1} + q_y\vec{b_2} + q_z\vec{b_3}$$

We want to represent point q in the standard basis coordinate system. To do this, we first need to find the transformation that transforms the standard basis vectors into point Q's basis vectors:

$$\begin{bmatrix} \vec{b_1} \\ \vec{b_2} \\ \vec{b_3} \end{bmatrix} = \mathbf{M} \begin{bmatrix} \vec{i} \\ \vec{j} \\ \vec{k} \end{bmatrix} \tag{3.2}$$

The matrix $\mathbf{M}$ equals:

$$\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}$$

Expanding Equation (3.2) gives:

$$\begin{bmatrix} \vec{b_1} \\ \vec{b_2} \\ \vec{b_3} \end{bmatrix} = \begin{bmatrix} m_{11}\vec{i} + m_{12}\vec{j} + m_{13}\vec{k} \\ m_{21}\vec{i} + m_{22}\vec{j} + m_{23}\vec{k} \\ m_{31}\vec{i} + m_{32}\vec{j} + m_{33}\vec{k} \end{bmatrix} \tag{3.3}$$

Solving Equation (3.3) using (3.1) gives:

$$\mathbf{M} = \begin{bmatrix} \vec{b_1}^T \\ \vec{b_2}^T \\ \vec{b_3}^T \end{bmatrix}$$

We want to take an arbitrary point q from the coordinate system that Q is defined in and find its representation with the standard basis coordinate system as c:

$$q^T \begin{bmatrix} \vec{b_1} \\ \vec{b_2} \\ \vec{b_3} \end{bmatrix} = c^T \begin{bmatrix} \vec{i} \\ \vec{j} \\ \vec{k} \end{bmatrix}$$

and since:

$$\begin{bmatrix} \vec{b_1} \\ \vec{b_2} \\ \vec{b_3} \end{bmatrix} = \mathbf{M} \begin{bmatrix} \vec{i} \\ \vec{j} \\ \vec{k} \end{bmatrix}$$

we get:

$$q^T \mathbf{M} \begin{bmatrix} \vec{i} \\ \vec{j} \\ \vec{k} \end{bmatrix} = c^T \begin{bmatrix} \vec{i} \\ \vec{j} \\ \vec{k} \end{bmatrix}$$

The points are now using the same representation so we can write:

$$q^T \mathbf{M} = c^T$$

finally:

$$c = \mathbf{M}^T q$$

So a point q from a coordinate system with arbitrary basis vectors can be represented in the standard basis coordinate system by multiplying that points coordinates with a matrix with the basis vectors from point Q's coordinate system as column vectors.

3.4 Minimizing the Distance Between Lines

One of the things that is necessary in the next chapter is to find an intersection between two 3D lines. When calculating the intersection between two lines in 3D space, the number precision available does not guarantee an intersection point. Instead we have to find the coordinates where the distance between the two lines is minimized. The method described here minimizes a function by squaring it and differentiating. The differentiated functions are then set equal to zero and solved.

The lines are described by a point, a direction vector, and the parameters α and β:

$$L_1 = p_1 + \alpha \vec{d_1} \tag{3.4}$$

$$L_2 = p_2 + \beta \vec{d_2} \tag{3.5}$$

To find the intersection point we set $L_1 - L_2 = 0$

$$f(\alpha, \beta) = \alpha \vec{d_1} - \beta \vec{d_2} + (p_1 - p_2) = 0 \tag{3.6}$$

To differentiate the function we need to square the function

$$min(f(\alpha, \beta)) = min(f(\alpha, \beta)^2) = min((\alpha \vec{d_1} - \beta \vec{d_2} + (p_1 - p_2))^2)$$

$$\begin{aligned} f(\alpha, \beta)^2 &= \alpha^2 \vec{d_1} \cdot \vec{d_1} + \beta^2 \vec{d_2} \cdot \vec{d_2} - 2\alpha\beta \vec{d_1} \cdot \vec{d_2} \\ &+ 2\alpha \vec{d_1} \cdot (p_1 - p_2) - 2\beta \vec{d_2} \cdot (p_1 - p_2) + (p_1 - p_2)^2 \end{aligned}$$

Partial differentiating and setting equal to zero gives us

$$\frac{\partial f}{\partial \alpha} = 2\alpha\vec{d_1} \cdot \vec{d_1} - 2\beta\vec{d_1} \cdot \vec{d_2} + 2(p_1 - p_2) \cdot \vec{d_1} = 0 \tag{3.7}$$

$$\frac{\partial f}{\partial \beta} = 2\beta\vec{d_2} \cdot \vec{d_2} - 2\alpha\vec{d_1} \cdot \vec{d_2} - 2(p_1 - p_2) \cdot \vec{d_2} = 0 \tag{3.8}$$

exchanging

$$\begin{aligned} a &= \vec{d_1} \cdot \vec{d_1} \\ b &= \vec{d_1} \cdot \vec{d_2} \\ c &= (p_1 - p_2) \cdot \vec{d_1} \\ d &= \vec{d_2} \cdot \vec{d_2} \\ e &= (p_1 - p_2) \cdot \vec{d_2} \end{aligned}$$

the derivatives become

$$\frac{\partial f}{\partial \alpha} = 2\alpha a - 2\beta b + 2c = 0 \tag{3.9}$$

$$\frac{\partial f}{\partial \beta} = 2\beta d - 2\alpha b - 2e = 0 \tag{3.10}$$

solving Equation 3.9 for α

$$\alpha = \frac{2\beta b - 2c}{2a} = \frac{\beta b - c}{a} \tag{3.11}$$

inserting Equation 3.11 into Equation 3.10 and solving for β

$$\beta = \frac{ea - cb}{ad - b^2} \tag{3.12}$$

inserting the value of β into Equation 3.11 and inserting the value of α into Equation 3.4 and β into Equation 3.5, returns the intersection point or the point that gives the minimum distance between the lines.

Chapter 4

Camera Calibration

In this chapter the process of taking a 2D image and finding the camera configuration of that scene is explained. Some reasons for error and instability are pinpointed.

When 3D data is projected onto a 2D plane, all depth information is lost. By observing how the projection works, some insight can be obtained on how the reconstruction back to 3D may be accomplished. The process of reconstructing a scene from a 2D image is not a goal of this thesis. However, extrapolating the camera configuration is an integral part of the reconstruction process which is what this chapter will cover.

The techniques presented in this chapter use the marker positions acquired earlier to obtain a rotation matrix and a translation matrix that will later be used to position the camera in the final scene.

4.1 Camera Calibration Overview

A large number of methods to calibrate a camera exists. Many of these techniques were evaluated before deciding on [15] as the calibration method for this thesis. There are many different ways too calibrate a camera, for example:

[28, 30] both use complex calibration patterns that demand a much more complex recognition algorithm. Even though both techniques deliver precise data about the camera the resource usage is higher than the selected method.

[18] uses a circular calibration pattern with thin lines. This would be a problem because the quality of the camera might make the lines indiscernible at a distance.

[12] this technique uses a 3D structure to perform the calibration, but since we want the projection area to be flat this method is not suitable.

[15] uses vanishing points to calibrate the camera. The vanishing points are calculated based on the position of only four points. By only needing four points the recognition algorithm does not need to be very complex. A variant of this is method is we have selected.

4.2 Camera Calibration Intrinsics and Extrinsics

The process of finding the intrinsics and extrinsics of a physical camera based on images taken by a camera is called *camera calibration*. Intrinsics are the internal geometric and optical characteristics and extrinsics are the 3D position and orientation relative to a world coordinate system.

The intrinsic parameters consist of focal length, lens distortion and image center. The extrinsics consist of rotation and translation.

The only intrinsic parameter that will be used is the focal length. The lens distortion will be ignored, while the center of the image plane will be assumed to be $\frac{width}{2}$ and $\frac{height}{2}$. Both rotation and translation from the extrinsic parameters will be calculated.

4.2.1 Focal Length

To calculate the rotation and translation needed to emulate the location of a camera, the focal length needs to be known. The focal length of a camera changes when a camera zooms or the focus changes. Since the cameras used in this thesis has a static focus setting and does not have a zoom option, the focal length can be static. This is true as long as the focus setting remains unchanged.

In this thesis the focal length has been found through a manual process. The process consists of aligning a box to the marker square by manually increasing or decreasing the focal length. Figure 4.1 illustrates this briefly.

4.3 Finding the Rotation Matrix

To calculate the rotation matrix the spatial location of the four markers in a counter clockwise orientation must be known see Figure 4.2(a).

$$p_i = \{x_i, y_i\}, i = 1, 2, 3, 4$$

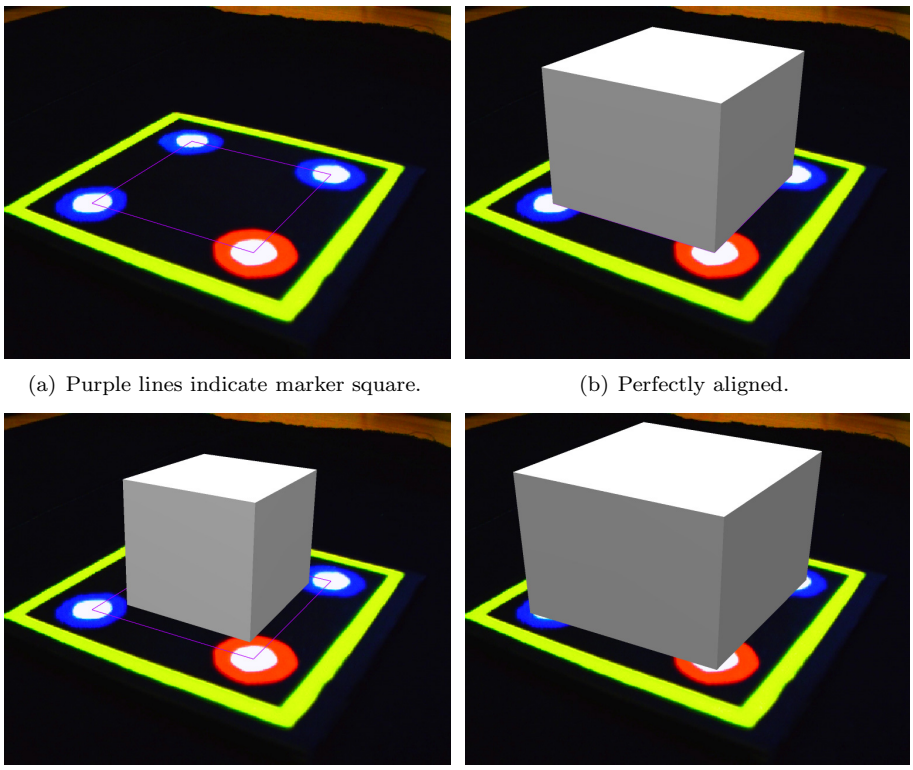
It is assumed that all four markers lie in the same plane, as well as being placed in a square pattern.

