Development of a Stereo Vision Mixed Reality Framework

by

Christiaan Johannes Hendrik le Roux

Thesis presented in partial fulfilment of the requirements for the degree Master of Science in Engineering at Stellenbosch University

Supervisor: Dr. Gert-Jan van Rooyen
Department of Electrical and Electronic Engineering

March 2012

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

March 2012

Abstract

Augmented reality is a fairly young research field, still in an infancy stage at Stellenbosch University. Since this is one of the first augmented reality projects, one goal is to present a theoretical study of augmented reality. This study is given in the literature study, along with a review of the available development solutions.

While there are various tools available with which one can create marker-based augmented reality applications, these tools are not meant for testing new techniques and algorithms in an augmented or mixed reality. The remaining goals of this project is to create a platform for the rapid design of augmented reality applications, and to expand the capabilities of this platform beyond marker-based augmented reality.

In this project we present the design and implementation of a pragmatic mixed-reality framework capable of a wider variety of applications. A design is shown where marker tracking can be used alongside other computer vision techniques to design new applications. The framework utilises stereo cameras to find the position of real world objects, and a 3D display to make the mixed reality environment as immersive as possible.

Proof of concept test applications built with the framework are presented. Colour based techniques are used to find a user's hand and create a virtual representation of it. This allows the user to interact with a virtual object in an augmented reality scene by 'touching' it with her hand.

Samevatting

Toegevoegde realiteit is 'n jong navorsingsveld by Universiteit Stellenbosch. Aangesien hierdie een van die eerste projekte is wat fokus op toegevoegde realiteit, is 'n teoretiese studie van toegevoegde realiteit as 'n doel gestel. Dit word verskaf in die literatuurstudie, tesame met 'n oorsig oor bestaande oplossings vir die ontwikkeling van toegevoegde realiteit sagteware.

Bestaande oplossings is gefokus op die ontwikkeling van merker-gebaseerde toegevoegde realiteit, maar los min ruimte vir die toets van nuwe tegnieke toepasbaar op die veld. Dit lei tot die oorblywende doelwitte van die projek: om 'n platform te ontwerp vir die ontwikkeling van merker-gebaseerde toegevoegde realiteit programme, asook om die platform uit te brei.

Ons lewer 'n pragmatiese ontwikkelingsraamwerk wat dit moontlik maak om 'n verskeidenheid nuwe toegevoegde realiteit programme te ontwikkel. Die raamwerk is ontwerp sodat die ontwikkelaar merkers saam met ander rekenaarvisie tegnieke kan gebruik om sagteware te skep. Stereo kameras word gebruik om die posisie van werklike voorwerpe te vind. Die raamwerk maak ook gebruik van 'n 3D skerm om virtuele objekte te vertoon.

Toetsprogramme gebou as 'n bewys van die konsep, word vertoon en bespreek. 'n Kleur-gebaseerde tegniek word gebruik om 'n gebruiker se hand te vind, en 'n virtuele voorstelling van die hand word geskep. Die gebruiker kan virtuele voorwerpe laat reageer deur dit met haar hand aan te raak.

Acknowledgements

I would like to express my sincere gratitude to the following people and organisations:

- My parents and family for your ongoing love and support.
- Nicolene for always being there.
- The guys at Bazooka Games.
- The Goblin XNA and GameDev.net communities.
- Dr. Gert-Jan van Rooyen for the advice, the expert guidance and the amazing amount of effort.

Contents

Acknowledgements **iv**

Contents **v**

List of Figures **viii**

List of Tables **xi**

List of Listings **xii**

Nomenclature **xiii**

Abbreviations xiii

Mathematical Symbols xiv

1 Introduction **1**

1.1 Background 1

1.2 Objectives 2

1.3 Contributions 2

1.4 Overview 3

2 Literature Review **5**

2.1 Introduction 5

2.2 A Review of PC Augmented Reality Solutions 6

2.3 DirectX 13

2.4 Single-View Geometry 21

2.5 Multiple View Geometry 30

2.6 NVidia 3D Driver Basics 39

3 System Design **44**

3.1 Introduction . . . 44
3.2 Conceptual Design . . . 44
3.3 Framework Overview . . . 47
3.4 Eye . . . 50
3.5 Visual Cortex . . . 51
3.6 Graphics . . . 55
3.7 Utilities . . . 57

4 Detailed Design **60**
4.1 Introduction . . . 60
4.2 Image Extractors . . . 61
4.3 Stereoscopic Vision . . . 71
4.4 Stereoscopic Display . . . 74

5 Tests and Implementation **79**
5.1 Introduction . . . 79
5.2 Performance Tests . . . 79
5.3 Implementing an Augmented Reality Application . . . 88
5.4 Implementing a 3D mixed-reality interface . . . 94
5.5 Expanded Augmented Reality Example . . . 100
5.6 Implementation Issues . . . 105

6 Conclusions and Recommendations **108**
6.1 Summary . . . 108
6.2 Further Research . . . 109

Bibliography **111**

Appendices **1**

A Comparison between AR Solutions **2**
A.1 ARTag and ARToolkit . . . 2
A.2 CPU Usage for ARToolkit, GoblinXNA and MxRFramework . . . 3

B Development Computer Specifications **4**

C Background Subtraction Test Data **5**
C.1 Dataset . . . 5

D Background Subtraction Test Results **14**
D.1 Image Extractor Performance 14
D.2 Extractor Output Images 18

E Framework demonstration application **30**
E.1 DVD content 30

List of Figures

2.1 Milgram's Reality Virtuality Continuum [1]. 6
2.2 AR application examples. 7
2.3 A Unifeye design example. 8
2.4 Goblin XNA application examples. 11
2.5 The ARRacer scene graph. 11
2.6 The ARRacer program flow. 12
2.7 The relationship between the application and the graphics hardware. 15
2.8 DirectX virtual camera model. 16
2.9 Presenting the front buffer causes addresses to swap instantly. . . . 17
2.10 The pinhole camera model. Figure from [2]. 21
2.11 The image plane becomes a projection plane between the pinhole and the object. Figure from [2] 22
2.12 The action of homography. Image from [2]. 26
2.13 Radial component of the distortion model. Image from [2]. 27
2.14 Tangential component of the distortion model. Image from [2]. . . . 29
2.15 An estimate of the horizontal human field of vision, as found in [3]. 32
2.16 Relationship between two imagers [2]. 33
2.17 Frontal parallel configuration. Figure reproduced from [2]. 35
2.18 Finding parallax from a top-down view. 40
2.19 The relationship between parallax and vertex depth. Image reproduced from [4]. 41
2.20 The stereoscopic frustum. 42

3.1 Framework overview. 48
3.2 The Eye class diagram. 50
3.3 Visual Cortex use case diagram. 51
3.4 The marker tracker and depth mapper class diagrams. 52

3.5 The relationship between ExtractorController class and the extractors. . . . 53
3.6 The ImageExtractor class diagram. . . . 54
3.7 The Body2DExtractor class diagram. . . . 55
3.8 The Body3DExtractor class diagram. . . . 56
3.9 Graphics class use case diagram. . . . 57
3.10 Using MRMeshInstance in an application. . . . 58
3.11 Relationship between mesh classes. . . . 59

4.1 The background subtraction process. . . . 62
4.2 Shadows in foreground. . . . 65
4.3 Shadow suppression. . . . 66
4.4 Finding a bounding shape. . . . 68
4.5 Displaying a pair of stereo images. . . . 70
4.6 Block-matching features with two images. Figure from [2]. . . . 73
4.7 Finding a depth map with MxRFramework 74
4.8 Stereo frame format. . . . 76
4.9 Displaying a pair of stereo images. . . . 78

5.1 Samples from the artificial extractor test set. . . . 82
5.2 The accuracy of BoundingBoxExtractor. . . . 86
5.3 Comparing depth from the marker tracker and the disparity map. . . . 87
5.4 Comparison between the distance from the marker tracker and disparity map. . . . 88
5.5 Depth error from the marker tracker and disparity map. . . . 89
5.6 AR application flow chart. . . . 90
5.7 AR application written with framework. . . . 91
5.8 Basic Augmented Reality Application Comparison. . . . 91
5.9 Average CPU and Memory Usage. . . . 92
5.10 AR solutions processing time comparison. . . . 93
5.11 3D Vision Based UI 94
5.12 MR application flowchart. . . . 96
5.13 Flowchart for the worker thread. . . . 97
5.14 MR application screen shot. . . . 98
5.15 Flowchart for the main thread. . . . 99
5.16 AR scene placed on the marker. . . . 101

5.17 Finding the bounding sphere using the disparity map to find depth. (b) The bounding sphere is illustrated with a rendered sphere. (a) The depth of the bounding sphere is found from the disparity map. 104
5.18 Markerless AR user interaction. 104
5.19 Perception along the Reality-Virtuality Continuum. Figure from [5]. 105

A.1 Marker tracker comparison in varying light conditions: ARToolkit performance with low and high threshold values are shown in the left and middle images. The right image shows ARTag performance under the same conditions. Image courtesy of [6]. 2
A.2 Comparing AR solution CPU usage. 3

C.1 Background image. 5
C.2 Test images. 6
C.3 Test images (continued). 7
C.4 Test images (continued). 8
C.5 Test images (continued). 9
C.6 Ground truth foreground and background. 10
C.7 Ground truth foreground and background (continued). 11
C.8 Ground truth foreground and background (continued). 12
C.9 Ground truth foreground and background (continued). 13

D.1 The accuracy of ForegroundExtractor1. 15
D.2 The accuracy of ForegroundExtractor2. 16
D.3 The accuracy of ColourExtractor. 17
D.4 Foreground extractor output images. 18
D.5 Foreground extractor output images (continued). 20
D.6 Foreground extractor output images (continued). 21
D.7 Foreground extractor output images (continued). 22
D.8 Foreground extractor output images (continued). 23
D.9 Foreground extractor output images (continued). 24
D.10 Foreground extractor output images (continued). 25
D.11 Foreground extractor output images (continued). 26
D.12 Foreground extractor output images (continued). 27
D.13 Colour extractor output images. 28
D.14 Colour extractor output images (continued). 29

List of Tables

5.1 Processing time of framework algorithms ([1]OpenCV based functions). 80
5.2 Accuracy average for the dataset. 85
5.3 Error in depth. 88

A.1 ARTag and ARToolkit processing times using a Pentium 4 3.0 GHz PC. Table reproduced from [7]. 2

List of Listings

4.1 Using a stereo rig with MxRFramework. 75
4.2 The stereo off-screen surface. 75
4.3 Recognising a surface as a stereo pair. 77
4.4 Storing the left and right image to the off-screen stereo surface. 77
5.1 Calibrating cameras and finding the disparity map. 98
5.2 Initializing a basic marker-based AR application. 101
5.3 Main-loop code for a basic marker-based AR application. 102
5.4 Extractor and depth map code example. 103

Nomenclature

Abbreviations

Software Libraries

ALVAR	A Library for Virtual and Augmented Reality
ARToolkit	Augmented Reality Toolkit
CImg Library	Cool Image Library
D3D	Direct3D
DWARF	Distributed Wearable Augmented Reality Framework
NVAPI	NVidia Application Programming Interface
OpenCV	Open Source Computer Vision
OpenGL	Open Source Graphics Library
OpenSG	OpenSceneGraph
OSGART	OpenSceneGraph and ARToolkit
VXL	Vision X Library

Misc

2D	Two Dimensional
3D	Three Dimensional
6DOF	Six Degrees of Freedom
AGP	Advanced Graphics Port
API	Application Programming Interface
AR	Augmented Reality
blit	Block Image Transfer,
CPU	Central Processing Unit

DDR	Double Data Rate
GPU	Graphics Processing Unit
GS	Greyscale
HAL	Hardware Abstraction Layer
HUD	Head Up Display
HSV	Hue Saturation Value
Hz	Hertz
I/O	Input/Output
LCD	Liquid Crystal Display
MIDI	Musical Instrument Digital Interface
MR	Mixed Reality
PC	Personal Computer
PCI	Peripheral Component Interconnect
PCI-E	Peripheral Component Interconnect Express
QVGA	Quarter Video Graphics Array
RAM	Random Access Memory
RGB	Red Green Blue
SAD	Sum of Absolute Differences
SDK	Software Development Kit
USB	Universal Serial Bus
VGA	Video Graphics Array
VR	Virtual Reality
XML	Extensible Markup Language

Mathematical Symbols

View Geometry

B	interaxial distance
d	disparity
c_x	optical x-axis offset

c_y	optical y-axis offset
F	physical focal length
f	focal length(in pixels)
f_x	focal length in x direction (in pixels)
f_y	focal length in y direction (in pixels)
$g(r)$	radial distortion
k_n	radial distortion constants
p_n	tangential distortion constants
s_x	pixel length
s_y	pixel height
x_c	corrected x coordinate
y_c	corrected y coordinate
Z	depth
$\mathbf{B}$	interaxial distance (vector)
$\mathbf{E}$	essential matrix
$\mathbf{F}$	fundamental matrix
$\mathbf{I}$	identity matrix
$\mathbf{H}$	homography
$\mathbf{M}$	camera intrinsics matrix
$\mathbf{M_{rect\_i}}$	rectified intrinsics matrix for camera i, where i can be l or r
$\mathbf{P}$	perspective projection
$\mathbf{R}$	rotation matrix
$\mathbf{T}$	translation matrix
$\mathbf{W}$	transformation matrix

DirectX and Stereoscopic Rendering

B	interaxial distance
h_B	back buffer height
P	parallax
S	screen depth
w_B	back buffer width

W	vertex depth
Z	frustum depth

Extractors

α_h	bounding box test ratio (height)
α_w	bounding box test ratio (width)
$\Delta R(x, y)$	red value difference between input image and background
$\Delta G(x, y)$	green value difference between input image and background
$\Delta B(x, y)$	blue value difference between input image and background
ϵ	standard deviation
θ_G	edge direction
A	Morphology kernel
d_E	disparity from the extractors
D_R	ray direction
G	edge gradient
h_B	height of a bounding box in pixels
h_O	height of a foreground object in pixels
K_n	depth constants, with $n \in \{1, 2\}$
O_R	ray origin
O_C	origin camera
T_C	total correct foreground pixels found
T_I	total pixels incorrectly found as foreground
V_C	percentage correct pixels identified
V_I	percentage pixels incorrectly identified as foreground
w_B	width of a bounding box in pixels
w_O	width of a foreground object in pixels
Z_M	depth found with the marker tracker
Z_D	depth found with the disparity map
I	input image
$I(x, y)$	pixel located at coordinates (x, y) on image I
$I_R(x, y)$	red value of pixel at (x, y) on image I

$I_G(x, y)$	green value of pixel at (x, y) on image I
$I_B(x, y)$	blue value of pixel at (x, y) on image I
$I_H(x, y)$	hue value of pixel at (x, y) on image I
$I_S(x, y)$	saturation value of pixel at (x, y) on image I
$I_V(x, y)$	brightness value of pixel at (x, y) on image I
$B(x, y)$	pixel located at coordinates (x, y) on the background image
$F(x, y)$	pixel located at coordinates (x, y) on the extractor output image

Misc

Δ	difference
s	second(s)
t	time

Chapter 1

Introduction

1.1 Background

An augmented reality (AR) environment is one in which virtual entities are merged with the physical world in real time. AR research is a fairly young field, and at the time of writing, augmented reality research at Stellenbosch University is still in an infancy stage. At the outset of this project, augmented reality enjoys increased popularity, exemplified by the high number of monthly downloads of the most popular tool: ARToolkit [8]. This type of marker-based augmented reality has been further expanded in solutions like GoblinXNA and OSGART, that allow the creation of more complex AR scenes. Other solutions became available that attempt to simplify the authoring environment, making it possible to create AR scenes without any coding knowledge.

All these tools focus on giving a developer access to an existing, proven real-world estimation technique, marker tracking, and allow them to experiment with the creation of an AR world bounded by the limits of marker tracking . The problem is that markers are artificial objects introduced into a scene, and that interaction takes place between virtual entities and the markers, and no other real world objects. However, markers are proven to work as a trusted reference[9] and can be used in various conditions. Currently there are no real alternatives that can accomplish this. Nevertheless, computer vision, and in particular image processing, is a research field that currently enjoys rapid advances, with many new algorithms and approaches that are applicable to augmented reality. At the time of writing no AR solutions exist that offer a simple way to include new algorithms.

The main goal of this project was to build an expandable AR development platform from which further research can be done. This project focuses on giving developers a simple interface to test new ideas applicable to augmented reality or human-computer interaction. This framework should give developers a wider range of possible physical world interpretation tools, and more freedom in the type of application that can be developed than what is currently possible with existing solutions.

With the release of powerful mobile devices capable of rendering 3D scenes, much of the focus of augmented reality research has shifted to mobile devices (at the time of writing). However, personal computers (PCs) have the benefit of higher possible processing power and high power dedicated graphics devices. Consequently, this project focusses solely on the PC based AR development.

1.2 Objectives

At the outset of this study, it was required that the following goals be fulfilled:

- A theoretical study of augmented reality, and a review of development solutions must be done.
- A platform for the rapid design of augmented reality applications must be developed.
- Capabilities must be expanded beyond marker-based augmented reality.

1.3 Contributions

Through the course of this project we have implemented a framework that can be used to create augmented reality applications similar to existing marker-based AR applications. This framework was then expanded to offer new functionality. The new contributions of this project are:

A stereo correspondence tool with a simple interface. The framework contains tools to find the depth of a scene with a stereo camera rig. A simple interface to this tool is supplied, allowing the developer to concentrate on creating the content of her augmented reality world.

Markerless interaction. The framework can create virtual representations of real-world objects in 2D and 3D space, making markerless interaction possible.

A flexible approach to computer vision. The framework was designed to be expandable. It is possible to include a variety of new computer vision techniques, and use it alongside algorithms and trackers already included in the framework.

A development framework for the creation of a wide variety of mixed-reality applications. The framework allows a developer to create various kinds of mixed reality applications. Examples are given in chapter 5.

Stereoscopic vision based mixed reality[1]. The framework makes use of a 3D monitor. Using this monitor we created the illusion of virtual objects floating in front of the screen. With the framework we then created an application in which it was possible to interact with these "floating" virtual objects.

1.4 Overview

Chapter 2 provides the required background information for this study. Section 2.2 discusses the term augmented reality. As motivation for the creation of the framework, the existing augmented reality design solutions are then reviewed, and the resulting trends that are developing in the field are shown.

The rest of chapter 2 describes background and models used in the design of the framework. Section 2.3 introduces DirectX, and describes the rendering process. Background on the NVidia stereoscopic vision driver is also provided.

The mathematical background needed for the computer vision algorithms used in the framework is also discussed, and the models used for single and stereoscopic cameras are presented.

After the state of augmented reality design solutions has been discussed, chapter 3 discusses what we attempt to achieve with the framework design in this project, i.e. what the framework offers the user that is not present in current solutions. We discuss the design choices and functionality of this project. The chapter presents the design of each component of the framework,

[1]In this project, mixed reality refers to any environment that contains physical and virtual entities. This is further discussed in chapter 2.

as well as the integration of the components.

Chapter 4 presents the design of the computer vision algorithms included in the framework. The completed project contains functions for stereo correspondence, colour-based object detection, foreground detection, and contouring. In this chapter we discuss the design of each of these algorithms in detail. We also show how the stereoscopic display is utilised. The creation of virtual 2D and 3D entities representing the object is also discussed.

In chapter 5 the functionality and performance of the framework is tested. The design of applications created with the framework are discussed. These applications are used as demonstration of the design process and versatility of the framework.

Chapter 5.6 discusses some of the issues faced during the implementation of the framework.

Chapter 6 concludes the work with a discussion on contributions made and recommendations regarding the path forward and the expansion of the framework.

Chapter 2

Literature Review

2.1 Introduction

This chapter presents background information on the various topics covered in this work. Some key concepts upon which the rest of the work is built are introduced. An understanding of the topics in this chapter is imperative to understand not only the focused design in the project, but also the motivation for the project and the reasoning behind design choices.

In section 2.2 we discuss the history of augmented reality, and give an overview of the current state of desktop PC-based augmented reality. In the next section a basic overview of DirectX is given. The functionality of some DirectX components and methods that are later used in the design of this project are given.

Section 2.4.2 discusses the modeling of the real-world camera as used in the framework. This model is expanded for stereoscopic cameras in the next section, which discusses the coordinate system used and how we describe the relationship between the cameras.

After some understanding of DirectX and multiple view geometry has been provided. Section 2.6 presents the Nvidia stereoscopic driver. The section covers the rendering of 3D scenes and the design considerations associated with using the driver.

2.2 A Review of PC Augmented Reality Solutions

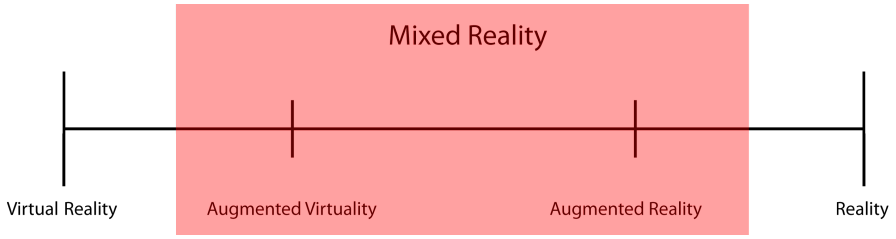


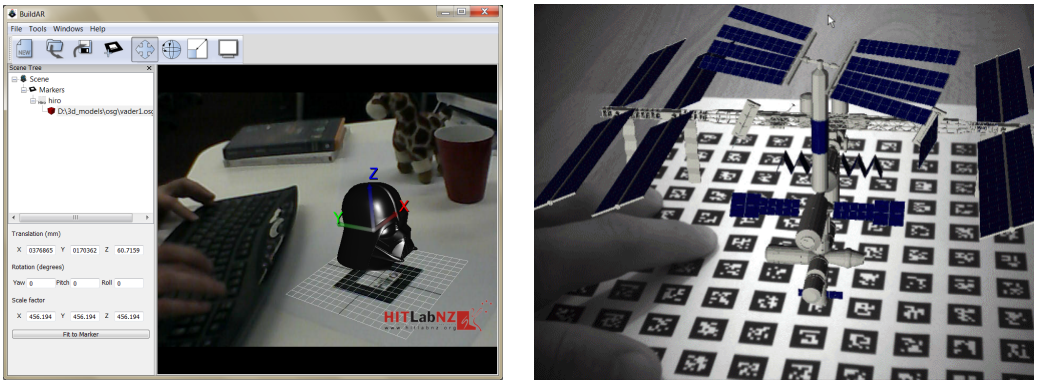
Figure 2.1: Milgram's Reality Virtuality Continuum [1].

The term augmented reality was originally used by Ivan Sutherland in the 1960s and refers to an environment where virtual objects are dynamically superimposed over real-world objects. He demonstrated this concept by displaying simple computer-generated wireframes through a basic head-mounted display [10], but the limits of technology limited interest and research. Due to the rapid progress of technology, the past two decades have seen growing interest in the field of augmented reality. Researchers have put strong emphasis on real-world object tracking and augmented display [11]. However, this technology is still new, and applications are few and not widely used. In order to hasten progress, multiple solutions and tools were developed for aiding design of AR-programs, each with its own focus area and strong points. We will now look at some of the most prominent solutions.

2.2.1 Toolkits

AR systems require some indication of where the augmentation of virtual objects should take place, i.e. a reference point in the real-world. As AR is inherently real time, this is typically accomplished using the simplest form of marker [12]. These markers have a unique pattern, which, if visible to the AR camera, can be identified. We refer to these markers as fiducial markers, where the term "fiducial" indicates that it is used as a trusted reference. When the marker is found its pose is calculated and virtual objects are superimposed onto the marker. Figure 2.2 shows examples of fiducial marker-based

augmented reality. As alternatives to fiducial marker tracking, invisible markers [13], markerless object tracking or scene mapping [14, 15, 16] and GPS [17] are currently under active research. Markerless tracking would be ideal, as it would mean no artificial identification method has to be added to the scene. However, markerless AR has not advanced enough to perform as well as marker-based solutions [12].



(a) (b)

Figure 2.2: (a) BuildAR, an ARToolkit-based application. (b) A scene built with ARTag. Image from [6].

ARToolkit [8] and ARtag [6] make it possible to overlay virtual imagery onto the real-world using fiducial marker tracking. At the time of writing, both of these toolkits use OpenGL to render the virtual objects. The fiducial marker tracking method differs between the libraries, each with its own strong and weak points [18]. ARToolkit uses binary thresholding and finds all quadrilateral contours in a scene that could possibly be marker boundaries. Fast template matching is then done (by correlation) on these regions in order to identify the marker [19, 20]. If the edges of the image are not visible, the marker will not be recognised as a quadrilateral contour (polygon with four sides), and will therefore not be recognised as a marker. Another possible problem occurs in varying lighting conditions, where the binary threshold could be too high or low to recognise the contours. ARToolkit is the most downloaded and searched augmented reality toolkit on the Internet [21]. At the time of writing, older versions of ARToolkit are available under the GNU general public license, while newer versions and support are only available commercially.

ARTag does not use correlation for matching, but encodes the content of a marker in a 36-bit binary code and matches it against a database of known marker codes. The detection algorithm is described in [7]. It does not use binary thresholding, and therefore performs better with varying light in a scene. False detection rates are also vastly improved over ARToolkit [7]. ARTag can identify a marker even if the border of the marker is slightly occluded. It does, however, have a longer processing time per frame than ARToolkit's detection method[1] [7].

2.2.2 Authoring Tools

BuildAR is essentially a graphical interface to ARToolkit, allowing a user to link a virtual 3D scene to a marker. Multiple scenes and markers can be used at the same time for building simple AR scenes [22]. The Tiles project is a generic model with interaction techniques for creating a tangible augmented reality interface. Interaction takes place using cardboard markers to represent objects and actions, and the package uses ARToolkit for tracking [11].

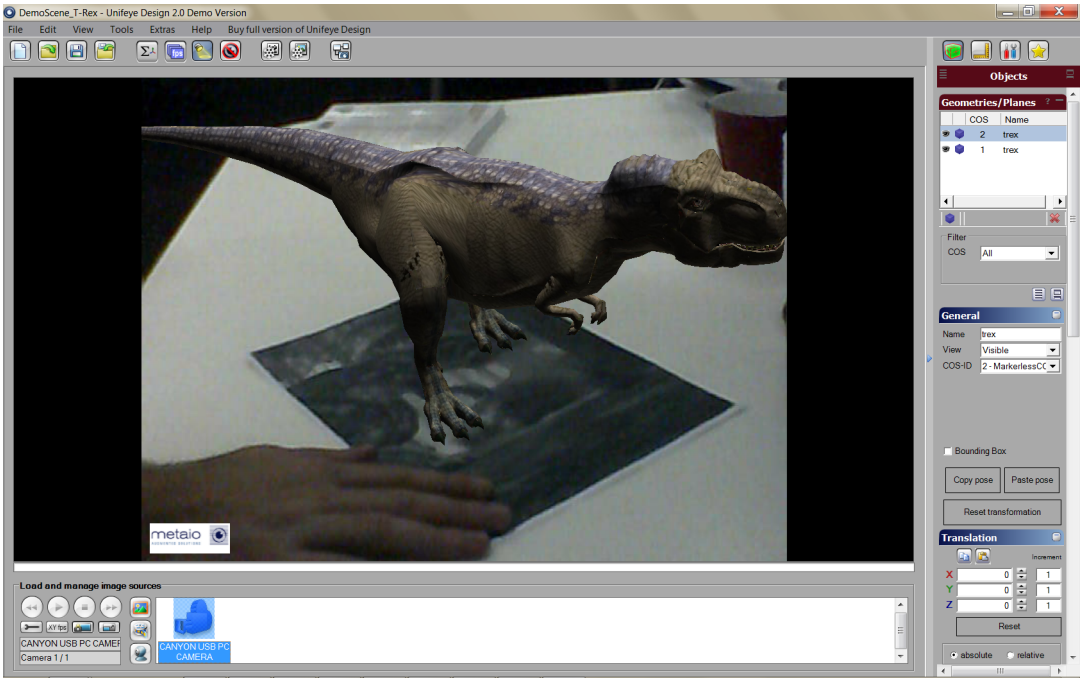


Figure 2.3: A Unifeye design example.

[1] This is only true if a small number of markers are used; comparisons for larger amounts of markers are shown in table A.1 in appendix A.

Unifeye Design is the first professional augmented reality design solution, allowing the user to superimpose high-quality 3D animations onto real-world 2D images using a simple user interface [23]. Unifeye uses dense image tracking superior to what is offered by the mentioned toolkits in terms of stability and occlusion. While Tiles, Unifeye, BuildAR and most other existing author tools attempt to simplify the process of integrating media objects and the real-world, they limits the user to a single type of application. The only interaction that the user has with the virtual objects happens by moving the marker and thus adjusting the object spatially. MARS [24] and DART [25] focuses on the narrative composition. DART combines ARToolkit and Macromedia Director [26], which gives the designer a simple drag-and-drop interface [25]. Macromedia Director requires scripting to make full use of available features. MARS requires no programming skills, and allows designers to insert media in a predefined environment. The program uses a 3D map of the real environment to be used for the augmented experience. MARS allows the designer to insert media that can be time-specific or triggered by an action [2, 27].