4.3.1 Vanishing Points

The calibration techniques presented in this chapter depend on vanishing points. Using the results from Section 3.1.1 we want to find the vanishing point of two lines that we know in the real world to be parallel, specifically the lines generated by the square of markers.

First we need to consider the special cases when the projection of parallel lines are parallel, these cases are:

- The image plane is parallel to one of the sets of lines generated by the markers.
- The camera is placed directly above the set of lines looking towards the center, in this case both lines in each set are parallel.



(a) Purple lines indicate marker square.

(b) Perfectly aligned.

(c) Focal length too long.

(d) Focal length too short.

Figure 4.1: Alignment of focal length.

In the cases when parallel lines arise, special considerations are necessary. These considerations are simple:

- If only one set contains parallel lines rotate 90°, 180°, or 270° around the z-axis. The rotation to use depends on the position of the red marker.

- If both sets contain parallel lines rotate 90° around the x-axis and the rotation around the z-axis can be found by calculating the angle of the red marker to the x and y axis of the image. Finally scale can be derived by the size of the marker square.

Empirically the situation of parallel lines does not occur, probably due to the low precision of the captured data.

A vanishing point is calculated along the lines consisting of the projected points $\{p_1, p_2\}$ and $\{p_3, p_4\}$, illustrated in Figure 4.3(b), which defines two lines that are known to be parallel, as for the lines $\{P_1, P_2\}$ and $\{P_3, P_4\}$ in Figure 4.3(a). The equations for the lines and their intersection can be found in Section 1.8.5. These lines generate the first vanishing point. The second vanishing point is calculated the same way but the points used for the line equations are $\{p_1, p_4\}$ and $\{p_2, p_3\}$.

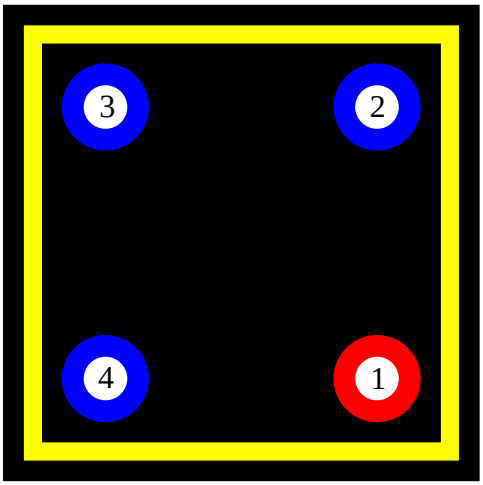


Figure 4.2: Example setup of markers.

4.3.2 Camera Coordinates

In Figure 4.4 camera coordinates are illustrated. The ilustration shows how the real world objects, $\{P_1, P_2, P_3, P_4\}$, are project onto the image plane, $\{p_1, p_2, p_3, p_4\}$, and where the vanishing points reside $\{F_u, F_v\}$. The vanishing points lie on the image plane, and the distance from COP to the image plane is the focal length as described in section 3.1. The point C is the image center and is the orthogonal projection of COP onto the image plane. The basis vectors $\vec{i}$ and $\vec{j}$ show what typically is defined as the x and y direction of the image plane, while the vector $\vec{k}$ shows the z direction.

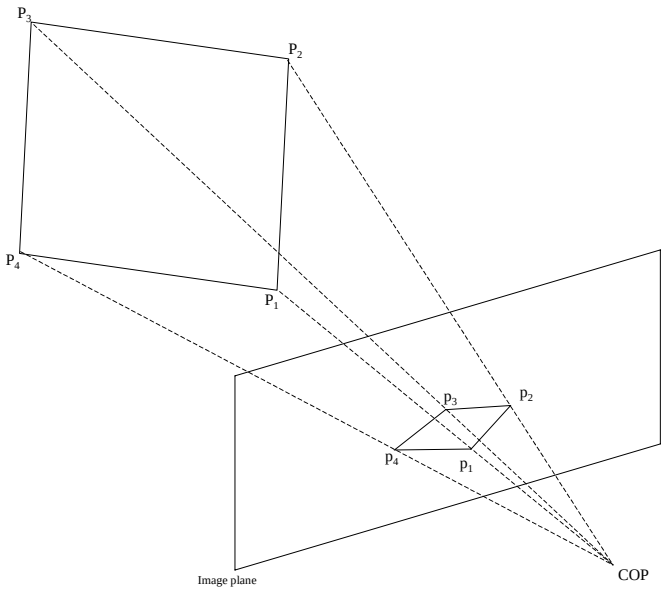
4.3.3 The Rotation Matrix

The method explained in this section returns the rotation matrix needed for correct camera placement. The method has been taken from [15].

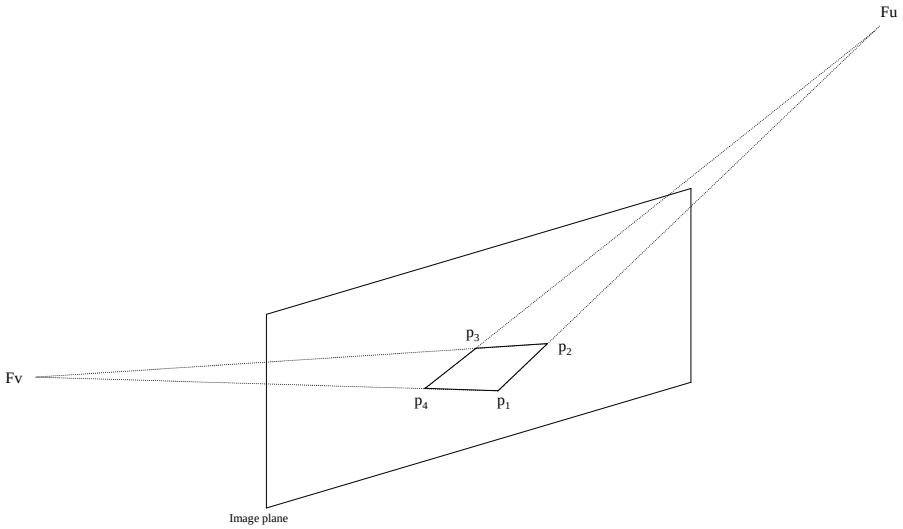
Figure 4.5 shows the local coordinate system of the object placed at P_1, and at COP another coordinate system is defined based on the vanishing points. The lines from COP to Fu and Fv define the vectors $\vec{u'}$ and $\vec{v'}$, and $(\vec{u'} \times \vec{v'}) = \vec{w'}$. Since the projections of these lines has the vanishing points Fu and Fv respectively, then $\vec{u} \parallel \vec{u'}$ and $\vec{v} \parallel \vec{v'}$. Since the two different coordinate systems are identical we can use $\vec{u'}$, $\vec{v'}$, and $\vec{w'}$ as basis vectors for changing coordinate system from camera coordinates.

$$\vec{u'} = \frac{\vec{COP}Fu}{\|\vec{COP}Fu\|} = \frac{1}{\sqrt{Fu_i^2 + Fu_j^2 + f^2}} \begin{bmatrix} Fu_i \\ Fu_j \\ f \end{bmatrix}$$

$$\vec{v'} = \frac{\vec{COP}Fv}{\|\vec{COP}Fv\|} = \frac{1}{\sqrt{Fv_i^2 + Fv_j^2 + f^2}} \begin{bmatrix} Fv_i \\ Fv_j \\ f \end{bmatrix}$$



(a) Projection of markers and parallel lines. COP is the center of projection.



(b) Vanishing Points.

Figure 4.3: Shows the projection of a set of points and the vanishing points in the projection.

$$\vec{w'} = (\vec{u'} \times \vec{v'})$$

The rotation matrix becomes:

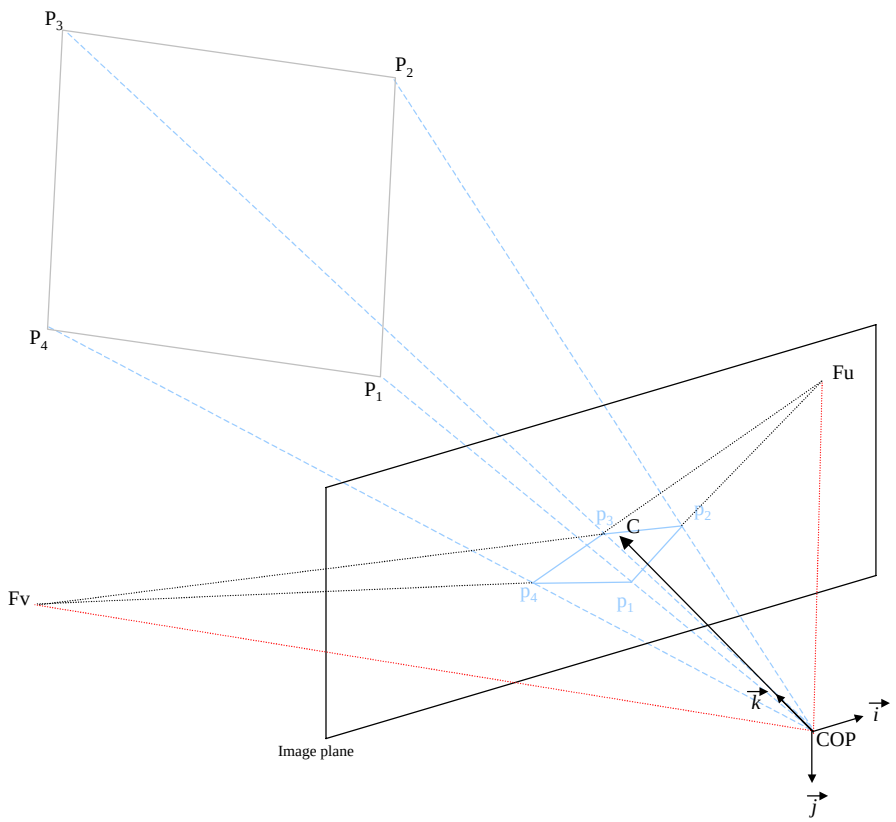


Figure 4.4: Camera Coordinates. C is the image center and COP is the center of projection.

$$\mathbf{M} = \begin{bmatrix} \vec{u'} & \vec{v'} & \vec{w'} \end{bmatrix} = \begin{bmatrix} \frac{Fu_i}{\sqrt{Fu_i^2+Fu_j^2+f^2}} & \frac{Fv_i}{\sqrt{Fv_i^2+Fv_j^2+f^2}} & w'_i \\ \frac{Fu_j}{\sqrt{Fu_i^2+Fu_j^2+f^2}} & \frac{Fv_j}{\sqrt{Fv_i^2+Fv_j^2+f^2}} & w'_j \\ \frac{f}{\sqrt{Fu_i^2+Fu_j^2+f^2}} & \frac{f}{\sqrt{Fv_i^2+Fv_j^2+f^2}} & w'_k \end{bmatrix}$$

4.3.4 The Cross-Product $\vec{w'}$

When calculating the cross product $(\vec{u'} \times \vec{v'})$ for the rotation matrix the vector $\vec{w'}$ can point in two different directions depending on which direction the vectors point in relation to each other. The direction vector $\vec{u'}$ is always calculated based on the lines $\{p_1, p_2\}$ and $\{p_3, p_4\}$ but in which direction $\vec{u'}$ will point relative to the lines is unknown. This is illustrated in Figure 4.6.