2.2.3 Frameworks

We refer to a framework as a library that simplifies the development of more complex augmented reality or mixed reality applications by combining various development tools with a tracking library (such as above mentioned toolkits).

OSGART (OpenSceneGraph and ARToolkit) [28] is a framework that enhances ARToolkit to work with OpenSceneGraph [29]. OpenSceneGraph is an open-source graphics library based on the scene graph concept [30] expanding OpenGL. It provides an object-oriented approach to 3D graphics as well as other utilities for rapid development of graphics applications. The OSGART platform wraps ARToolkit and existing video libraries, and makes it possible to use other marker-tracking libraries as plug-ins. Multiple programming interfaces are available. The framework supports C++ and Python, and includes built-in basic authoring tools [21]. Like ARToolkit, OSGART is available under the GNU public licence.

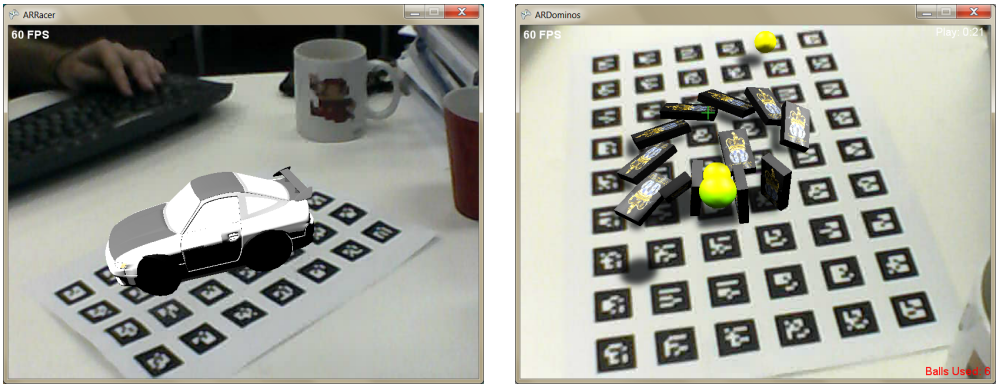
Goblin XNA is similar to OSGART, but it is built on Microsoft's XNA game development platform, for the C# language. The licensing agreement allows for redistribution and use in source and binary forms, with or without modification [31]. The framework supports marker tracking through the

ALVAR tracking system. It also supports physics through the Newton Game Dynamics library. The complete physics implementation is only for virtual-to-virtual interaction. This means that if we can create a 3D scene with enabled physics, it can contain collision detection and dynamic behavior and reaction, taking into account elasticity and friction of surfaces [32]. In an AR application, a simulation of a physics environment can be created and bound to a marker position. If two virtual objects are created, each bounded to its own marker, Goblin XNA can detect collisions between these objects. However, no dynamics will be taken into account during this collision. This is because the physics simulation cannot account for the velocity and torque introduced when we move the marker in the physical world. All dynamic behaviour is therefore relative to the marker, and not the real-world.

2.2.3.1 Goblin XNA Design Example

OSGART and Goblin XNA both use a node-based approach to scene building. The logical flow of program design in the two frameworks are therefore similar. From our tests, it is our opinion that Goblin XNA allows a developer to create intricate AR scenes with more ease than any other solution. Figure 2.4 shows two applications built using Goblin XNA, demonstrating keyboard or mouse input, marker tracking, physics and shadows. The application shown in figure 2.4a was designed for testing purposes, while the application in figure 2.4b is included with the framework as a demonstration. To illustrate the capabilities of currently available design solutions as motivation for our project, we present a brief overview of the design process for the AR driving game from figure 2.4a.

In Goblin XNA, we set up the AR scene using a scene graph. The typical use of a scene graph in computer graphics, would be to place 3D objects, transforms, cameras, lights, and other scene elements into a tree structure in order to simplify relationships between node objects. This application allows a user to control a virtual car with a keyboard in an augmented reality scene. The completed scene graph is shown in figure 2.5. The first step would be to create a ground plane for the virtual car to drive on. The plane is defined relative to a marker position. The marker position is used to place the virtual ground plane relative to a plane in the physical world. The virtual plane object is set to invisible, and placed on top of the marker to represent the marker plane in the virtual physics engine. The car object is build up out of four



(a) (b)

Figure 2.4: Goblin XNA application examples. (a) ARRacer. (b) ARDominoes, a project included with the Goblin XNA download.

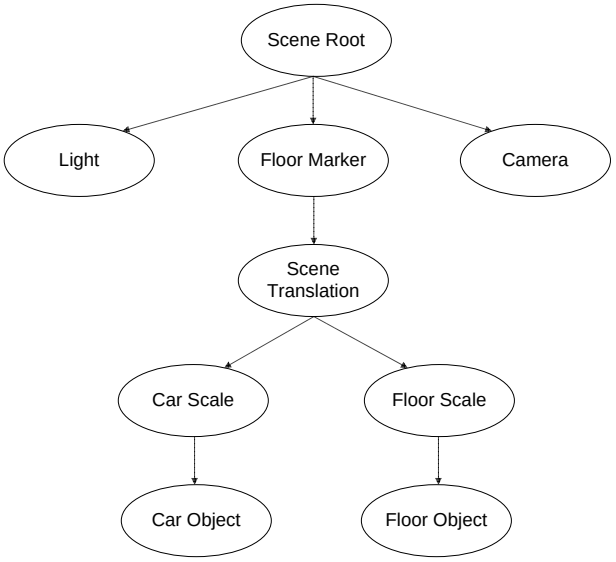


Figure 2.5: The ARRacer scene graph.

wheels objects and a body object, all added to the physics engine. Using the physics engine, the rotation of the front wheels are adjustable, while torque can be applied to the back wheels. The scene graph approach simplifies the design of composite objects with moving parts such as the car, since we are able to edit groups of nodes with a single parent node. The plane and the car are then placed in the scene graph.

Once the scene is set up, the code follows an XNA-based program flow, shown in figure 2.6. To transform the scene according to the marker position,

we associate the tracker to the scene object and the marker position is automatically found at the beginning of every game loop cycle. All of the marker tracker's children nodes are then updated accordingly. This allows us to focus on the virtual content when designing the application [33].

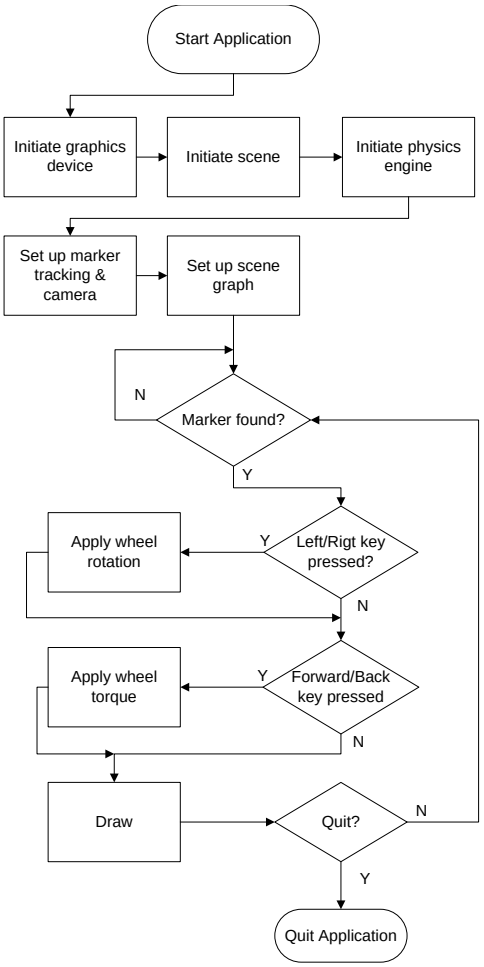


Figure 2.6: The ARRacer program flow.

2.2.4 Conclusion

The design example has been included to give some indication of how the design of virtual worlds have been simplified with Goblin XNA (and similarly OSGART), but in this example the only interaction that occurs with the real-world is the placement of the scene according to a specified position. Goblin XNA allows for the introduction of other elements placed on other markers

into the world. As the physics engine has no knowledge of the velocity and torque of the markers relative to each other, the only interaction that can take place between objects on different markers, is collision.

This has lead to the idea of creating an environment where virtual representations of real-world objects can be created and merged with the virtual world. From this mixed environment the designer should have the freedom to choose which real objects and which virtual objects should be displayed. This means the applications will not necessarily be categorised as augmented reality, but rather be classified under the broader term of mixed reality. As a means to increase immersion and broaden possibilities of the framework, 3D display technology is incorporated. The only option for 3D display when the project was started, was using DirectX and NVidia Geforce 3D technology. The next section provides the needed background on DirectX.

2.3 DirectX

2.3.1 Introduction

This project uses Direct3D, a component of the DirectX API for graphics device interfacing. While it is unnecessary and out of the scope of this project to go into specifics of exactly how Direct3D interfaces with the system, basic knowledge is needed to understand how Direct3D is used in this project.

DirectX was originally developed by Microsoft in an attempt to make the Windows platform more attractive for game development. The first DirectX API was released in 1995 for Windows 95. It has been in constant development and is currently in its 11th iteration. DirectX 10 and DirectX 11 are not supported on Microsoft's most popular (yet older) platform at the time of writing, Windows XP. For this reason the majority of DirectX applications are still developed with DirectX 9 support. DirectX consists of a set of API components. Each API component controls a set of low-level functions that access the specific hardware [30]. The components of the API are:

DirectDraw. This API is used for drawing in 2D with direct access to hardware accelerated capabilities of a computers video adapter.

Direct3D. The Microsoft Direct3D API interfaces with the computers graphics device, and is used to draw 3D scenes.

DirectInput. Input from various peripherals, such as mouse or keyboard are handled by the DirectInput API.

DirectSound. This API interfaces with sound devices, allowing for sound wave capture and playback. DirectSound is capable of low-latency mixing, and hardware acceleration where available.

DirectMusic. DirectMusic works with message-based musical data (MIDI-format) that is converted into digital audio by the sound card or software synthesizer.

DirectPlay. The Microsoft DirectPlay API simplifies access to communication services over a network. It provides applications with a means to communicate independent of the underlying protocol or online service.

DirectShow. The DirectShow API has been moved out of the DirectX package and included in the Microsoft Platform SDK. However, it is included in the discussion because it is used for interfacing. The Microsoft DirectShow API plays multimedia files located locally or on Internet servers, and captures multimedia streams from devices, such as video cameras.

In this project, Direct3D is used extensively, while the framework offers DirectShow USB camera support. Direct3D is used to interface with graphics hardware. Direct3D functionality is manipulated to create mixed-reality scenes.

Direct3D is a low-level API that forms the interface to the graphics device. Direct3D uses Component Object Model (COM) technology to form this interface, allowing it to be language independent and backwards compatible. Direct3D has a defined set of interfaces and functions usable by the programmer. Different graphic cards have different features, limitations and different ways of implementation.

The Hardware Abstraction Layer (HAL) is a set of device-specific code that instructs the device to perform a specific function (in this case, a Direct3D function). The HAL is provided by the device manufacturer that forms the intermediate layer between Direct3D and the hardware. Calling a function not supported by the device does not get passed on to the HAL. This results in failure unless the functionality is emulated in software. HAL allows the user to create applications using DirectX, while the device-specific implementations are done automatically.

The remainder of this section discusses the process of creating a 3D scene

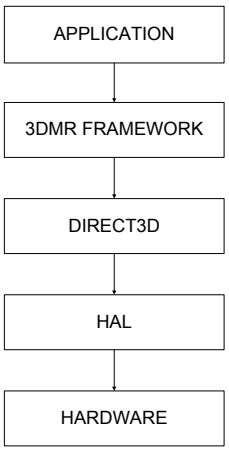


Figure 2.7: The relationship between the application and the graphics hardware.

and presenting it to a screen as a 2D image. Surface data structure and how Direct3D places it in memory are also discussed.

2.3.2 The Virtual Camera

The camera specifies the part of the virtual scene that will be displayed on screen. Figure 2.8 shows the camera model. The orientation and position of the camera can be specified, as well as the amount of space in the scene that will be visualised. The gray volume indicates the viewing volume, or frustum, and is bounded by the near and far clipping planes, and the vertical and horizontal viewing angles. All vertices not inside the viewing frustum are discarded from further processing by a process known as clipping [34]. A vertex is then projected onto the projection plane, representing the screen. The normalised projection window is defined at $Z = 1$ as the rectangular plane lying between $(-1, -1, 1)$ and $(1, 1, 1)$. It should also be noted that DirectX uses a left handed coordinate system, as shown in figure 2.8 with the positive Z axis pointing into the screen.

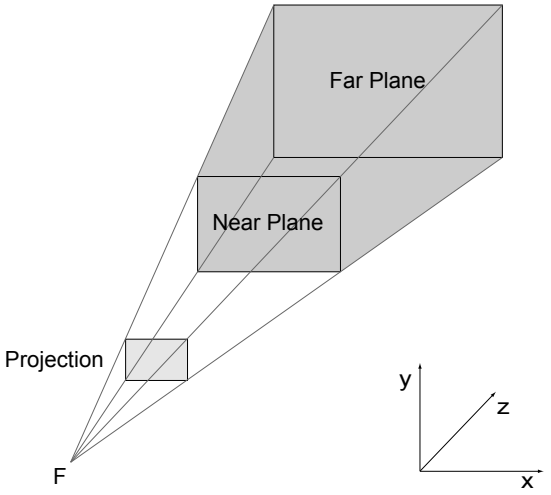


Figure 2.8: DirectX virtual camera model.

2.3.3 Rendering Pipeline

After the camera and 3D virtual scene have been set up, the application can move into the main program loop and start the process of displaying the scene on screen. This process is known as the rendering pipeline. The rendering pipeline is an optimised series of operations, and changing this could impact application performance significantly [35]. We now discuss the steps in the rendering pipeline sequentially.

Local Space. Local space is the coordinate system in which a model's triangle list is defined. If a designer were to create a 3D model in a 3D modeling program, she would typically start modeling at the origin. When the model is loaded into the DirectX application, the same coordinate system would be the local space.

World Space. Once all 3D objects are placed, each in their own local space, they can be placed in the 3D scene, or world space. An object's coordinate system is transformed from its local space to the world space via the world transform, consisting of rotation, translation and scaling matrices.