We must find which side the point vp is located relative to the lines. To do this, we can check that the point vp is on the left-hand or right-hand side of the line $\{p_1, p_4\}$. To do this we first calculate the vectors:

$$\vec{a} = (p_4 - p_1)$$

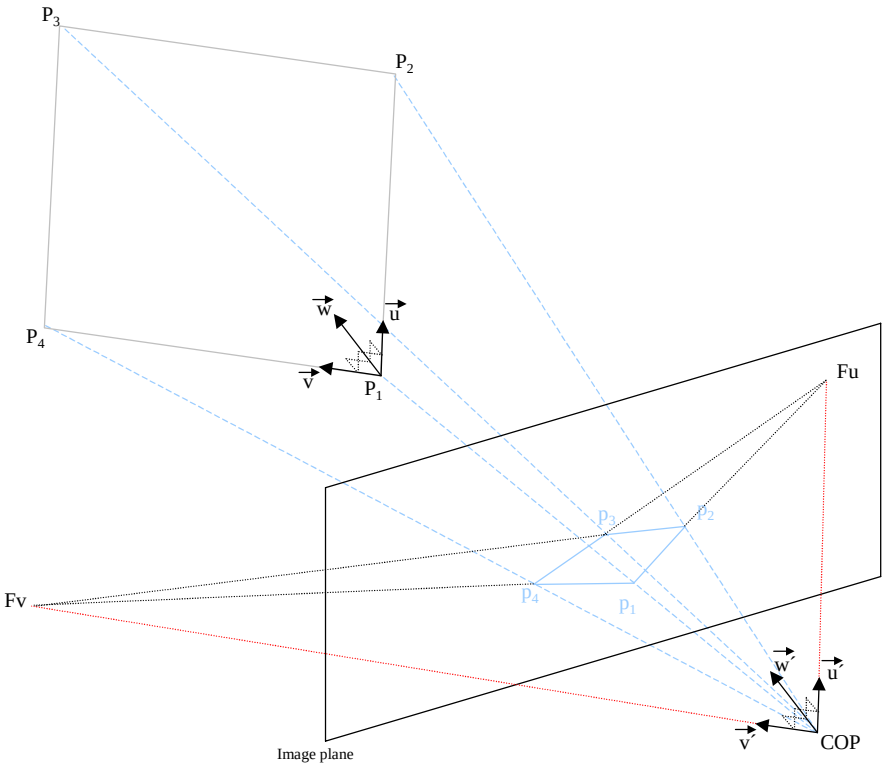


Figure 4.5: Shows the local coordinate system of the object and the estimated coordinate system at COP.

$$\vec{b} = (vp - p_1)$$

then we take the 2D cross product:

$$c = (\vec{a} \times \vec{b})$$

The sign of the scalar value c will tell what side the point vp is on:

$$\begin{array}{ll} \text{If } c < 0 & vp \text{ is on the right-hand side of } \{p_1, p_4\} \\ \text{If } c > 0 & vp \text{ is on the left-hand side of } \{p_1, p_4\} \end{array}$$

The case when $c = 0$ is not considered since a vanishing point on the line $\{p_1, p_4\}$ is not possible. The same calculations must be done for the line pair $\{p_1, p_4\}$ and $\{p_2, p_3\}$, but here the vanishing point is checked against the line $\{p_1, p_2\}$.

The reason for these checks is that when the vector $\vec{w'}$ is calculated, we want it to point in the right direction. Figure 4.7 illustrates this. Figure 4.7(a) shows the initial case. If the vector $\vec{v'}$ changes direction, as in the change from Figure 4.7(a) to Figure 4.7(b), the calculated vector $\vec{w'}$ points down instead of up. The same happens when $\vec{u'}$ changes direction, as in Figure 4.7(c).[1] If both vectors

[1] In an application, this is observed as an object being turned upside down.

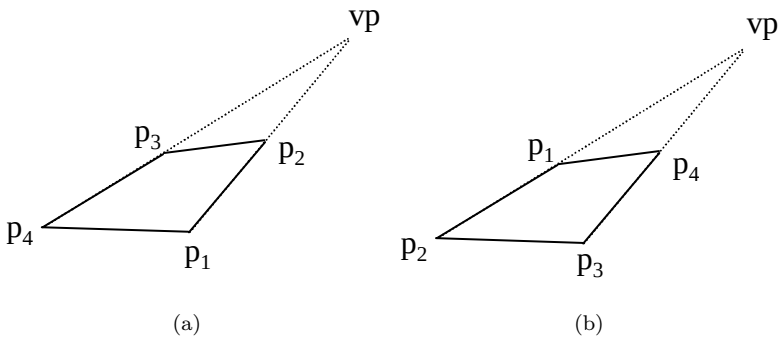


Figure 4.6: Different directions for vanishing points.

change direction, the vector $\vec{w'}$ remains unchanged, as in Figure 4.7(d). So when a vector changes direction, we want the vector $\vec{w'}$ to point in the opposite direction. This is done by negating the vector $\vec{w'} \rightarrow -\vec{w'}$.

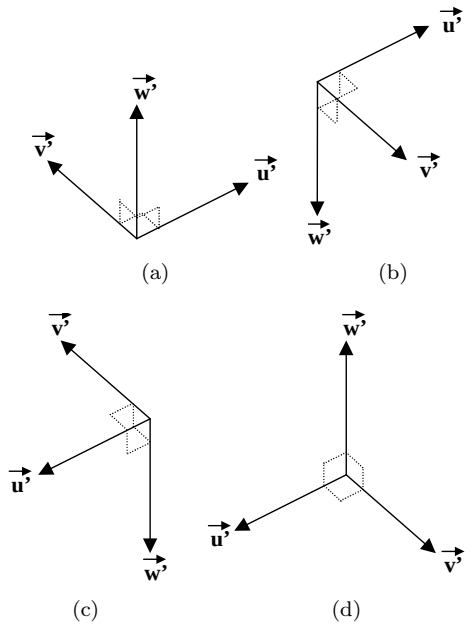


Figure 4.7: Different orientations of $\vec{u'}$ and $\vec{v'}$ and the resulting cross product $\vec{w'}$.

4.4 The Translation Vector and its Length

The translation vector gives us the direction and distance to move a 3D object. The method described here is based on the work done by [15].

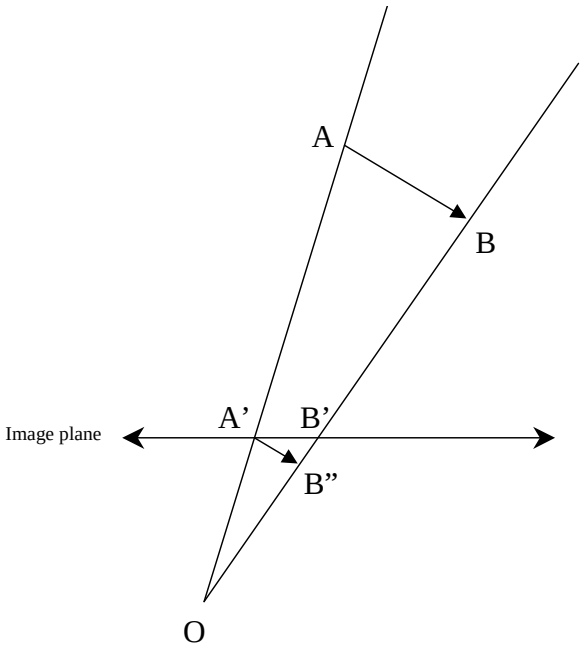


Figure 4.8: Translation.

From Figure 4.8 we want to find the distance from O to A. The direction vector is already known, and is written as:

$$\frac{\vec{OA}}{\|\vec{OA}\|} = \frac{\vec{OA'}}{\|\vec{OA'}\|} = \frac{1}{\|\vec{OA'}\|} \begin{bmatrix} A_x^{'} \\ A_y^{'} \\ f \end{bmatrix} = \frac{1}{\sqrt{A_x^{'2} + A_y^{'2} + f^2}} \begin{bmatrix} A_x^{'} \\ A_y^{'} \\ f \end{bmatrix}$$

The distance $\|\vec{AB}\|$ is set to 1, thus the distance calculated for $\|\vec{OA}\|$ will only be useable as a scale factor. For our purposes, this is sufficient. In our setup, the distance $\|\vec{AB}\|$ can easily be measured and used since it is a static distance, but this will only add complications later in the visualization process.

In Figure 4.8 the point $B^{''}$ is the intersection of the line through $A^{'}$ in the same direction as $\vec{AB}$ and the line OB.

The distance that we don't know is $\|\vec{A'B''}\|$. To find this distance the point $B^{''}$ has to be calculated first. This is done by using the minimization formula stated in Section 3.4.