View Space. While the camera can be placed anywhere in the virtual scene (world space), we define the camera coordinate as (0,0,0) looking towards the positive Z direction in camera space, or view space. The camera is shifted since projecting the 3D scene to the 2D projection plane is less efficient if the camera is at an arbitrary position. This transformation is called the view

space transformation, and the geometry is said to reside in view space after this transformation [35].

Culling. When creating a 3D object in local space, the normal vectors are specified for each of the triangle faces. In the culling step, the faces that do not face forward in the view space, are discarded from further processing, as they will not be 'seen' by the camera.

Lighting. Light sources are defined in world space. Placing a directional light is similar to placing a camera, with a specified position and direction. The lights are then transformed to view space, where they are applied to the scene.

Clipping. At this stage only vertices in the frustum shown in figure 2.8 are needed, so all vertices not in this area can be discarded. This process is called clipping. Clipping can be applied to whole objects, but also to parts of objects not in the viewing frustum.

Projection. The viewing frustum in view space is projected to the projection plane representing the screen with a perspective projection, as discussed in section 2.3.2. Direct3D creates a default perspective projection, based on the screen size and standard near and far clipping planes. This projection is presented to the screen.

2.3.4 Swap Chain

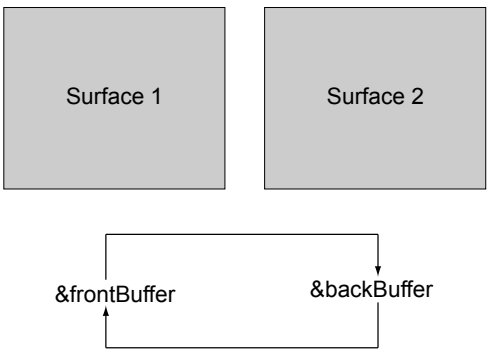


Figure 2.9: Presenting the front buffer causes addresses to swap instantly.

The graphics device contains a surface of pixels holding the information currently shown on the screen. When rendering a scene or surface, the surface

is updated with the information that is to be displayed on the monitor. The updated screen is redrawn from top to bottom. If the monitor was refreshing while a model is being rendered, the display could be cut in two with the top containing the old image, and the bottom the updated image. This effect is called tearing [36]. Direct3D implements a technique called page flipping in order to avoid this.

Figure 2.9 shows a basic swap chain with two surfaces. In this case, surface 1 is currently displayed on the monitor. If a model is rendered, surface 2 is updated with the new information. When all rendering is done, Direct3D will display surface 2, discarding the information on surface 1. This is done by the front and back buffer pointers, with the front buffer being displayed and the back buffer used for drawing, the pointers are simply swapped when surface 2 is ready to be displayed. Surface 2 then becomes the front buffer and surface 1 the back buffer. The process of promotion to the front buffer is called presenting [37].

2.3.5 Depth Buffering

The depth buffer is a surface that contains depth information about pixels. Each pixel in the depth buffer corresponds to a pixel in the render target.

To determine which pixels of an object are in front of another, DirectX uses a technique called depth buffering or z-buffering [34]. Depth buffering computes a depth value for each pixel and compares the depths of pixels competing to be written to a particular pixel location. The pixel with the depth value closest to the camera gets written since it obscures the competing pixels [34].

2.3.6 Surfaces

A Direct3D surface is a linear array of pixels primarily used for storing two-dimensional image data. Although the surface is stored as a linear array, it has properties that make it easy to visualise as a matrix. The height and width of the surface is measured in pixels, and the pitch is measured in bytes.

While surfaces have a wide array of uses as data structures, for our design purposes we distinguish between three types of surfaces: off-screen plain surfaces, off-screen render targets and standard render targets. Off-screen plain

surfaces are used to store data for use by the GPU. The standard render target, usually the back buffer, is where the rendered data to be displayed on the screen is placed. Off-screen render targets are used to render vertices that will not immediately be displayed, for example to render objects with a different projection than the main target. The location of the surface in memory is chosen according to use. We differentiate between three locations:

Video Memory. The default memory pool is allocated in the video memory, which is RAM located on the graphics device local to the GPU. It typically has high throughput to the GPU. Access to and from the CPU occurs over the graphics device bus, and is inherently slow. Typically static meshes, and other data that does not change during runtime, are placed in the video memory [37].

AGP Memory. Dynamic buffers are placed in what is generally referred to as AGP memory (even when using PCI and PCI-E graphics devices), located in CPU-local RAM where its memory can be updated quickly from the CPU. Dynamic buffers are not processed by the GPU as quickly as static buffers because the data must be transferred to video memory before rendering, but the benefit of dynamic buffers is that they can be updated reasonably fast with CPU writes. If the content in a buffer needs to be updated frequently, it should be placed in AGP memory. Examples of elements that should be placed in dynamic buffers are particle systems and skinned, animated meshes [36]. As with video memory, reading AGP memory from an application is slow enough to significantly impact application performance [37].

System Memory. This pool is located in CPU local RAM, designated for use with the graphics device. The memory is cached, and reasonably fast to read from and write to with the CPU and from an application. However, GPU access to system memory is typically slow. If render data needs to be read from application, it would be best placed in the system memory pool, as reading and writing to video and AGP memory from an application is slower [35]. It can also be used as temporary resource storage. For example, a large file could be loaded into system memory. We could then copy small segments of the data into the video memory pool [34, 36].

2.3.7 Accessing Surfaces

There are several ways to access the data in Direct3D surfaces [37]. Two methods are used in this project. The first method, refers to the *LockRect()* function. It allows us to attain a pointer to surface memory and specify a rectangle of pixels in the surface to be edited. After editing, the memory has to be 'unlocked' before the surface can be used by the application. This method gives the advantages of being able to create a pointer to a specific byte and it can be used on surfaces allocated anywhere in memory, but this method is relatively slow. Therefore, it is not ideal for use in the application's main loop. The alternative method is called blitting (from block image transfer) or stretching, and refers to transferring a whole rectangle of pixels between surfaces. The Direct3D *StrechRect()* function, primarily used for resizing surfaces, can be used to blit between surfaces. This method is more suitable for regular calls from the main loop, but works with blocks of data, and has limitations as to the destination and origin surface types that are supported. The function's limitations that need to be considered when using it between surfaces [38] are:

- the source and destination surfaces must be created in the default memory pool.
- blitting is not supported between source and destination rectangles on the same surface.
- blitting is not supported if the destination surface is an off-screen plain surface but the source is not.

2.3.8 Vertex Processing

In its most basic form, a vertex is a point where two edges of a polygon meet. Direct3D allows us to construct a vertex data type to include additional non-special properties. Although Direct3D can process vertices using either software or hardware, in this project a device capable of hardware processing is used, and also a requirement for use of the framework. In Direct3D, a scene or 3D model is created by a triangle mesh approximation. These triangles, or polygons, consist of three edges. These meshes can be used to build virtual worlds (or the virtual part of an augmented reality world).

2.4 Single-View Geometry

2.4.1 Introduction

The previous section presented a basic overview of DirectX, used in the project for the design of the virtual part of an AR application. For the processes involving the interpretation of the physical world, computer vision is used. Computer vision accounts for a large part of this project, both in the design and the implementation of algorithms. This section presents the camera modeling and view geometry associated with the model, to be used throughout the project.

2.4.2 The Pinhole Model

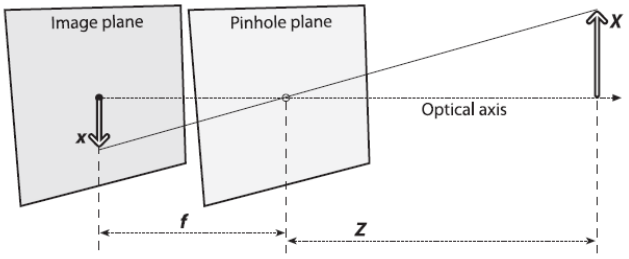


Figure 2.10: The pinhole camera model. Figure from [2].

A useful simple way to model a camera is by using the pinhole camera model [2]. The basic premise for this model is that a single ray of light from a single point is allowed to pass through an aperture: the pinhole. This light ray is then projected onto an image plane and forms the image. Consequently, the image produced is always in focus. The size relationship of the distant object and the image is easily obtained from a single parameter, the focal length f. In this idealised instance the focal length is exactly the length from the image plane to the pinhole plane, as shown in figure 2.10. Here f is the focal length of the camera, Z is the distance from the object to the pinhole, X is the height of the object and x is the height of the object on the image. From figure 2.10:

$$x = -f\frac{X}{Z}$$

In figure 2.11 the image plane is placed in front of the pinhole plane. This is the equivalent to the model shown in figure 2.10, with image plane no longer upside down. In this model, the pinhole is interpreted as the centre of projection. An image is generated by intersecting light rays between a point Q and the center of projection with the image plane. In this ideal model, the optical axis is perpendicular to the image plane, and intersects the image plane at the center point. The intersection of the image plane and the optical axis is referred to as the principal point.

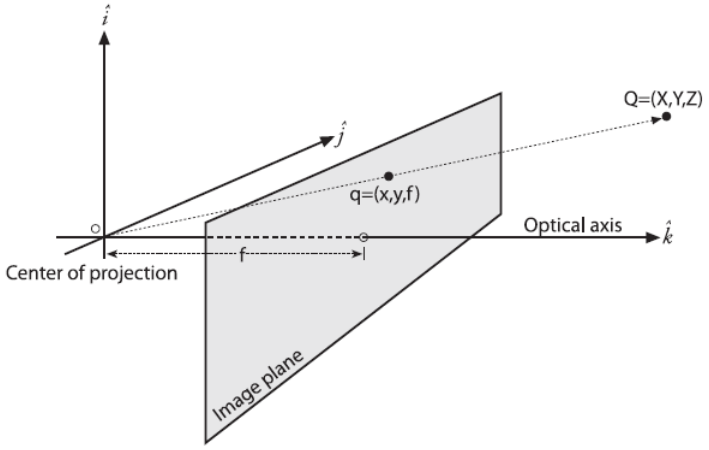


Figure 2.11: The image plane becomes a projection plane between the pinhole and the object. Figure from [2]

In reality a certain amount of light is needed to form an image on the image plane, which is why cameras usually use larger openings instead of pinholes, and lenses to compensate for this. These lenses introduce distortions that are not taken into account in this model. Undistortion is discussed in section 2.5.6. The actual camera is not itself ideal, and we cannot assume that the principal point of the camera lies on the optical axis. Two new parameters are introduced to account for this, namely c_x and c_y. These components are combined with the model in figure 2.11. The result of this is that point Q, with coordinates (X, Y, Z), is projected onto point q, with coordinates (x, y), by the following equations:

$$x = f_x(X/Z) + c_x$$
$$y = f_y(Y/Z) + c_y$$

The reason for introducing two focal lengths f_x and f_y is due to the fact that the individual pixels on low cost imagers are rectangular and not square. The focal length f_x is actually the product of the physical focal length F and the individual pixel size s_x, i.e. $f_x = s_x F$. We use f_x as we cannot compute s_x or F from the calibration process [2]. The same holds for f_y, with $f_y = s_y F$.

2.4.3 Basic View Geometry

The mapping of the point Q in the physical world with coordinates (X, Y, Z) to the point q on the image plane with coordinates (x, y) is known as the projective transform. When working with these transforms it is convenient to use homogeneous coordinates.

The homogeneous coordinates associated with a point in n-dimensional projection space, is a $(n+1)$-dimensional vector. A point q located at (x, y) would then be expressed as $\tilde{\mathbf{q}} = [x'\ y'\ w]^T$. A restriction is added that any two points with proportional homogeneous coordinates can be considered equivalent. Thus we can find the actual pixel coordinates by dividing by w.

We now arrange the camera parameters introduced in section 2.4.2 into a 3×3 matrix, $\mathbf{M}$, which is called the camera intrinsic matrix [39]. Points from the physical world can be projected onto the image plane (or imager) with:

$$\tilde{\mathbf{q}} = \mathbf{M}\mathbf{Q} \tag{2.4.1}$$

where

$$\tilde{\mathbf{q}} = \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

$$\mathbf{M} = \begin{bmatrix} f_x & 0 & -c_x \\ 0 & f_y & -c_y \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{Q} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

From this we find:

$$\tilde{\mathbf{q}} = \begin{bmatrix} f_x X + c_x Z \\ f_y Y + c_y Z \\ Z \end{bmatrix}$$

2.4.4 Rotation, Translation and Scaling Matrices

Any object in space can be described by its position and orientation, i.e. the object pose. A translation and rotation is used to describe $\mathbf{Q}$ in the camera coordinate system as $\mathbf{q}$. This section discusses how translations and rotations are described.

The simplest way to describe a three-dimensional rotation matrix, is as the product of the rotation matrices around each axis, i.e. $\mathbf{R_x}(\theta)$, $\mathbf{R_y}(\phi)$, and $\mathbf{R_z}(\psi)$ [36].

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

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

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

Therefore, $\mathbf{R} = \mathbf{R_x}(\psi)\mathbf{R_y}(\phi)\mathbf{R_z}(\theta)$, and since we can rotate back and forth, $\mathbf{R}^{-1} = \mathbf{R}^T$. Note that rotating the coordinate system of a point by angle θ is equivalent to rotating the point by $-\theta$ [2].

The translation vector from a point $\mathbf{Q_1}$ to a point $\mathbf{Q_2}$ would simply be the offset. With $t_x = P_x - Q_x$, $t_y = P_y - Q_y$ and $t_z = P_z - Q_z$ translation is described by:

$$\mathbf{T} = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

In homogeneous space the rotation matrix is expanded to 4×4 with $w = 1$, and translation is described by the matrix:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In Direct3D, the transposed homogeneous rotation and translation matrices rotate and translate a vertex, since the coordinates are expressed as a row matrix.

2.4.5 Homography

Given a point in 3D space we can calculate its position on the imager as pixel coordinates using perspective projection. The perspective transformation is a type of homography, usually relating to two perspective projections, but not necessarily to two different centers of projection [2].

Consider a point in physical space $\mathbf{Q}$, with homogeneous coordinates:

$$\tilde{\mathbf{Q}} = \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Projection onto the imager is defined as:

$$\tilde{\mathbf{q}} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The action of homography is then given by:

$$\tilde{\mathbf{q}} = s\mathbf{H}\tilde{\mathbf{Q}} \tag{2.4.2}$$

Where s is a scale factor, and $\mathbf{H}$ is the homography, a three-by-three orthogonal matrix. $\mathbf{H}$ contains two parts, the physical transformation for locating the object plane, and the actual projection of the object plane to the imager.