To find the distance $\|\vec{OA}\|$ we use Euclid on the triangles OAB and $OA^{'}B^{''}$ in Figure 4.8

$$\frac{\|\vec{A'B''}\|}{\|\vec{AB}\|} = \frac{\|\vec{OA'}\|}{\|\vec{OA}\|} \tag{4.1}$$

which we solve and get

$$\|\vec{OA}\| = \frac{\|\vec{AB}\|\|\vec{OA'}\|}{\|\vec{A'B''}\|} \tag{4.2}$$

This results finally in the vector

$$\vec{OA} = \|\vec{OA}\|\frac{\vec{OA'}}{\|\vec{OA'}\|} \tag{4.3}$$

which is the translation vector.

$$\vec{\mathbf{T}} = \vec{OA} \tag{4.4}$$

In the application the point P_1 is used as A and point P_2 or P_4 is used as B. This results in p_1 being used as $A^{'}$ and p_2 or p_4, the projection of P_2 or P_4, being used as $B^{'}$. Finally the vectors $\vec{A'B''}$ and $\vec{AB}$ are equal to either the vector F_u or F_v corresponding to the choice of P_2 or P_4.

Chapter 5

Visualization

The main goal of this thesis is to merge the real world with an artificial 3D world. The final step to achieve this is the visualization. This chapter describes the techniques necessary to use the captured image as a background and the rotation matrix and translation vector to position the 3D model at the correct location. The information provided in this chapter has been compiled from [6–8, 13, 29]. This is only a brief introduction into the field of computer graphics, and covers only the topics necessary to complete the visualization part of this thesis.

5.1 OpenGL

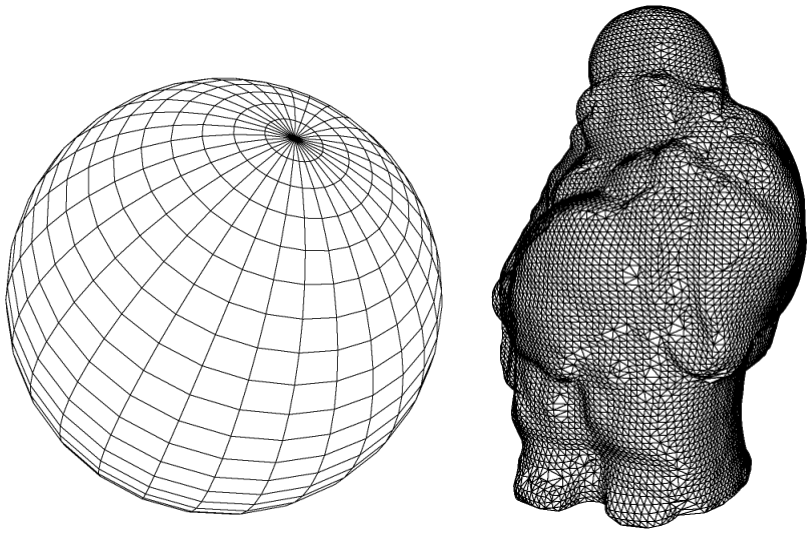
The graphics library used in this thesis is OpenGL (GL stands for Graphics Library) introduced by SGI in 1992. It has since become the industry's most widely used 3D graphics application programming interface. Implementations of the library can be obtained for most platforms and programming languages. OpenGL is also supported in hardware in most modern graphics cards. OpenGL is now maintained by The OpenGL Architecture Review Board (ARB).

5.2 Vertices, Triangles, and Quadrilaterals

3D models are usually composed of triangles and quadrilaterals. Quadrilaterals are four sided polygons. The polygons that a model is composed of are called faces. Each corner of a polygon is called a vertex. Figure 5.1(a) illustrates how a sphere can be created by quadrilaterals and triangles. The triangles are placed at the poles of the sphere. The lines of the wireframe model intersect at the vertices of the sphere. Figure 5.1(b) illustrates a 3D model of Buddha composed exclusively of triangles. The Buddha model was obtained from [2].

5.3 Translations, Rotations, and Scaling

To move and size a 3D model, translation, rotation, and scaling operations are necessary. In OpenGL these operations can be performed by using the following library routines:



(a) A wireframe sphere composed of quadrilaterals and triangles at the poles.

(b) A wireframe Buddha model composed exclusively of triangles.

Figure 5.1: Wireframe 3D models.

glTranslatef(x, y, z) where x, y, and z are distances along the different axis.

glRotatef(a, x, y, z) where a is the angle and x, y, and z describe a vector to rotate around.

glScalef(x, y, z) where x, y, and z describe the scaling factor along each axis.

Internally, OpenGL uses matrices to perform the transformations, and it is possible to use a matrix directly to perform the same operations. The method for this is $glMultMatrix(m)$, where m is a 4×4 matrix.

$T(x, y, z)$, $R_x(\theta)$, $R_y(\theta)$, $R_z(\theta)$, and $S(x, y, z)$ are the matrices that perform transformations equivalent to the previously mentioned OpenGL methods.

$\mathbf{T}(\mathbf{x}, \mathbf{y}, \mathbf{z})$ is the translation matrix where x, y, and z are the translation distances.

$$\mathbf{T}(\mathbf{x}, \mathbf{y}, \mathbf{z}) = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$\mathbf{R_x}(\theta)$, $\mathbf{R_y}(\theta)$, **and** $\mathbf{R_z}(\theta)$ are the rotations matrices for each axis and θ is the rotation angle. The rotations only apply to $R_x(\theta) = glRotatef(\theta, 1, 0, 0)$, $R_y(\theta) = glRotatef(\theta, 0, 1, 0)$, and $R_z(\theta) = glRotatef(\theta, 0, 0, 1)$.

$$\mathbf{R_x}(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

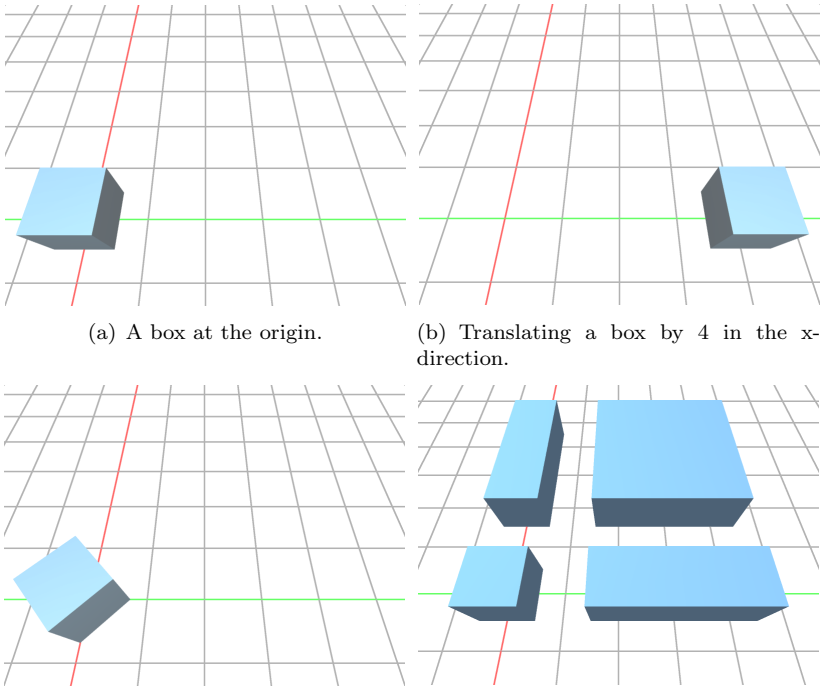
$$\mathbf{R_y}(\theta) = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R_z}(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$\mathbf{S(x, y, z)}$ is the scaling matrix where x, y, and z are the scaling factor for each axis.

$$\mathbf{S(x, y, z)} = \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 5.2 illustrates the result of using these operations on a box.



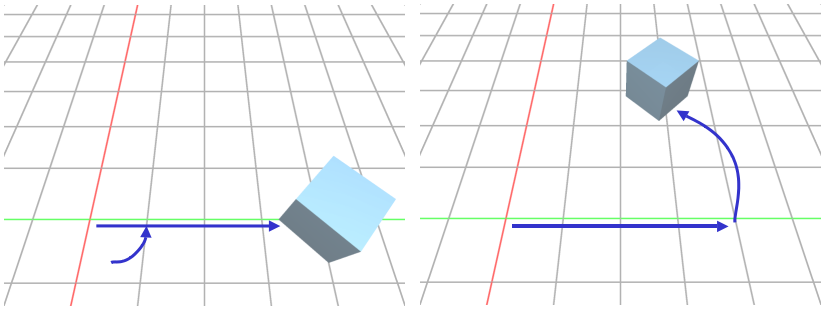
(a) A box at the origin.

(b) Translating a box by 4 in the x-direction.

(c) Rotating a box by 45° around the y-axis.

(d) Scaling of a box.

Figure 5.2: Transformations on a box.



(a) Rotating a box by 45° and then translating by 4.

(b) Translating a box by 4 and then rotating by 45°.

Figure 5.3: Order of transformations.

Transformations can be multiplied together to create composite transformations. By multiplying rotation matrices together it is possible to create rotations around arbitrary axes. The order of multiplying transformations is important. This is illustrated in Figure 5.3(a) and Figure 5.3(b). As seen in the figures; doing a rotation before translating is very different from doing the transformations in the opposite order. It is important to note that the center of rotation is at the origin. The same care of transforming in the right order is necessary with the other transformations. Transformation-operations of equal types can be composed in any order.

5.4 Shading

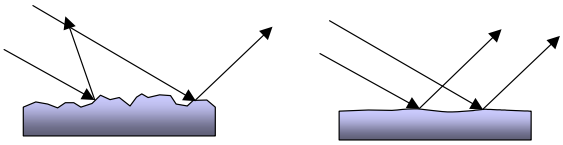
Drawing a 3D model as a wireframe or filling every face with a color will create a flat looking object. To create the illusion of 3D every face must be shaded. The shading is dependent on light position, position of viewer, and the surface of the model. All surfaces reflect light, and it is this fact that enables us to see objects in the real world. There are three types of reflections available in OpenGL: ambient, diffuse, and specular.

Ambient reflection is background light or diffusely reflected light from other objects.

Diffuse reflection is the light that reflects off of surfaces that are rough. The roughness scatters the light in all directions and this makes diffusely lit surfaces appear the same from all directions; see Figure 5.4(a).

Specular reflection is the light that is reflected from smooth surfaces; see Figure 5.4(b). This creates a highlight on the shaded model which is the reflection of the lightsource. The smoother a surface is, the more the surface resembles a mirror. Because it acts as a mirror the reflection is dependent on the position of the viewer.

The final result is composed of the sum of all three reflection types for each lightsource. OpenGL provides two methods of shading a model:



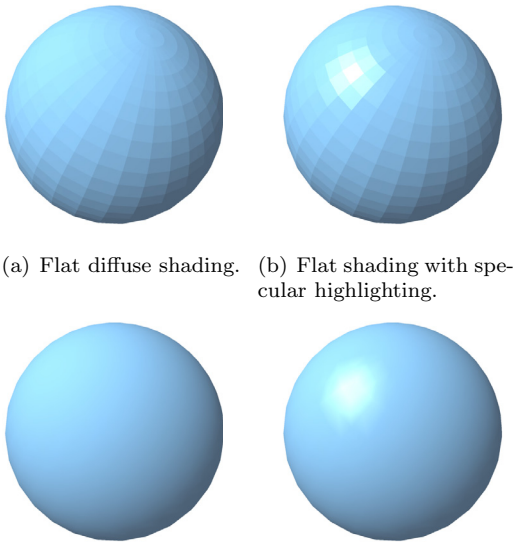
(a) Rough surface. (b) Smooth surface.

Figure 5.4: Rough and smooth surfaces.

Flat shading is the fastest and also the simplest method. Shading of this type only checks the reflections for each face and colors the entire face with that color.

Smooth shading is slower but renders a smooth surface more realistically. Shading of this type, called Gouraud shading, sums up the reflections for each vertex, and linearly interpolates the color over the face.

Figure 5.5 illustrates the two different types of shading on a sphere. In addition, the spheres show the difference between purely diffuse surfaces, Figures 5.5(a) and 5.5(c), and surfaces that have a specular highlight, Figures 5.5(b) and 5.5(d). Finally Figure 5.6 illustrates the shading of a complex model.



(a) Flat diffuse shading. (b) Flat shading with specular highlighting.

(c) Smooth diffuse shading. (d) Smooth shading with specular highlighting.

Figure 5.5: Results of using different shading models on a sphere.

5.5 Texture Mapping

To improve the realism of a rendered model, texture is applied. A texture can be a 2D image as shown in Figure 5.7(a). The s and t axes in the image refer to

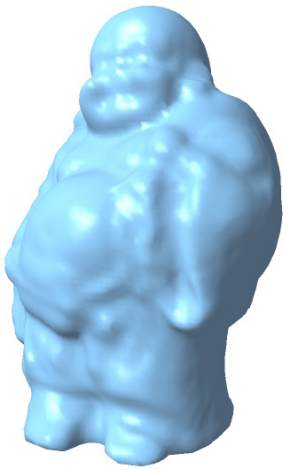
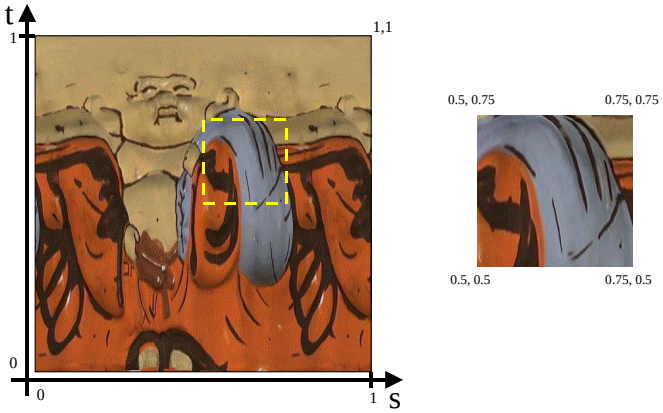


Figure 5.6: A smooth shaded Buddha.



(a) Texture mapping coordinates and a texture map.

(b) A Textured quadrilateral.

Figure 5.7: Texture mapping coordinates.

texture coordinates. Each vertex of a 3D model can have an associated texture coordinate. The yellow square shows the area of the texture referenced from the quadrilateral in Figure 5.7(b). The result of using the texture in Figure 5.7 on a model, can be seen in Figure 5.8.

In this thesis, texture mapping is used in two cases; the first is to apply texture on the models used, and the second is to use the captured image as a backdrop picture.

5.6 Stereoscopic Viewing

All the images created with OpenGL are projected into 2D. This makes the images look flat. The human vision is capable of perceiving depth informa-

Figure 5.8: A smooth shaded Buddha with texture mapping.

tion because each eye sees an object from a slightly different position. This is illustrated in Figure 5.9. This fact can be exploited by using a HMD with two independent monitors to achieve the same effect. Each monitor displays an image of the scene, but from slightly offset camera positions. The offset emulates the distance between the eyes. The result is an effect that fools the brain to believe that the generated image has depth.

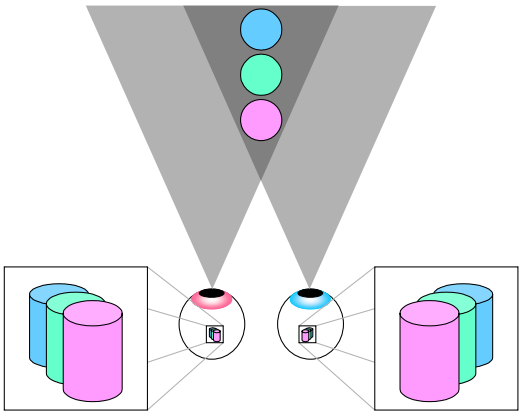


Figure 5.9: Stereoscopic viewing.

Chapter 6

Results

In this chapter, the result of assembling all the pieces from the previous chapters is presented. We provide a number of examples that demonstrates the capabilities and flexibility of the developed system. This chapter also illustrates and proposes solutions to some inadequacies that appeared during development and usage of the system.

6.1 Example Setups

This thesis has in detail explored how to create a MR application. The illustrations provided here show the physical aspect of the application. Figure 6.1 shows a person using the HMD and camera with the handheld marker. Figure 6.2 illustrates how the application functions when the camera has a static location. The image on the monitor is updated in real time.

The Logitech camera is used with the HMD, because this camera is very easy to attach to the goggles without needing to modify the camera or the goggles.

Figure 6.1: A user wearing a HMD with the Logitech camera attached, is exploring the application with the handheld markers.

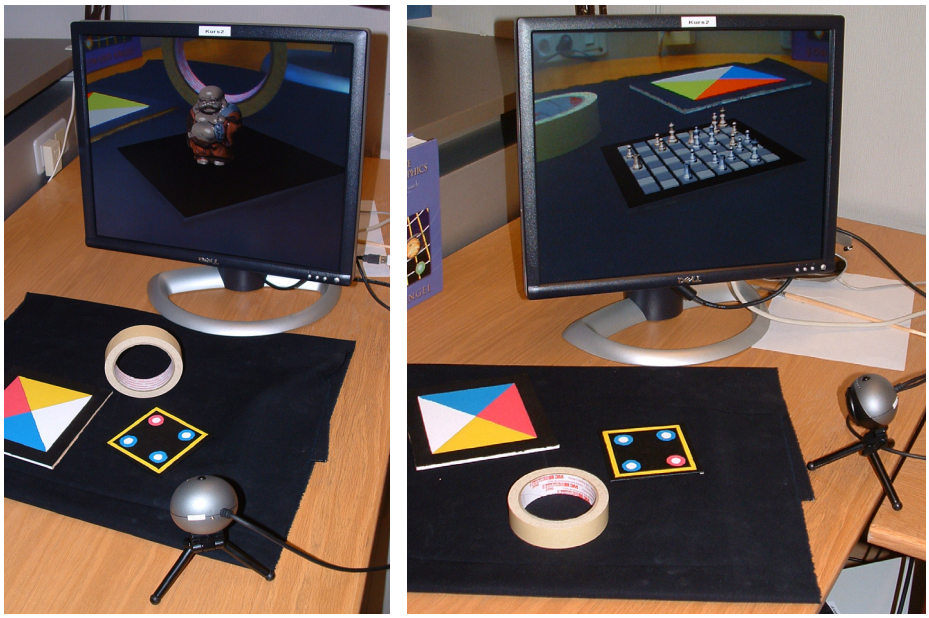


Figure 6.2: Static positioned camera and the result on a LCD monitor.

6.2 Handheld Markers

This section illustrates results gained when using the small handheld marker pattern shown in Figure 6.3. The handheld markers lets the user, with some limitations, rotate, scale, and translate a virtual object using his hands. Figure 6.4 shows a landscape and a game of chess projected onto the handheld marker square. Figure 6.5 demonstrates a virtual Rubik's cube. The Rubik's cube is animated and interactive; the light blue frame indicates which face is active.

This small marker pattern works very well because it is easy to keep the markers inside the field of view of the camera and it allows the camera to get very close to the markers.

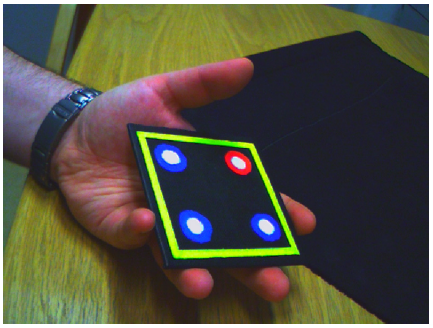
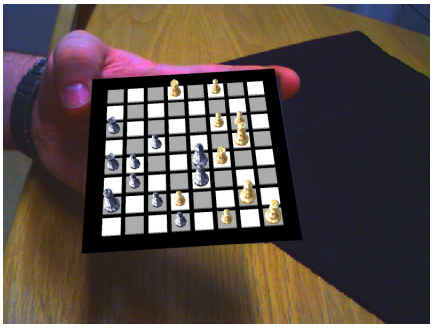
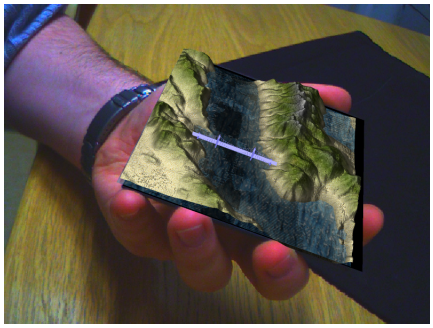


Figure 6.3: Using the handheld markers.