The physical transformation is described by a rotation and a translation. We define a new matrix to represent the physical transformation:

$$\mathbf{W} = \begin{bmatrix} \mathbf{R} & \mathbf{T} \end{bmatrix}$$

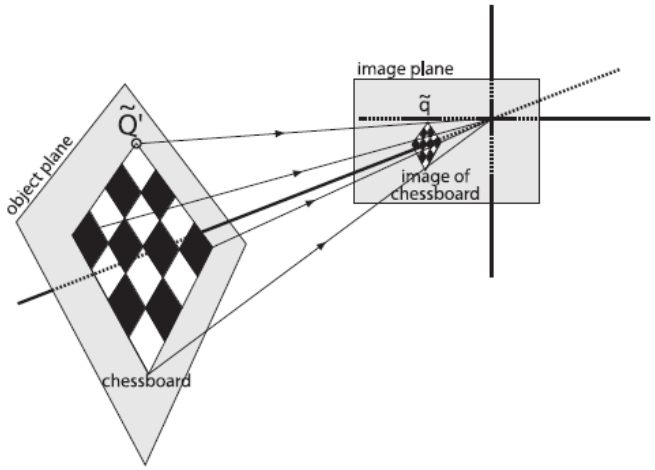


Figure 2.12: The action of homography. Image from [2].

Then $\mathbf{H}$ can be described as :

$$\mathbf{H} = \mathbf{MW} \tag{2.4.3}$$

Where $\mathbf{M}$ is the camera intrinsic matrix from section 2.4.3.

$$\mathbf{M} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Equation 2.4.3 then becomes:

$$\tilde{\mathbf{q}} = s\mathbf{MW}\tilde{\mathbf{Q}}$$

To further simplify the calculations, $\tilde{\mathbf{Q}}'$ is used, which is point $\mathbf{Q}$ on the object plane. The object plane is then defined at Z = 0. With $Z = 0$ only $\mathbf{r_1}$ and $\mathbf{r_2}$ are needed.

$$\begin{bmatrix} r_{11} & r_{22} & r_{13} & T_x \\ r_{21} & r_{22} & r_{23} & T_y \\ r_{31} & r_{32} & r_{33} & T_z \end{bmatrix} \begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{22} & T_x \\ r_{21} & r_{22} & T_y \\ r_{31} & r_{32} & T_z \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

$$= \mathbf{W}' \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

With $\mathbf{H} = \mathbf{MW}'$,

$$\tilde{\mathbf{q}} = \mathbf{H}\tilde{\mathbf{Q}}' \tag{2.4.4}$$

Section 2.4.4 showed that a translation in 3D is described by three offsets and a rotation by three angles, six values. $\mathbf{M}$ also contains four unknown variables.

2.4.6 Distortion

While the pinhole model described in section 2.4.2 is a useful model for geometry in computer vision, in practice it is not feasible to use such a camera in this (or a similar) project. Very little light enters through a pinhole which has an inherently small opening, therefore it would require too much time for light to accumulate on the imager. In order to achieve more light to obtain an image faster, a lens is used. A lens allows for more light to accumulate on the imager, but introduces distortion.

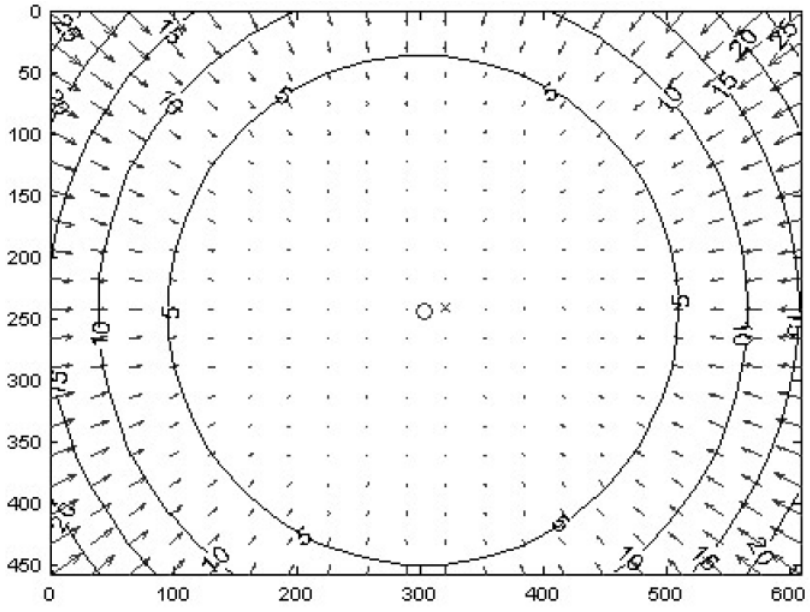


Figure 2.13: Radial component of the distortion model. Image from [2].

Because of the use of non-ideal materials during the manufacturing process, it can be assumed that cameras used with this framework will cause a

significant amount of distortion in its representation of the real-world. The two main types of lens distortion are radial and tangential distortion. Radial distortion is due to the shape of the lens, whereas tangential distortion comes from the assembly process of the actual camera.

Radial distortion is usually formed by a lens that becomes stronger (thicker) further away from the center. Light rays are therefore bent more at the edges of the imager than in the middle. Figure 2.13 shows an example of what happens to images subject to radial distortion. The arrows indicate that points further away from the center are affected more than the center points, where distortion is 0. Since radial distortion is relatively small, it can be characterised by using a Taylor series expansion containing only a few terms. With radial distortion $g(r)$ we have

$$g(r) = k_0 + k_1 r + k_2 r^2 + k_3 r^3 + k_4 r^4 + ...$$

From figure 2.13 we can see $f(0) = 0$, and therefore $k_0 = 0$. As the function should be symmetric in r, the coefficients of uneven powers of r will be zero. As higher order powers of r will quickly converge to zero, the distortion can be categorised using the coefficients of r^2 and r^4 [40]. The radial location of points will be rescaled according to the following equations:

$$x_c = x(1 + k_2 r^2 + k_4 r^4) \tag{2.4.5}$$

$$y_c = y(1 + k_2 r^2 + k_4 r^4) \tag{2.4.6}$$

Tangential distortions are mainly caused by manufacturing defects, as a result of the lens not being exactly parallel to the imaging plane. Tangential distortion is shown in figure 2.14. From [41], we find two additional parameters, p_1 and p_2, used to characterise the tangential distortion. The following equations account for tangential distortions [2].

$$x_c = x + [2p_1 y + p_2(r^2 + 2x^2)] \tag{2.4.7}$$

$$y_c = y + [p_1(r^2 + 2y^2) + 2p_2 x] \tag{2.4.8}$$

2.4.7 Single Camera Calibration

OpenCV uses Zhang's method [42] to find the four camera intrinsic parameters and six external translation and rotation parameters, and a method based on Brown's [43] to solve the distortion parameters. A detailed derivation of these

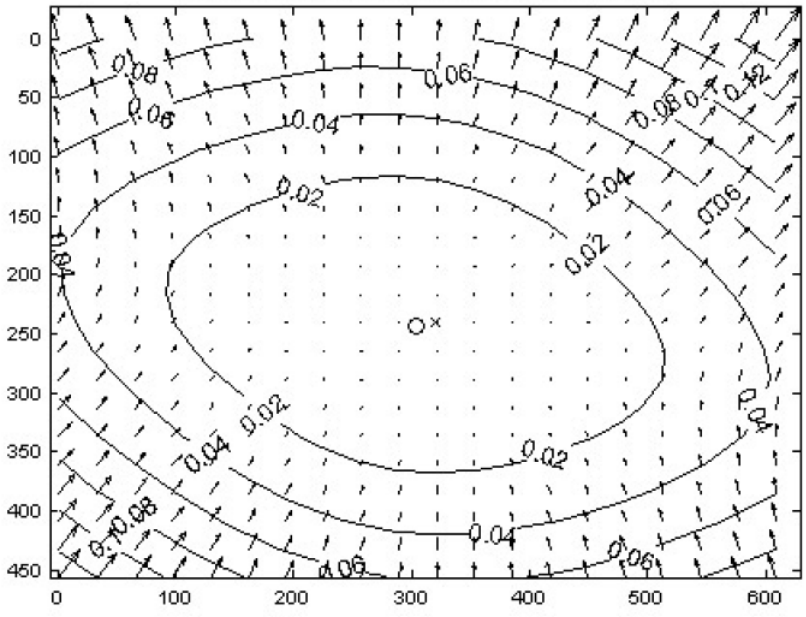


Figure 2.14: Tangential component of the distortion model. Image from [2].

algorithms are beyond the scope of this project, but we present the resulting equations as they provided insight into the calibration process. Equation 2.4.3 is described using column vectors:

$$\mathbf{H} = \begin{bmatrix} \mathbf{h_1} & \mathbf{h_2} & \mathbf{h_3} \end{bmatrix} = \mathbf{sM} \begin{bmatrix} \mathbf{r_1} & \mathbf{r_2} & \mathbf{T} \end{bmatrix} \tag{2.4.9}$$

The matrix $\mathbf{B}$ [2] is defined as

$$\mathbf{B} = \mathbf{M^{-T}M^{-1}}$$

The general solution for $\mathbf{B}$ is found in [42] as

$$\mathbf{B} = \mathbf{M^{-T}M^{-1}} = \begin{bmatrix} \frac{1}{f_x^2} & 0 & \frac{-c_x}{f_x^2} \\ 0 & \frac{1}{f_y^2} & \frac{-c_y}{f_y^2} \\ \frac{-c_x}{f_x^2} & \frac{-c_y}{f_y^2} & \frac{c_x^2}{f_x^2} + \frac{c_y^2}{f_y^2} + 1 \end{bmatrix}$$

Using $\mathbf{B}$, the solutions for the parameters are derived in [42] .

$$f_x = \sqrt{\lambda / B_{11}} \tag{2.4.10}$$

$$f_y = \sqrt{\lambda B_{11} / (B_{11}B_{22} - B_{12}^2)} \tag{2.4.11}$$

$$c_x = -B_{13} f^2 / \lambda \tag{2.4.12}$$

$$c_y = (B_{12}B_{13} - B_{11}B23) / (B_{11}B_{22} - B_{12}^2) \tag{2.4.13}$$

with

$$\lambda = B_{33} - (B_{13}^2 + c_y(B_{12}B_{13} - B_{11}B_{23}))/B_{11}$$

The rotation and translation parameters can now be found from equation 2.4.9.

$$\begin{aligned}\mathbf{r_1} &= \lambda \mathbf{M^{-1}} \mathbf{h_1} \\ \mathbf{r_2} &= \lambda \mathbf{M^{-1}} \mathbf{h_2} \\ \mathbf{r_3} &= \mathbf{r_1} \times \mathbf{r_2} \\ \mathbf{T} &= \lambda \mathbf{M^{-1}} \mathbf{h_3}\end{aligned}$$

The position of a point if there was no lens distortion is (x_c, y_c) with x and y the perceived position. The corrected position is found taking the combined effect of radial and tangential distortion shown in section 2.4.6 into account.

$$\begin{bmatrix} x_c \\ y_c \end{bmatrix} = (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 2p_1 xy + p_2(r^2 + 2x_d^2) \\ p_1(r^2 + 2y_d^2) + 2p_2 xy \end{bmatrix} \tag{2.4.14}$$

In this project we use OpenCV's *CvCalibrateCamera2()* to calibrate cameras. A chessboard like the one shown in figure 2.12 is used for calibration. The size and shape of the chessboard is known, meaning the camera intrinsic and extrinsic parameters, as well as the distortion parameters can be solved given enough chessboard points (x, y) from the input frames. $CvCalibrateCamera2()$ is described in detail in [2]. Taking noise into account, an appropriate approximation of $\mathbf{H}$ can be found within 7-8 frames containing the chessboard [2].

2.5 Multiple View Geometry

2.5.1 Introduction

We distinguish between stereoscopic vision and stereoscopic display. Stereoscopic vision (in the field of computer vision) refers to computing real-world

depth information in an application, using stereo cameras as input. Stereoscopic display (also known as 3D display) is the process of displaying virtual content in artificial 3D using a 3D-capable output device.

The previous section discussed the basic camera model and related geometry. In this project we use OpenCV to find real-world information from a stereo camera rig. This section shows an expanded model for stereoscopic cameras. The OpenCV functions used to set up the rig for use are then described.

2.5.2 Visual Depth Perception

We will start by looking by looking at human depth perception, as this is the base for the modeling of the framework's stereo vision system. Core concepts used in the stereoscopic 3D display are also discussed.

When a human views a scene, the interplay between the two 2D images from the two separate eyes allows the brain to reconstruct depth information about the scene. Since a single image only gives two-dimensional information, we need two or more views of a point in order to have a perception of its depth.

The perceptual transformation differences between the images from the two eyes are called stereopsis. The images viewed by the eyes are perceived to be shifted slightly horizontally, and rotated around the vertical axis when comparing the images from one eye to the other. This is due to the horizontal separation between the eyes, known as the interocular distance. For adults the average interocular distance is 63mm [9].

The light from a single point on an object is emitted onto different parts of the two retinas. The retinal disparity is the relative displacement between the projection of the same point onto two focal planes, in this case the retinas. If the disparity between the two projections becomes too great, the brain does not merge the points with depth information, and the object is displayed twice. This double vision is called diplopia [9].

In order to find the depth correctly, the eyes are rotated around the vertical axis so that they face the focal point. This is referred to as vergence. Divergence occurs when the focus point moves away from the eyes and they are rotated outward, as parallel eye positions imply the focal point is at infinity. Convergence occurs when the eyes are rotated towards each other as the focal point moves towards the eyes.

Figure 2.15 is a representation of the average horizontal human vision field

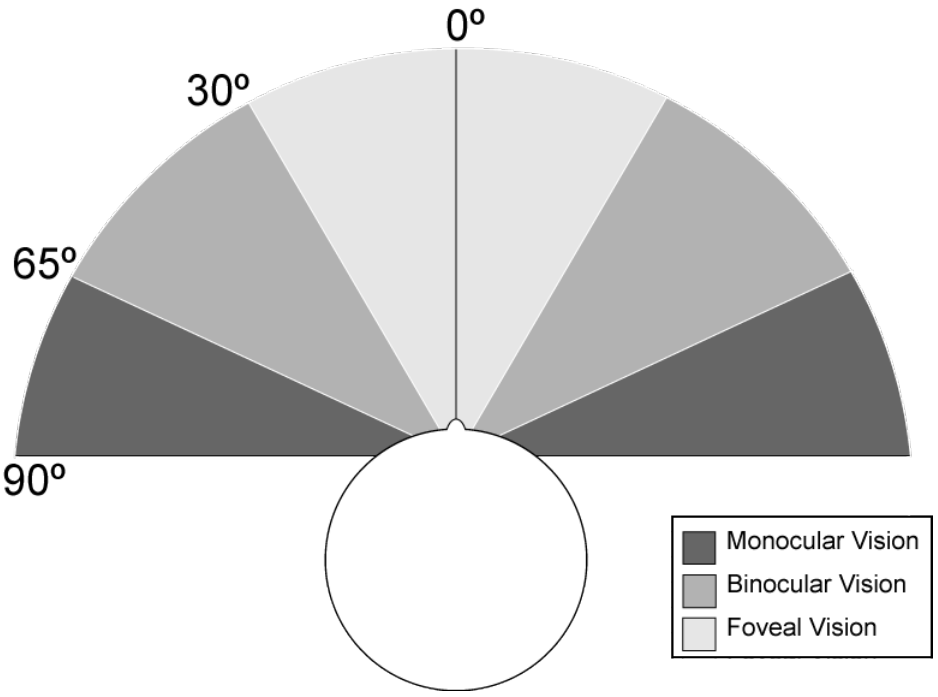


Figure 2.15: An estimate of the horizontal human field of vision, as found in [3].