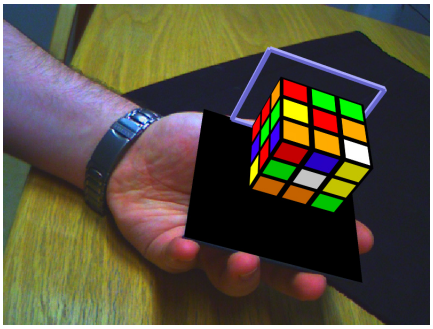


(a) A handheld game of chess.

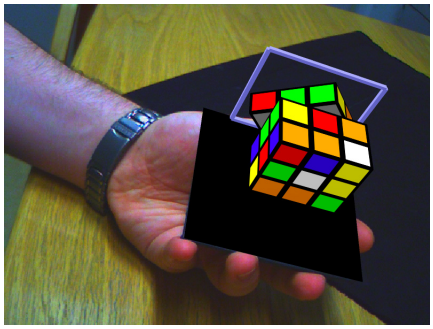


(b) A handheld landscape.

Figure 6.4: Using the handheld markers.



(a) Rubik's cube.



(b) Rubik's cube animated.

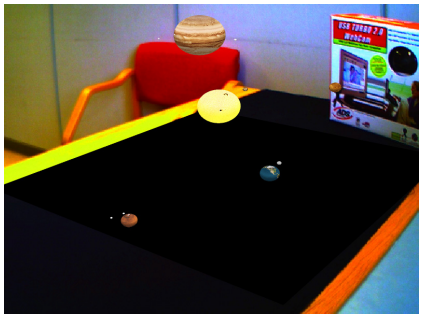
Figure 6.5: Playing with a Rubik's cube.

6.3 Examples

The following shows examples of what the setup can be used for. Figure 6.6 illustrates the point of view used in Figure 6.7. Figure 6.7(a) is an animated model of the solar system. Figure 6.7(b) shows a game of chess. Figure 6.7(c) shows a landscape created from a heightmap. Real world data can be used for simulation. For example, a construction company can use this to illustrate the impact of building a bridge in a certain area. Finally, Figure 6.7(d) shows a 3D model of a Buddha statue.

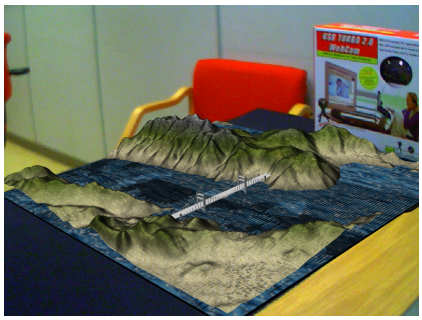
Figure 6.8 illustrates the result of a camera capturing at different locations along a path.

Figure 6.6: Point of view.



(a) The local solar system.

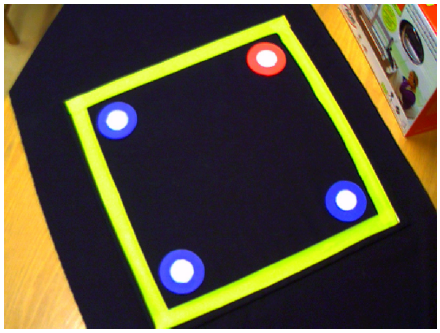
(b) A game of chess.



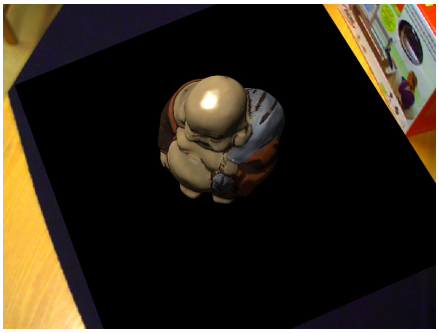
(c) Explore 3D landscape.

(d) Decorate your office.

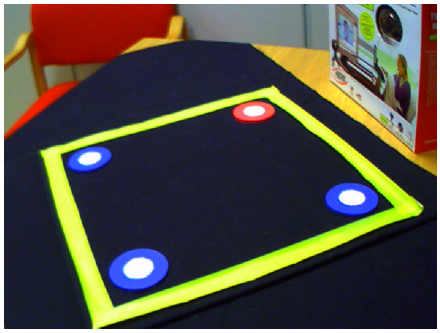
Figure 6.7: Pretty Images.



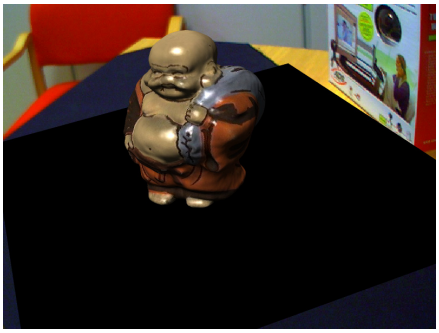
(a) Captured top view.



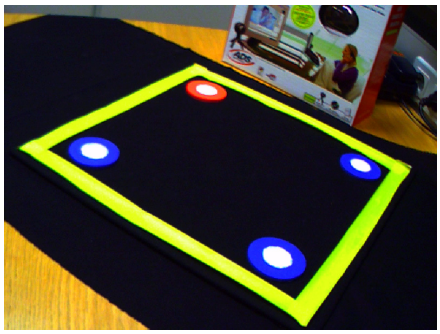
(b) Mixed reality top view.



(c) Captured side view.



(d) Mixed reality side view.



(e) Captured front view.

(f) Mixed reality front view.

Figure 6.8: The result of a moving camera.

6.4 Illusion Breakers

Many problems occur that breaks the illusion of realism. This section describes several of these problems with proposals on how to solve them.

6.4.1 Hardware and Software Development Tools Implications

The most important implication both hardware and software has on the application is reduced frame rate. For example the driver-software of the ADSTech camera used up to 60 % of the computer's resources while capturing, leaving the application with only 40 % of the resources. With too little computing power, the number of frames processed per second is reduced which results in a lowered frame rate. This could have been solved with a more powerful computer with dual CPUs or a camera that did not use the same amount of resources. Another software related impact on frame rate is Java. Java is an interpreted language and does not directly utilize single instruction multiple data (SIMD) instructions on the CPU. The use of these instructions could have improved the speed of the thresholding algorithm.

The quality of the cameras used has an impact on the marker recognition. Both cameras have a limited amount of interaction with the internal settings, like brightness, gamma, exposure, and white balance. Some of these settings are automatically applied or not available at all. For instance, the Logitech camera is difficult to set up correctly, and even at the optimal settings produces images with distorted colors which influences the recognition algorithms.

6.4.2 Loosing Track

For a number of previously mentioned reasons, the software may fail in recognizing the markers. If the application is functioning and then looses the markers, the image in Figure 6.9 illustrates what will most likely happen. By design, the 3D model stays at the last known marker position.

Figure 6.9: Loosing track of the markers.

6.4.3 Unstable Translation Calculations

A problem that occured early was that the 3D object seemed to be incorrectly positioned inside the marker square. With a cube placed in the center, one of the sides was longer or shorter than the other. An example of this can be seen in 6.11. The reason for this visual error is related to the length of the translation vector. The length of the vector varies depending on which axis is selected (see Figure 6.10). If the distance along the x-axis from the red marker to the blue marker is chosen. the length has one value. Using the distance along the y-axis from the red marker to the blue marker results in another value. Figure 6.11 illustrates what happens to the y-axis when the distance along the x-axis is chosen.

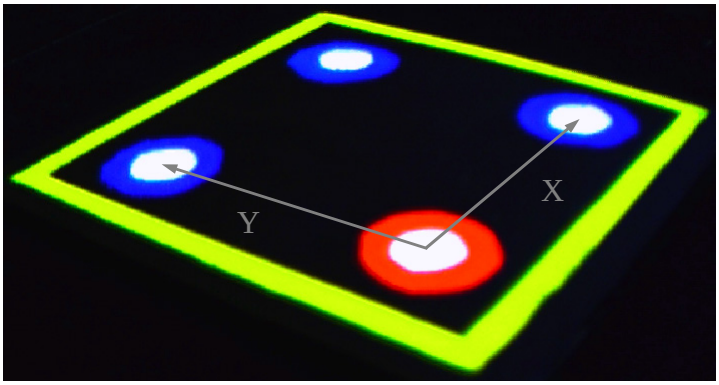
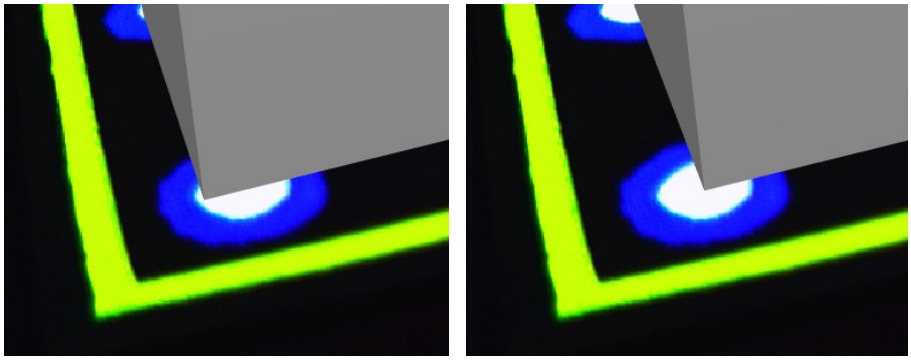


Figure 6.10: The marker square axes.



(a) Scale along y-axis is incorrect.

(b) Error compensated for.

Figure 6.11: Unstable translation vector length.

To compensate for the erroneous scale, the following method has been developed. The results for both vectors are calculated, but only one is used. If the x-axis is chosen, the ratio of the y component of both results are calculated, and the 3D model is scaled in the y-axis with this ratio. This again results in a new type of visual error: deformation of the 3D model. The deformation creates a slightly too fat or too thin model. This problem, though, is less noticeable and

annoying than the incorrect positioning.

There are several reasons for the unstable vector calculations but the main reasons are lens distortion and the assumption that the image center is at $\frac{width}{2}$ and $\frac{height}{2}$.

6.4.4 In Front or Behind?

Figure 6.12 illustrates one of the most commonly occurring problems. Figure 6.12(a) shows a correctly rendered scene with the 3D model rendered in front of the tape roll. Figure 6.12(b) shows what happens when the tape roll is moved in front of the markers; the 3D model is still rendered on top of the tape roll, which destroys the illusion of depth. The reason for the error is the fact that the background image does not contain any depth information, and is always rendered in the background.