[3]. With stereo vision and display we are only concerned with the binocular and foveal field. The binocular field of view is the field viewable by both eyes. Both eyes can only focus within the foveal field [9].

2.5.3 Epipolar Geometry

Epipolar geometry is the basic geometry used in the modeling of a stereo imaging system. The geometry combines pinhole models representing cameras with epipoles [35] (refer to figure 2.16). Each camera has a separate center of projection, $\mathbf{O_l}$ and $\mathbf{O_r}$. The real-world point $\mathbf{Q}$ is referred to as $\mathbf{Q_l}$ when its coordinates are given in the left camera's coordinate system, and $\mathbf{Q_r}$ in the right. $\mathbf{Q}$'s projections on the projection planes are given by $\mathbf{q_l}$ and $\mathbf{q_r}$. The epipole is the projected image of the center of projection from the other camera onto the image plane. In figure 2.16 the epipoles are $\mathbf{e_l}$ and $\mathbf{e_r}$. The point $\mathbf{Q}$ and the two epipoles form a plane called the epipolar plane. The lines $\mathbf{q_l e_l}$ and $\mathbf{q_r e_r}$ form epipolar lines. Given a point in one image, its matching view on the other camera imager must lie along the corresponding epipolar line. This is known as the epipolar constraint and reduces the search for matching

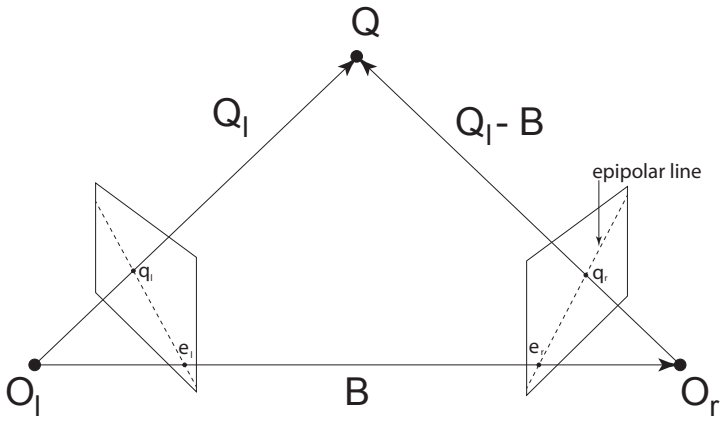


Figure 2.16: Relationship between two imagers [2].

features from the two dimensional image plane to one dimensional along the epipolar lines. This way, many incorrect matches are rejected and computation is greatly reduced [2, 39].

2.5.4 The Essential and Fundamental Matrices

The essential matrix $\mathbf{E}$ describes the relationship between the observed locations $\mathbf{q_r}$ and $\mathbf{q_l}$ of point $\mathbf{Q}$ on two imagers [2].

See figure 2.16. Assume the world coordinate system is aligned with the left camera. Since $\mathbf{Q_l}$ and $\mathbf{B}$ lie on the epipolar plane and the vector $\mathbf{B} \times \mathbf{Q_l}$ is orthogonal to this plane, we can find $\mathbf{Q_l}$:

$$(\mathbf{Q_l} - \mathbf{B})^{\mathbf{T}}(\mathbf{B} \times \mathbf{Q_l}) = 0 \tag{2.5.1}$$

$\mathbf{Q_r}$ is separated $\mathbf{Q_l}$ by translation $\mathbf{B}$ and rotation $\mathbf{R}$. Using $\mathbf{Q_r} = \mathbf{R}(\mathbf{Q_l} - \mathbf{B})$ with equation 2.5.1 yields:

$$(\mathbf{R^T Q_r})^{\mathbf{T}}(\mathbf{B} \times \mathbf{Q_l}) = 0 \tag{2.5.2}$$

The cross-product is rewritten as a matrix multiplication by defining a matrix $\mathbf{S}$.

$$\mathbf{B} \times \mathbf{P_l} = \mathbf{S}\mathbf{P_l}$$

$$\mathbf{S} = \begin{bmatrix} 0 & -B_z & B_y \\ B_z & 0 & -B_x \\ -B_y & B_x & 0 \end{bmatrix}$$

Substitute $\mathbf{S}$ in equation 2.5.2:

$$(\mathbf{Q_r})^{\mathbf{T}}\mathbf{RSQ_l} = 0$$

The essential matrix $\mathbf{E}$ is defined as $\mathbf{RS}$.

$$(\mathbf{Q_r})^{\mathbf{T}}\mathbf{EQ_l} = 0$$

The projective equation states $\mathbf{Q_r} = \mathbf{f_r}\frac{\mathbf{Q_r}}{\mathbf{Z_r}}$, and therefore:

$$\mathbf{q_r^T E q_l} = 0 \tag{2.5.3}$$

The essential matrix is an algebraic representation in epipolar geometry. It contains rotation and translation information relating the two cameras in physical 3D space.

The fundamental matrix $\mathbf{F}$ contains the information in $\mathbf{E}$, and additional information about the particular qualities of the cameras, relating information in pixel coordinates of the two imagers. Suppose that $\mathbf{M_l}$ and $\mathbf{M_R}$ are the camera intrinsic matrices of the left and right cameras. The pixel coordinates $\bar{\mathbf{q}}_\mathbf{l}$ and $\bar{\mathbf{q}}_\mathbf{r}$ are then:

$$\bar{\mathbf{q}}_\mathbf{l} = \mathbf{M_l q_l}$$

$$\bar{\mathbf{q}}_\mathbf{r} = \mathbf{M_r q_r}$$

or

$$\mathbf{q_l} = \mathbf{M_l^{-1}}\bar{\mathbf{q}}_\mathbf{l}$$

$$\mathbf{q_r} = \mathbf{M_r^{-1}}\bar{\mathbf{q}}_\mathbf{r}$$

Using the above equations with equation 2.5.3 gives:

$$(\mathbf{M_r^{-1}}\bar{\mathbf{q}}_\mathbf{r})^{\mathbf{T}}\mathbf{E}(\mathbf{M_l^{-1}}\bar{\mathbf{q}}_\mathbf{l}) = \mathbf{0}$$

The fundamental matrix $\mathbf{F}$ contains the camera intrinsic matrices and the essential matrix.

$$\mathbf{F} = (\mathbf{M_r^{-1}})^{\mathbf{T}}\mathbf{E}(\mathbf{M_l^{-1}})$$

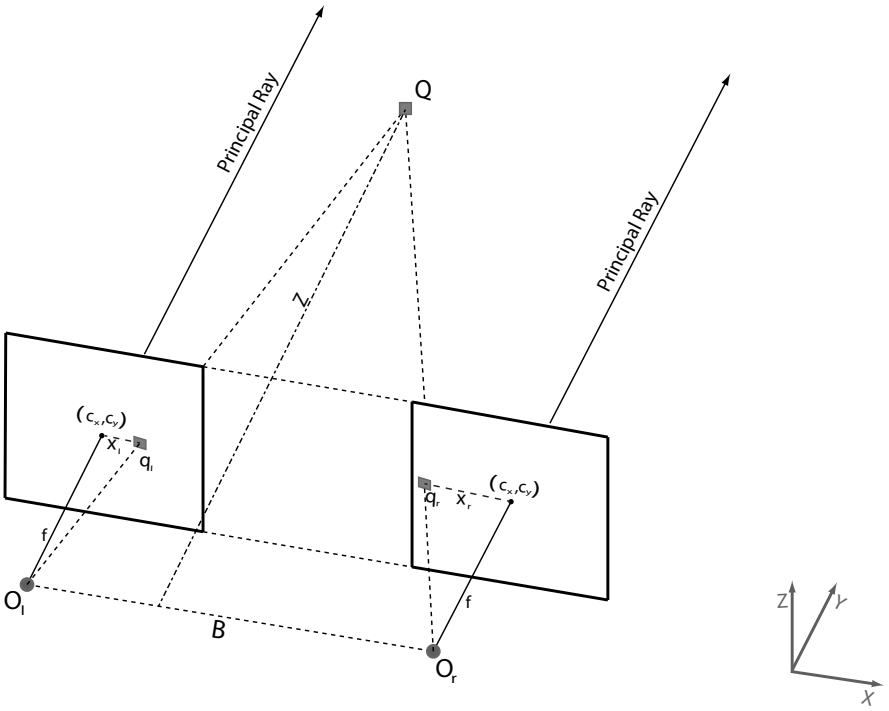


Figure 2.17: Frontal parallel configuration. Figure reproduced from [2].

and

$$\bar{\mathbf{q}}_\mathbf{r}^\mathbf{T}\mathbf{F}\bar{\mathbf{q}}_\mathbf{l} = \mathbf{0}$$

Both **E** and **F** are rank-deficient. They are of rank 2, with seven constraints each [2]. **F** (and **E**) projects a point in one image plane to a line in another. This means that there is a correspondence. However, **F** is not full rank. This means **F** is not a proper correlation; there is no inverse mapping as **F** is not invertible [2, 39].

2.5.5 Disparity

To find the depth of a point $\mathbf{Q} = [X\ Y\ Z]^T$ with two cameras, we need to find the point on the two imagers. Figure 2.17 shows a stereo camera setup with two cameras with equal focal length f, whose image planes are coplanar and optical axes are parallel and a known distance B apart. The cameras are

aligned so that the rows of the two imagers (image planes) are horizontally aligned. This is called a frontal parallel configuration [2]. Assuming that there is no lens distortion, it would be possible to find the depth Z of point $\mathbf{Q}$ from basic geometry, given that the projection points $\mathbf{q_l}$ and $\mathbf{q_r}$ are known. Consider recovering the position of $\mathbf{Q}$ from its projections $\mathbf{q_l}$ and $\mathbf{q_r}$. Suppose the cameras have different coordinate systems with:

$$\mathbf{Q_l} = [X_l \; Y_l \; Z_l]^T$$

$$\mathbf{Q_r} = [X_r \; Y_r \; Z_r]^T$$

The offsets x_l and x_r in figure 2.17 can then be found as:

$$x_r = f\frac{X_r}{Z_r}$$

$$x_l = f\frac{X_l}{Z_l}$$

or

$$X_r = \frac{x_r Z_r}{f} \tag{2.5.4}$$

$$X_l = \frac{x_l Z_l}{f} \tag{2.5.5}$$

Since the cameras are in frontal parallel configuration, they are related by the following transformation:

$$\mathbf{Q_r} = (\mathbf{Q_l} - \mathbf{B}) \tag{2.5.6}$$

where

$$\mathbf{B} = [B \; 0 \; 0]^T$$

With $Z_r = Z_l = Z$, $X_r = X_l - B$ and equations 2.5.4 and 2.5.5 the relationship between Z, x_l an x_r is found.

$$\frac{x_l Z}{f} - B = \frac{x_r Z}{f}$$

$$Z = \frac{Bf}{x_l - x_r}$$

Disparity is defined as the difference in the position between the corresponding points in the two imagers.

$$d = x_l - x_r \tag{2.5.7}$$

Therefore the disparity between views is inversely proportional to the depth Z [2].

$$Z = \frac{fB}{d} \tag{2.5.8}$$

This means that solving depth for a known point is computationally inexpensive, with the camera configuration shown. Unfortunately, the stereo setup described above is very difficult to reproduce practically. It is very difficult to align the cameras in perfect frontal parallel configuration, and the cameras typically have significant lens distortion. To counter this we mathematically find projections and distortion maps to undistort and rectify the real-world stereo rig into frontal parallel configuration [2, 39, 44].

2.5.6 Stereo Calibration

Stereo calibration is the process of calculating the geometrical relationship between two cameras, while stereo rectification is the process of aligning the horizontal rows of the image planes from the two cameras. Although it is possible to rectify the image without calibration, all sense of scale is lost [39] since the physical rig properties like differing camera focal lengths are not taken into account.

Using the process shown in sections 2.4.7 and 2.5.4, it is possible to independently find the camera intrinsics for the two cameras. It is also possible to find the rotation and translation relating objects to a camera plane. This section discusses a way to relate the geometry between the two cameras.

Given a point $\mathbf{Q}$ in space, the positions of $\mathbf{Q}$ on the left and right imagers are

$$\mathbf{q_l} = \mathbf{R_l Q} + \mathbf{T_l} \tag{2.5.9}$$

$$\mathbf{q_r} = \mathbf{R_r Q} + \mathbf{T_r} \tag{2.5.10}$$

Section 2.5.3 showed that

$$\mathbf{Q_l} = \mathbf{R^T}(\mathbf{Q_r} - \mathbf{T}). \tag{2.5.11}$$

Equations 2.5.9, 2.5.10 and 2.5.11 relate $\mathbf{T}$ and $\mathbf{R}$. $\mathbf{T}$ and $\mathbf{R}$ describe the relationship between the right camera plane and the left. These matrices are used to transform the right imager to be co-planar to the left imager. The process used by OpenCV to solve these equations is further described in [2].

With imagers stereo calibrated, they are co-planar; the next step is to align their horizontal rows to simplify the matching process.

2.5.7 Rectification

Bouguet's algorithm [45] implemented in OpenCV attempts to align the rows of the two images and maximise the common viewing area between the them, while minimising the amount of change that reprojection produces in the images. The rotation matrix $\mathbf{R}$ that rotates the left image plane to be coplanar to the right image plane is split into two parts. These matrices are $\mathbf{R_l}$ and $\mathbf{R_r}$ for the left and right cameras. These align the images to be coplanar. The next step is to align the rows of the two imagers with a rotation matrix referred to as $\mathbf{R_{rect}}$. If the left camera's epipole is at infinity and the epipolar line is horizontal, the two images will be in frontal parallel configuration [39]. With (c_x, c_y) as the origin of the left image, the epipole's normalised direction vector lies along the translation vector between the center of projection of the two images.

$$\mathbf{v_1} = \frac{\mathbf{T}}{\| \mathbf{T} \|}$$

Refer to figure 2.17. The principal ray is defined as the ray passing through the camera origin and image origin. In a frontal parallel configuration, the two principal rays are parallel.

We define a direction vector $\mathbf{v_2}$ as orthogonal to $\mathbf{v_1}$ and otherwise unconstrained. To find a value for $\mathbf{v_2}$, it is chosen to be parallel with the image plane by making it orthogonal with the principal ray. This means that $\mathbf{v_2}$ will be the cross product of $\mathbf{v_1}$ and the normalised principal ray.

$$\mathbf{v_2} = \frac{[-T_y T_x 0]^T}{\sqrt{T_x^2 + T_y^2}}$$

A third vector that is orthogonal to $\mathbf{v_1}$ and $\mathbf{v_2}$ is the cross product of the two.

$$\mathbf{v_3} = \mathbf{v_1} \times \mathbf{v_2}$$

From this $\mathbf{R_{rect}}$ would be

$$\mathbf{R_{rect}} = \begin{bmatrix} \mathbf{v_1} \\ \mathbf{v_2} \\ \mathbf{v_3} \end{bmatrix}$$

Row alignment is then achieved with

$$\mathbf{R'_l} = \mathbf{R_{rect}}\mathbf{r_l}$$
$$\mathbf{R'_r} = \mathbf{R_{rect}}\mathbf{r_r}$$

The new camera matrices $\mathbf{M_{rect\_l}}$ and $\mathbf{M_{rect\_r}}$ have to be found. The matrix $\mathbf{P}$ projects a 3D homogeneous point with $w = 1$ to a 2D point in homogeneous coordinates with coordinates given by $x/w, y/w$. The combination of these two matrices for a camera i is given by

$$\mathbf{P_i} = \mathbf{M_{rect\_i}}\mathbf{P'_i} = \begin{bmatrix} f_x & \alpha_i & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

With modern cameras, the pixel skew factor α_i can be assumed to be 0 [2]. P_i projects a three dimensional point to two dimensions. The projection matrix $\tilde{\mathbf{P}}$ maps a two dimensional point to three dimensions [2].

$$\tilde{\mathbf{P}} = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 0 & f \\ 0 & 0 & \frac{-1}{T_x} & 0 \end{bmatrix}$$

A 2D point can be projected to a homogeneous 3D position with:

$$\tilde{\mathbf{P}} \begin{bmatrix} x \\ y \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix}$$

2.6 NVidia 3D Driver Basics

2.6.1 Introduction

This project uses NVidia Geforce 3D Vision for stereoscopic display. It consists of shutter glasses and a screen capable of a vertical refresh rate of up to 120 Hz

used to create the illusion of three-dimensional sight. If the shutter glasses black out the right lens using a single-cell LCD, the image relating to the left eye is displayed. This process is then repeated for the other eye. The screen is synchronised with the shutter glasses, and up to 60 frames per eye are alternated per second. This process is handled by the NVidia stereoscopic driver. For the case of non-professional (non-Quadro) GPU cards, stereoscopic rendering is automatically handled by the stereo driver. The driver supports the display of both existing stereo content, such as 3D movies, and virtual 3D scenes rendered with the graphics device.

2.6.2 Stereoscopic Rendering

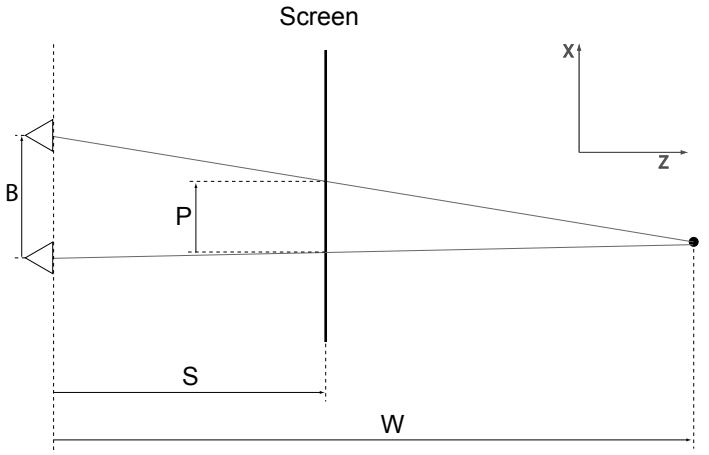


Figure 2.18: Finding parallax from a top-down view.

By simply activating 3D display, the Nvidia stereo driver is capable of rendering a DirectX virtual scene in stereoscopic 3D. However, some design considerations should be taken into account in order to use the stereoscopic rendering efficiently.

The driver makes three changes to the standard pipeline described in section 2.3.3 [46]. The render target's surface is duplicated, creating a stereo surface. The driver creates stereo draw calls instead of the single call used in the standard pipeline. The cameras for the stereo draw calls are created

using stereo separation and modifying the projection matrix. The single virtual camera is replaced with two cameras with a horizontal offset, as shown in figure 2.18. The geometry is then rendered twice per draw cycle, from the left and right view points, resulting in the stereoscopic images.

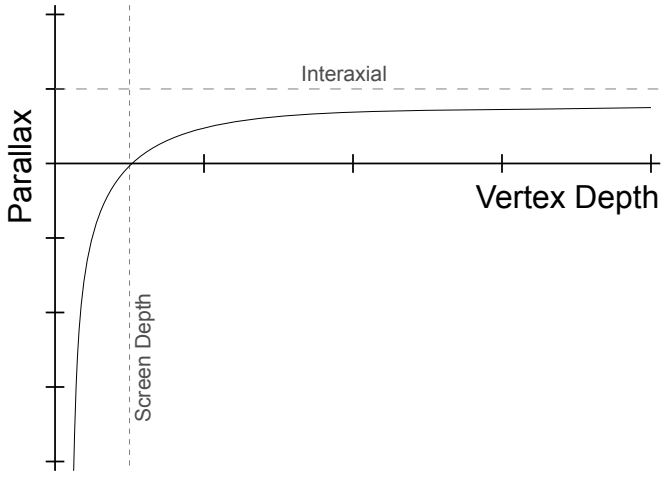


Figure 2.19: The relationship between parallax and vertex depth. Image reproduced from [4].

In this section, the eyes represent the virtual eyes in virtual space, as we are working with artificial binocular vision, this would be what is seen by the user. Refer to figure 2.18. Parallax P is defined as the signed distance on the screen between the projected positions of one vertex, with depth W in the left and right images. P relates to the interaxial distance B, which is the distance between the virtual eyes [4].

$$P = B \times (1 - S/W) \tag{2.6.1}$$

The screen depth S, as a virtual entity, refers to the vergence, which is measured as the depth at which the visual frustum of the two eyes intersect. Figure 2.19 shows that the parallax of a vertex is 0 at screen depth, which means the vertex is rendered on the same screen pixel for both views. Maximum parallax is limited, converging to B. When a vertex moves out of screen closer to the virtual eyes, it quickly diverges to negative infinity. When a vertex appears in front of the physical screen, it creates the effect that the vertex

is floating outside the screen. At $\frac{S}{3}$,

$$P = -2B$$

A parallax this big can cause eye strain or diplopia, especially when a user's eyes have not accustomed to the out-of-screen effect. Looking at what happens if a vertex moves deeper into the scene, a vertex positioned at $10S$ has $P = 0.9B$. Change in parallax happens slowly after this, with $P = 0.99B$ at $100S$, giving a 9% increase in parallax over a distance of $90S$, making small changes in depth difficult to detect [4].

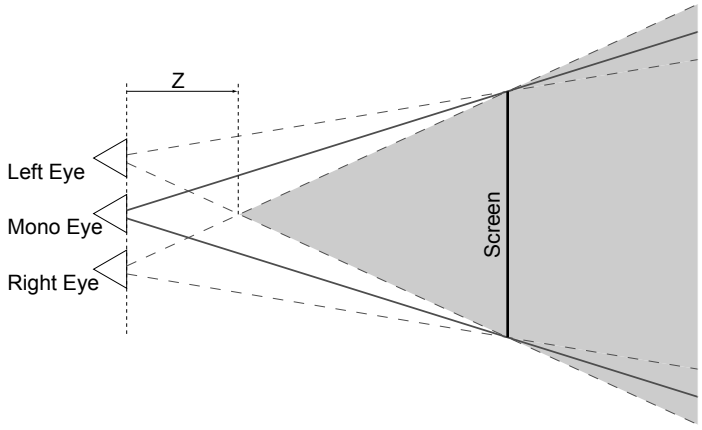


Figure 2.20: The stereoscopic frustum.

The other aspect to take into account is the culling frustum [46]. Using two eyes instead of one changes the viewing frustum, as shown in figure 2.20. When in front of the screen, binocular space is included in the frustum. The areas only covered in the monocular left or right frustums give us no sense of depth and will disturb stereo perception, causing diplopia. Within the screen, the mono frustum excludes a segment of the left eye frustum to the right of the screen, and a segment of the right eye frustum to the left of the screen. These should be included in the visible frustum. The clipping frustum is modified according to these considerations, and the resulting frustum is shown in figure 2.20 [4] with

$$Z = S/(1 + S/B).$$

2.6.3 Stereoscopic Images

The NVidia stereo driver is capable of displaying a pair of stereoscopic images in 3D. This is supported through a stereoscopic player, capable of displaying photo pairs or 3D movies. We will attempt to use this functionality in the project. Unlike the stereoscopic rendering, this process is not automated. The design is discussed in chapter 4.

Chapter 3

System Design

3.1 Introduction

This chapter discusses the design of the mixed reality framework, and how the approach differs to the solutions discussed in Chapter 2. Section 3.2 discusses the initial conceptualisation of the idea, and the reasoning behind the design choices made. The subsequent sections discuss the software design of the framework with a top-down approach. Section 3.3 presents an overview of the framework. Thereafter each of the individual packages that make up the framework are presented.

3.2 Conceptual Design

Section 2.2 gave an overview of currently available desktop AR solutions. Two methods were employed in the expansion of the basic toolkit approach seen in ARToolkit. The first is the simplification of the virtual world design (OSGART) making more intricate worlds possible. The second is the simplification of the authoring of AR applications (BuildAR). In all cases the main program loop could be summed up as:

1. Grab input camera frame.
2. Find markers.
3. Estimate pose.
4. Compute and render virtual world.

5. Output composite image.

Little deviation from this set of processes was seen through the testing of AR development and design solutions. Step 4 differs according to the intricacy of the framework, as OSGART and GoblinXNA allow for the creation of elaborate scenes. Other tools are also available to create realistic and intricate virtual components for use in these mixed reality environments [47, 48, 49]. Step 5 also had some variation on the type of output device. Many of the toolkits allow development for a variety of output devices, usually a monitor and HUD devices. This means that currently available augmented reality tools have been mainly expanded with the goal of creating better virtual components for the mixed world, and to better display the mixed world.

The approach to real-world estimation, as represented in steps 2 and 3, has not been expanded in any of the tested AR solutions. As stated earlier, markerless alternatives exist, but at the time of writing are not viable replacements for marker tracking since they are processing intensive and not as accurate. Despite this, there are computer techniques that could be used to widen possibilities in mixed world design.

These factors have lead to the design goal of this project: to create a framework capable of delivering standard marker-based AR applications while offering developers wider real-world and object estimation techniques. Technology and computer vision are improving fast. Taking this into account, it is important that the process of changing and adding computer vision algorithms should be as easy as possible.

The decision was made to incorporate a modular approach to real-world estimation by implementing interchangeable algorithms that find and interpret data from the real-world, and present it as virtual entities. These new representations can then be used alongside marker tracking to create new mixed reality environments.

While the framework's implementation is broader than, for example, ARToolkit, it should not be more difficult to create a typical AR application.

We divide the framework into three segments according to the abovementioned steps, offering the user various options to use in combination under the groupings input, real-world estimation (computer vision), and virtual world calculation (graphics). This is discussed in the next section.

Based on preliminary considerations, the following design choices were made.

Coding Language. The coding language used for this framework is C++. For abstraction we choose an object-oriented approach. As real time applications are created using the framework, performance and efficiency was the main consideration in making this choice. Considering that a large amount of time-critical calculations would occur, C++ typically outperforms other object orientated languages such as Java, Python and C# [50]. Some tools and documentation needed for the framework are available in C and C++ that do not have an equivalent in the other languages.

Graphics Library. The two libraries most widely used for graphics device interfacing are OpenGL (Open Graphics Library) and Microsoft's Direct3D. The libraries have comparable performance, and both are free to use for academic purposes. The framework uses Geforce 3D Vision technology, which is only officially supported in Direct3D at the time of writing. For this reason we use Direct3D, part of the DirectX package. DirectX 9 is chosen instead of the more powerful DirectX 10 due to lack of available study material on DirectX 10. Some documentation on the stereoscopic driver is available for DirectX 9 while no official documentation could be found for DirectX 10.

Platform. Since Direct3D is used as the graphics library, the framework will not be platform independent, but will only be compatible with the Windows operating system. This includes Windows Vista and Windows 7, 32 bit and 64-bit versions. The rest of the framework is independent of the graphics library due to the modular design approach, meaning that that other graphics libraries such as OpenGL can be implemented and used with the framework if needed. Due to the 3D vision driver, an NVidia Graphics card and a 3D ready display are needed[1].

Computer Vision. The focus of the framework would be to build a high-level development environment for mixed reality. Any previous work done should ideally not be duplicated, while it is impractical to have a large number of dependencies. OpenCV has 500 functions that span many areas of computer vision, including user interfacing, camera calibration and stereo vision. A machine learning sub-library is also included for tasks such as pattern recognition and clustering. The online user group has more than 40 000 active

[1] A full list of compatible devices can be found at [51]

users. Other open-source computer vision libraries include the CImg library [52], VXL [53] and IVT [54]. None of these libraries have documentation or user bases (and user base support groups) comparable to that of OpenCV. The framework should ideally be easy to expand, and OpenCV offers more options than any of the other computer vision libraries. For these reasons, OpenCV is chosen to simplify computer vision in the framework.

For marker tracking, the ALVAR marker tracking system is chosen. ALVAR functions and performs similarly to ARToolkit [31]. ARToolkit is available for use under the GNU General Public License, and due to the ownership of the intellectual property of this thesis project, it would not be ideal to include software with this licence agreement. ARTag is the other widely user tracking system, and performs better than ARToolkit and ALVAR under certain circumstances, as shown in appendix A. At the time of writing ARTag was no longer available from the NRC, where it was developed.