A solution to this can be found by using two cameras. Setting up two cameras with a horizontal offset that capture the same scene can be used to acquire depth information from the scene. First, corresponding pixels between the two images must be found, then the difference in position can be used to calculate the depth of each pixel. The depth information can then be used to selectively render pixels in front of or behind the 3D model. An example of this technique can be found in [23].

6.4.5 Matching Virtual and Real Light Sources

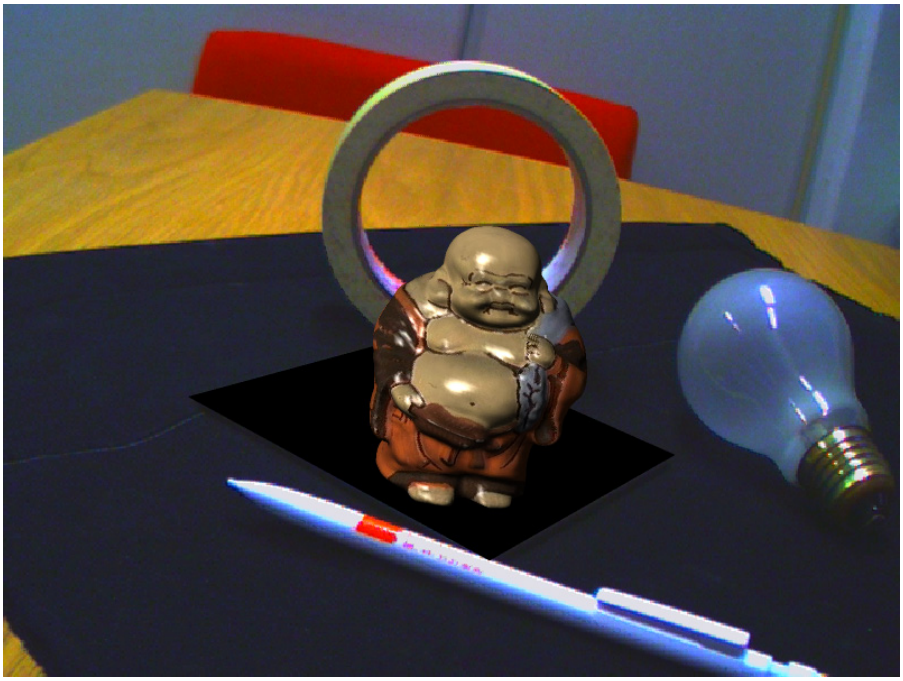
To create the best illusion of a 3D model occupying the space of the markers, the lighting of the virtual world should match the lighting of the captured world. Figure 6.13 illustrates the effect of matching and mismatching a light source. The positioning of the lights must be done manually.

There are methods to semi-automatically position a light source before using the system. A simple method of calibrating the geometry of light sources can be found in [20].

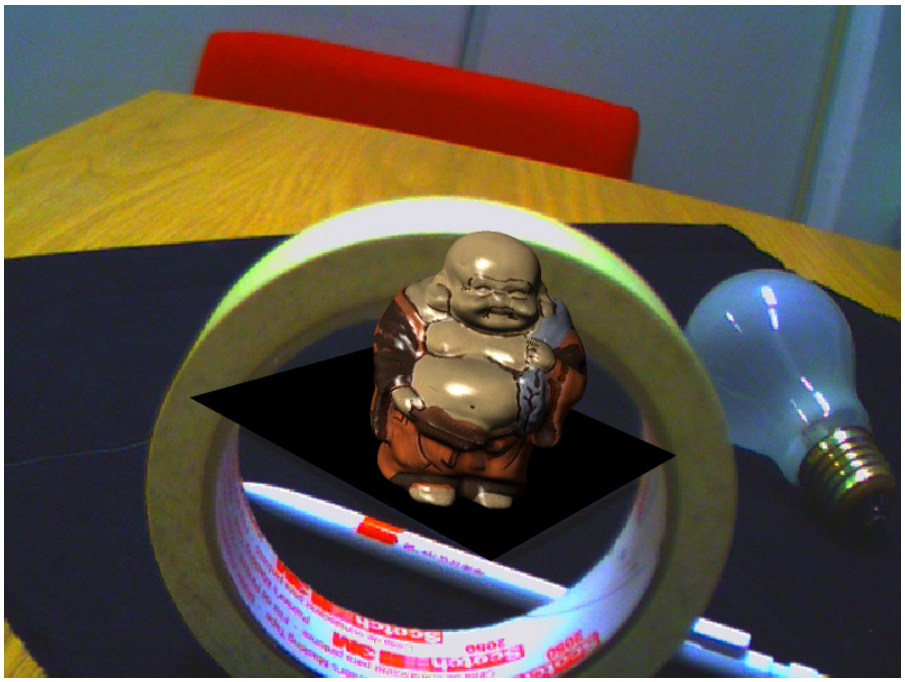
Another possibility would be to use image-based lighting using a light probe image [11]. Image-based lighting is a way to simulate the lighting of a scene using High Dynamic Range Images (HDRI). HDRI are images that record the position of every light source in a scene as well as the intensity at different exposures.

All positioning methods result in mismatching lights either if the markers are moved a considerable distance from their initial position, a light source moves, or the intensity of a light changes.

Figure 6.13 also illustrates another important lighting effect necessary to achieve realism: shadows. Here, the camera casts a shadow, but the statue does not. Implementing shadows would increase the level of realism considerably, but the problem of mismatching lights would have a greater impact because the flaw would be more noticeable. Applying shadows would also add a new range of problems to overcome. For example, if a real world object is supposed to receive a shadow, the shadow would be projected incorrectly, as the background image is flat and does not contain any 3D information. This can be solved much in the same way as the problem in Section 6.4.4



(a) Tape roll behind markers.



(b) Tape roll in front of markers.

Figure 6.12: Drawing order.

(a) Matching lights.

(b) Mismatching lights.

Figure 6.13: Matching and Mismatching lights.

Chapter 7

Conclusion and Future Work

This chapter presents the conclusion of this thesis and thoughts on how the system developed can be enhanced and extended in the future.

7.1 Conclusion

We have successfully created a complete system that enables a user to explore virtual objects mixed with the real world. By moving the head or physically move around a square of markers, the user can study the objects from any position.

Using the handheld markers the user is able to rotate, translate, and look closer at the projected objects with hand movements only.

An effective marker design was developed that enabled us to use a fast recognition algorithm. A method of obtaining markers from an image was developed by using conventional image analysis techniques and efficient data structures. We selected and adapted a camera calibration technique that was integrated into the system. Images from a capturing device was both used as a backdrop image and as input for the recognition and calibration subsystem.

Finally, to prove the functionality and flexibility of our system, a number of examples were created that illustrates possible application areas.

7.2 Future Work

The future brings faster computers, higher quality cameras, and HMDs with better displays and lighter weight. All these advances will improve the result of the developed software but there are also software issues that need to be resolved.

Better cameras and higher resolution The quality of the cameras used in this thesis has been low, and the level of manual customization of the camera properties has also been inadequate. It would have been an advantage to have a camera with higher resolution than 640x480 and higher

quality without using too much of the computer's resources while capturing. An example of better equipment that could have been used is the HMDs that the company Triviso [5] provide, with built in stereo cameras. By using a higher resolution input image to find the markers increases the precision of locating the center of markers, but by increasing the resolution the need for more computing power also increases.

Enhancing the recognition of markers The thresholding algorithm is very sensitive to the lighting conditions and is not completely robust. To improve this part of the software, it is necessary to implement a more adaptive thresholding algorithm that will operate successfully even with a moderate amount of highlighting in the scene.

Markers Creating markers that have an even more diffuse property than the current markers is necessary to reduce the highlighting issues that we experience. The current implementation of markers is not ideal. The software is dependent on seeing all of the markers at the same time, so it is not possible to only study a small part of a model. Maybe a grid of different symbols would make it possible to calibrate and only show the part of the model represented by that part of the grid.

Stereo capturing By using a stereo camera configuration, it would be possible to resolve the "in front or behind?" problem mentioned earlier, and to cast shadows correctly in the scene. In addition, the illusion of 3D would be enhanced since the background image would now be in stereo.

Shadows It would be nice to implement shadows. Shadows increase the illusion of depth. The current problem with shadows, and the reason for not implementing it in this iteration, is that the background image does not contain any depth information. This causes shadows to be incorrectly cast on objects. For example shadows will not wrap around round objects. Depth information is possible to acquire with stereo capturing.

Multiple Markers It would be easy to extend the software with support for multiple markers. By using either different colors or patterns inside the marker squares to differentiate between the markers, it would be possible to have more than one object to interact with. This would be useful for trying out different configurations of house placements in a landscape with one large marker and several small ones inside of the larger one.

Collaborative Multiple users using HMD and looking at the same markers, one user should be able to show all the others what he or she is focusing on. The users may see the model either from the point of view of the directing user, or from their own point of view.

Interaction In addition to be able to move multiple markers around, it would be nice to interact with the models without using a keyboard or a mouse. For example to rotate the model in place or to change a model's texture and color by some kind of hand gesture or speech.

Acknowledgments

I would like to thank Tyge Løvset, Fredrik Manne, Kåre P. Villanger, Lars Helge Stien, Inge Eliassen, Endre Lidal, Christian Michelsen Research, my parents, and Katrine.

Appendix A

Algorithms

Algorithm 1 Image Thresholding

```
1: function THRESHOLD(image)
2:     for all pixels ∈ image do
3:         r ← red-component of pixel
4:         g ← green-component of pixel
5:         b ← blue-component of pixel
6:         if isWhite(r, g, b) then
7:             pixel ← WHITE
8:         else if isRed(r, g, b) then
9:             pixel ← RED
10:        else if isBlue(r, g, b) then
11:            pixel ← BLUE
12:        else if isYellow(r, g, b) then
13:            pixel ← YELLOW
14:        else
15:            pixel ← BLACK
16:        end if
17:    end for
18: end function
```

Algorithm 2 Finding Connected Components

```
1: function findConnectedComponents(image)
2:     I ← Integer array with dimensions as image, all elements set to -1
3:     C ← indexable list of Component objects.
4:     currentindex ← 0

5:     for all pixels ∈ image do
6:         x ← x-coordinate of pixel
7:         y ← y-coordinate of pixel
8:         if pixel ≠ BLACK then
9:             label(C, I, image, x, y, currentindex)
10:        end if
11:        if y > 0 and x > 0 then
12:            if (I(x, y − 1) ≠ −1) and (I(x − 1, y) ≠ −1) then
13:                if image(x, y − 1) = image(x − 1, y) then
14:                    c1 ← C[I(x, y − 1)].getRepresentative()
15:                    c2 ← C[I(x − 1, y)].getRepresentative()
16:                    if c1 ≠ c2 then
17:                        c ← union(c1, c2)
18:                        C[I(x, y − 1)].setRepresentative(c)
19:                        C[I(x − 1, y)].setRepresentative(c)
20:                    end if
21:                end if
22:            end if
23:        end if
24:    end for
25:    for all components ∈ C do
26:        if component.getRepresentative() ≠ component then
27:            Remove component from list.
28:        else if size of component is too large or too small then
29:            Remove component from list.
30:        end if
31:    end for
32:    return C
33: end function

34: function union(c1, c2)
35:    c1.minY ← min(c1.minY, c2.minY)
36:    c1.maxY ← max(c1.maxY, c2.maxY)
37:    c1.minX ← min(c1.minX, c2.minX)
38:    c1.maxX ← max(c1.maxX, c2.maxX)
39:    c1.x ← c1.x + c2.x
40:    c1.y ← c1.y + c2.y
41:    c1.size ← c1.size + c2.size
42:    c2.setRepresentative(c1)
43:    return c1
44: end function
```

Algorithm 3 Labeling of pixels

```
1: function LABEL(C, I, image, x, y, currentindex)
2:     pixel ← image(x, y)
3:     if pixel ∈ {WHITE, RED, BLUE, YELLOW} then
           ▷ The following if lines compares the color of two pixels within the
           bounds of the image.
4:         if x > 0 and pixel = image(x − 1, y) then
5:             I(x, y) ← I(x − 1, y)
6:             C[I(x, y)].addPixel(x, y)
7:         else if x > 0 and y > 0 and pixel = image(x − 1, y − 1) then
8:             I(x, y) ← I(x − 1, y − 1)
9:             C[I(x, y)].addPixel(x, y)
10:        else if y > 0 and pixel = I(x, y − 1) then
11:            I(x, y) ← I(x, y − 1)
12:            C[I(x, y)].addPixel(x, y)
13:        else
               ▷ A new component has been found.
14:            I(x, y) ← currentindex
15:            currentindex ← currentindex + 1
16:            C[I(x, y)] = new Component(x, y, image, labels)
17:        end if
18:    end if
19: end function

20: function ADDPIXEL(component, x, y)
21:    component ← component.getRepresentative()
22:    component.addPixel(x, y)
23: end function

24: function NEWCOMPONENT(image, I, x, y)
25:    component = new Component(x, y)
26:    component.setColor(image(x, y))
27:    component.setIndex(I(x, y))
28:    return component
29: end function
```

Algorithm 4 Component Separation

```
1: function SEPARATE(C)
2:     w, r, b, y ← empty lists
3:     for all components ∈ C do
4:         if component.color = WHITE then
5:             w.add(component)
6:         else if component.color = RED then
7:             rb.add(component)
8:         else if component.color = BLUE then
9:             rb.add(component)
10:        else if component.color = YELLOW then
11:            y.add(component)
12:        end if
13:    end for
14:    return w, rb, y
15: end function
```

Algorithm 5 Identify Red and Blue Markers

```
1: function identifyRedAndBlueMarkers(w, rb)
2:     markerlist ← empty list of Markers
3:     for all components ∈ w do
4:         white ← component from w
5:         for all components ∈ rb do
6:             redorblue ← component from rb
7:             if inside(white, redorblue) then
8:                 markers.add(merge(white, redorblue))
9:                 rb.remove(redorblue)
10:                breakinnerforloop
11:            end if
12:        end for
13:    end for
14:    return markerlist
15: end function

16: function merge(c1, c2)
17:    marker ← new Marker
18:    marker.minY ← min(c1.minY, c2.minY)
19:    marker.maxY ← max(c1.maxY, c2.maxY)
20:    marker.minX ← min(c1.minX, c2.minX)
21:    marker.maxX ← max(c1.maxX, c2.maxX)
22:    marker.size ← c1.size + c2.size
23:    if c2 = RED then
24:        marker.color = RED
25:    else if c2 = BLUE then
26:        marker.color = BLUE
27:    end if
28:    (x, y) = c1.calculateCenter()
29:    marker.setCenter(x, y)
30:    return marker
31: end function
```

Algorithm 6 Identify Markers Inside Yellow Components

```
1: function identifyMarkersInsideYellowCmpnts(markerlist, y)
2:     markers ← empty list of Markers
3:     for all components ∈ y do
4:         yellow ← component from y
5:         for all markers ∈ markerlist do
6:             m ← marker from markerlist
7:             if inside(m, yellow then
8:                 markers.add(m)
9:             end if
10:        end for
11:    end for

12:    if markers.size > 4 then
13:        if markers contain repeated members then
14:            remove redundant members
15:        end if
16:    end if

17:    if markers contain 1 red and 3 blue markers then
18:        return markers
19:    else
20:        return null
21:    end if
22: end function
```

Algorithm 7 Sorting Markers

```
1: function SORT(markers)
2:     a ← markers[0]
3:     b ← markers[1]
4:     c ← markers[2]
5:     d ← markers[3]
6:     if isIntersetionInsideSquare(a, b, c, d) then
7:         if isOnRight(a, b, c) then
8:             markers ← {a, c, b, d}
9:         else
10:            markers ← {a, d, b, c}
11:        end if
12:    else if isIntersetionInsideSquare(a, c, b, d) then
13:        if isOnRight(a, c, b) then
14:            markers ← {a, b, c, d}
15:        else
16:            markers ← {a, d, c, b}
17:        end if
18:    else if isIntersetionInsideSquare(a, d, b, c) then
19:        if isOnRight(a, d, b) then
20:            markers ← {a, b, d, c}
21:        else
22:            markers ← {a, c, d, b}
23:        end if
24:    else
           ▷ The markers do not create a square. These are probably not the
           markers from the marker square.
25:    end if
26:    return markers
27: end function
```

Bibliography

[1] Artoolkit. http://www.hitl.washington.edu/artoolkit/.

[2] Buddha model from inspeck. http://www.inspeck.com.

[3] Lightweight java gaming library. http://www.lwjgl.org/.

[4] Mixed reality rendering with openrt. http://graphics.cs.uni-sb.de/MR/.

[5] Trivisio. http://www.trivisio.com/mixed-reality.html.

[6] Edward Angel. *Interactive Computer Graphics, A top down approach using OpenGL*. Addison Wesley, third edition, 2003.

[7] OpenGL Architecture Review Board. *OpenGL 1.4 Reference Manual*. Addison Wesley Professional, fourth edition, 2004.

[8] OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *The OpenGL Programming Guide - The Redbook: The Official Guide to Learning OpenGL, Version 1.4*. Addison-Wesley, fourth edition, 2003.

[9] David H. Brainard. Bayesian method for reconstructing color images from trichromatic samples. *IS&T's 47th Annual Conference/ICPS*, pages 375–380, 1994.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. The MIT Press, second edition, 2001.

[11] Paul Debevec. Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *Proceedings of SIGGRAPH 98*, pages 189–198. SIGGRAPH, 1998.

[12] Jiann der lee. A fast geometrical approach to camera extrinsic parameters. *Computers Math. Applic.*, 32(12):93–100, 1996.

[13] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practice*. Addison Wesley, second edition, 1997.

[14] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Prentice Hall, second edition, 2002.

[15] E. Guillou, D. Meneveaux, E. Maisel, and K. Bouatouch. Using vanishing points for camera calibration and coarse 3d reconstruction from a single image. *The Visual Computer*, 16:396–410, 2000.

[16] USB Implementers Forum Inc. Universal serial bus. http://www.usb.org.

[17] David C. Lay. *Linear Algebra and its Applications*. Addison Wesley, second edition, 1998.

[18] Xiaoqiao Meng and Zhanyi Hu. A new easy camera calibration technique based on circular points. *Pattern recognition*, 36:1155–1164, 2003.

[19] Paul Milgram and Herman Colquhoun Jr. *A Taxonomy of Real and Virtual World Display Integration*, chapter 1, pages 1–16. Ohmsha(Tokyo) & Springer Verlag(Berlin), 1999.

[20] Mark W. Powell, Sudeep Sarkar, and Dmitry Goldgof. A simple strategy for calibrating the geometry of light sources. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(9):1022–1027, September 2001.

[21] B. Prescott and G. F. McLean. Line-based correction of radial lens distortion. *Graphical models and image processing*, 59(1):39–47, January 1997.

[22] J. Schmidt, I. Scholz, and H. Niemann. Placing arbitrary objects in a real scene using a color cube for pose estimation. In B. Radig and S. Florczyk, editors, *Pattern Recognition, 23rd DAGM Symposium*, pages 421–428, Munich, Germany, September 12-14 2001. Springer-Verlag, Berlin, Heidelberg, New York. ISBN 3-540-42596-9. Lecture Notes in Computer Science 2191.

[23] Jochen Schmidt, Heinrich Niemann, and Sebastian Vogt. Dense disparity maps in real-time with an application to augmented reality. In *Proceedings Sixth IEEE Workshop on Applications of Computer Vision (WACV 2002)*, pages 225–230, Orlando, FL USA, December 3-4 2002. IEEE Computer Society. ISBN 0-7695-1858-3.

[24] Milan Sonka, Vaclav Hlavac, and Roger Boyle. *Image Processing Analysis and Machine Vision*. Thomson-Engineering, 2 edition edition, 1998.

[25] Sony. Eyetoy. http://www.eyetoy.com.

[26] Sun. Java media framework. http://java.sun.com/products/java-media/jmf/index.jsp.

[27] Thorsten Thormählen, Hellward Broszio, and Ingolf Wassermann. Robust line-based calibration of lens distortion from a single view. *Proceedings of Mirage 2003*, pages 105–112, March 2003.

[28] Roger Y. Tsai. A versatile camera calibration technique for high-accuracy 3d machine vision metrology using off-the-shelf tv cameras and lenses. *IEEE Journal of Robotics and Automation*, RA-3(4):323–344, 1987.

[29] Richard S. Wright and Michael Sweet. *OpenGL Superbible*. Waite Group Press, second edition, 1999.

[30] Zhengyou Zhang. A flexible new technique for camera calibration. Technical Report MSR-TR-98-71, Microsoft Research, 1998. URL `http://research.microsoft.com/~zhang`.