Design Philosophy. For abstraction and encapsulation of complexities, we use an object-oriented approach. In the design, ease of use was important. The functions are also designed so that different applications can follow a similar use pattern. While error checking is done at initiation time, it is kept minimal in the main program loop due to performance considerations. Computer vision algorithms are chosen with processing time per frame in mind, as real-time (or near real-time) execution is essential.

3.3 Framework Overview

3.3.1 High Level Overview

Figure 3.1 shows the framework overview and class grouping. A developer using the framework should be able to create an AR application in a method comparable with the other AR solutions. To adhere to this, the framework has been categorised according to the three basic steps needed to create a MR application:

- Camera input handling.
- Computer vision.
- Virtual world computation and graphics device output.

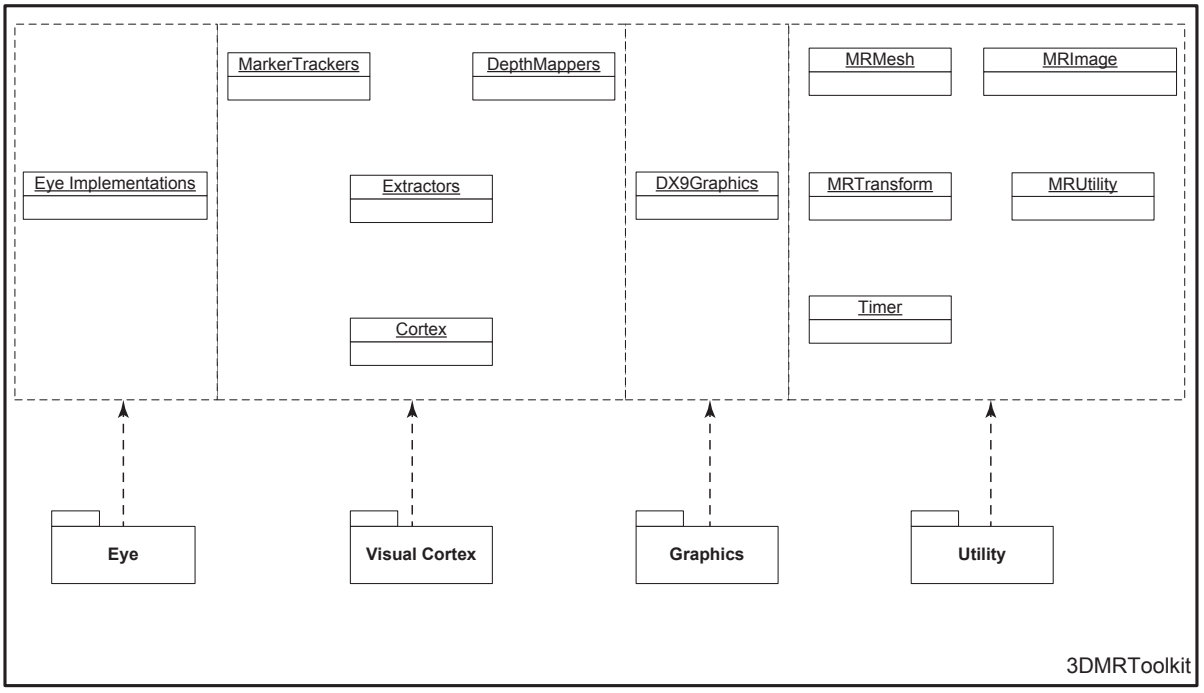


Figure 3.1: Framework overview.

Eye. This grouping contains classes for handling the input image interfacing. USB camera capture is supported through the DirectShow and CL-Eye APIs. Manipulations typically done on a driver level are also included as wrapper functions.

Visual Cortex. Once the developer has captured an image from a USB camera, he will typically want to extract some form of real-world information from the image, usable in a virtual environment. These computer vision algorithms are grouped together in the Visual Cortex.

Graphics. The Graphics object can be thought of as an interface with the graphics device. It is used to set up and draw 3D scenes and display video. All calculations to be handled by the GPU are encapsulated in this class.

Utility. This group does not represent one of the steps needed to create mixed reality application. It contains classes with helper functions, encapsulating commonly used rudimental functionality. This group is designed to simplify the use of the framework where possible.

3.3.2 Expanding the Framework

Marker tracking is not the only means the framework will provide to estimate the position of real-world entities. In the expansion of the Visual Cortex, design goals were set:

- The framework should have an expandable approach to computer vision.
- The developer should be able to add computer vision algorithms for use in the design of a mixed reality application.
- The framework should make it easy to add and use these algorithms as augmentations or replacements of algorithms already incorporated in the framework.

Refer to figure 3.1. The Visual Cortex envelops three real-world interpretation tools that can be used in parallel: a marker tracker, extractors, and a depth mapper. The marker tracker can be used to find the pose of a known marker in a real-world image. The extractors are used to interpret image data. The framework implements different levels of extractors, designed to extract some data from an image and convert it into a virtual three-dimensional entity that can be compared to other virtual entities in a virtual environment. These extractors are further discussed in section 3.5 and chapter 4.

A depth mapper is included to assist the extractors in finding the 3D position of some object in real space. The depth mapper uses stereo cameras to find the depth of a real-world entity. Section 3.5 describes how the Visual Cortex handles depth mapping. The algorithm used in the implemented depth mapper is described in chapter 4.

We stated that a developer using this framework should have the freedom to create applications across the entire mixed reality continuum. The designer can use any combination of Visual Cortex elements to create her program.

The Graphics object is expanded to control the 3D vision driver. This is done so that the framework can be used to experiment with the possibilities that 3D vision technology presents to mixed reality design.

The framework is designed to have interchangeable and editable components to serve as a testbed for new algorithms and design ideas. The design of the components are discussed in the following sections.

3.4 Eye

The abstract IEye class represents an interface. Its concrete implementations are used to handle input from camera (or file). The PlayStation Eye is the preferred camera used throughout this project. It is capable of capturing video with frame rates of 60 Hz at a 640×480 pixel resolution, and 120 Hz at 320×240 pixel resolution [55]. The Eye is capable of outputting uncompressed video at these rates. This sets it apart from other cameras in its price range capable of capturing at the same resolutions. The PlayStation Eye has a fixed-focus adjustable lens that can be set to a 56° field of view, for close-up framing, or a 70° field of view for long shot framing [56]. To stream video from the camera to the PC, the CLEye driver is used. As it is a driver specific to the camera, the framework supports DirectShow, included in the DirectX package, a driver that supports more common PC USB video capturing devices. Furthermore, it is convenient to have still image input from file, for debugging and testing purposes. These are all different implementations with similar functionality. For this reason we create the abstract class Eye, and create concrete subclasses derived from Eye to implement the different driver interfaces, as shown in figure 3.2. Other camera support can be added at a later stage without disrupting the rest of the framework.

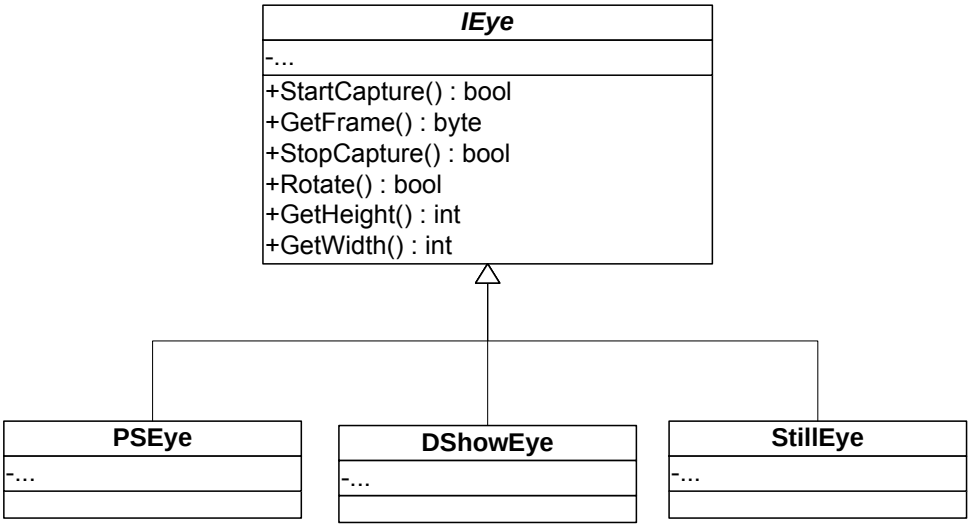


Figure 3.2: The Eye class diagram.

3.5 Visual Cortex

3.5.1 Introduction

The class is used as a container for all image processing algorithms. The image from the Eye class is stored in the framework image format, MRImage, described in section 3.7. Some real-world information would be extracted from a captured frame. The specific information extracted would vary according to the application. Figure 3.3 shows the possible uses for the Visual Cortex. The algorithms are classified according to use.

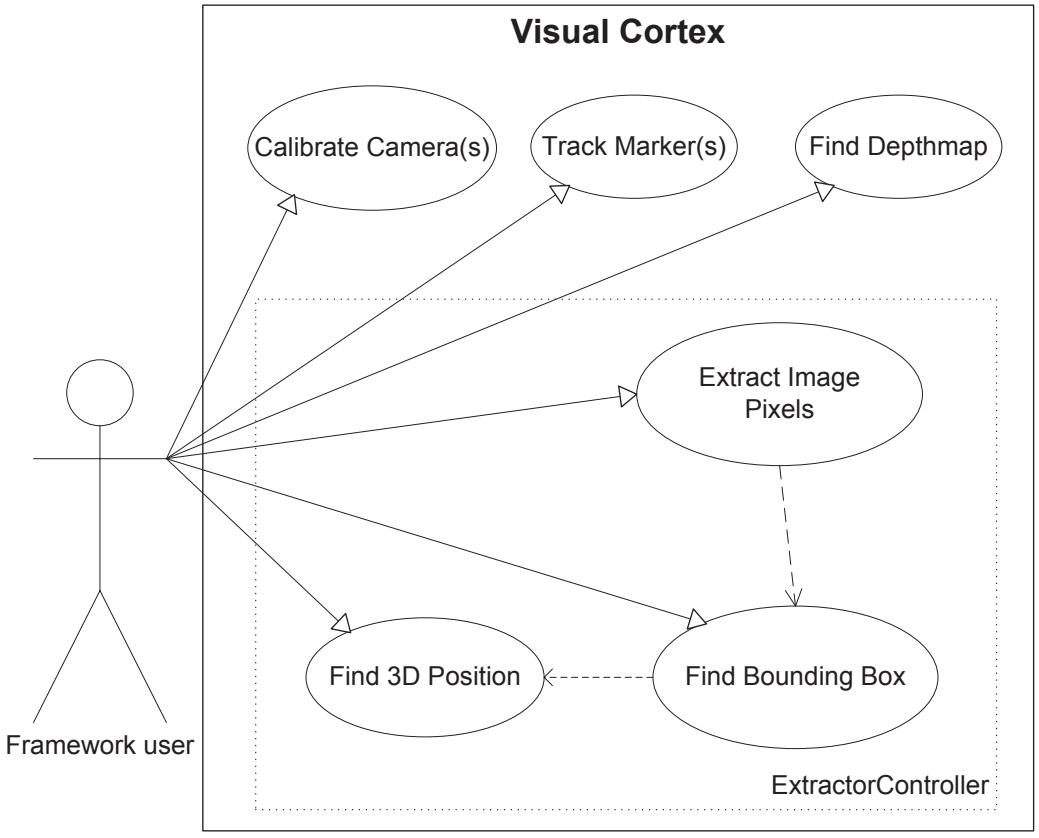


Figure 3.3: Visual Cortex use case diagram.

3.5.2 Marker Tracking

Figure 3.4 shows the Marker Tracker class diagram. The framework contains one implemented marker tracker: A high-level wrapper to the ALVAR tracking

library.

ALVAR has similar functionality and performance to ARToolkit. Single marker detection is supported, as well as array marker detection, using multiple markers for pose detection. The tracker differentiates between 256 indexed markers that can be used for tracking. To use the marker tracker, the developer must initialise it at the start of the application. He can then find markers in an image using a single function call that returns the 3D pose of the marker. The original ALVAR pose is returned for use in a right-handed (OpenGL) coordinate system. The framework functions convert the translation and rotation matrices for a left-handed coordinate system before creating the new DirectX pose matrix.

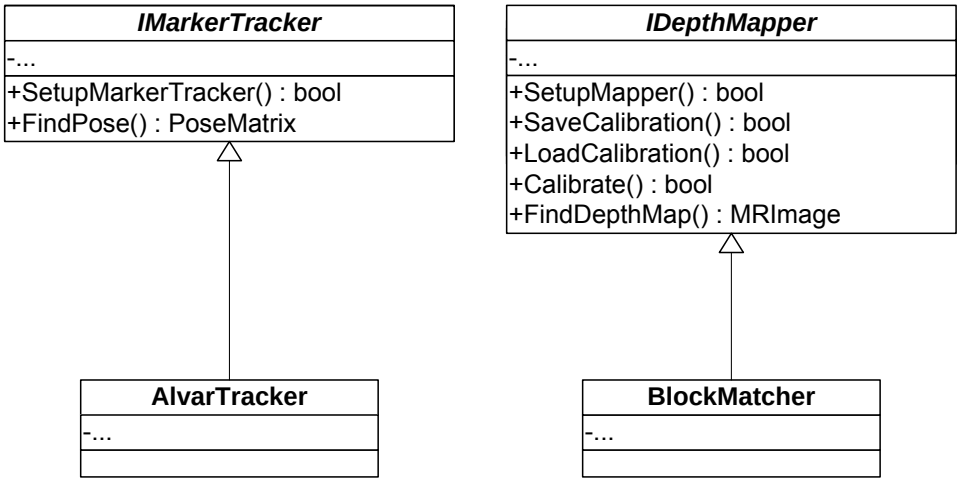


Figure 3.4: The marker tracker and depth mapper class diagrams.

3.5.3 Depth Mapping

In order to find depth in the scene, a stereo camera rig is used. The depth mapper class diagram is shown in figure 3.4. IDepthMapper contains a set of functions for calibrating the rig, and finding the relating depth map as returning it as a single channel image. A block matching depth mapper is implemented in the framework and discussed in chapter 4.

3.5.4 Extractors

A mixed-reality environment contains both real and virtual entities. With the marker tracker a virtual representation of the marker can be created. The extractors are included in the framework so that a developer can create a virtual representation of real-world entities without the use of markers. Taking an image from the input cameras, the developer should be able to use a extractor to extract a desired entity from the image. This extracted entity will only have 2D coordinates relative to the image from which it has been extracted. The developer can then find the 3D pose of the 2D entity from the disparity map in the depth mapper to create a 3D representation of the entity.

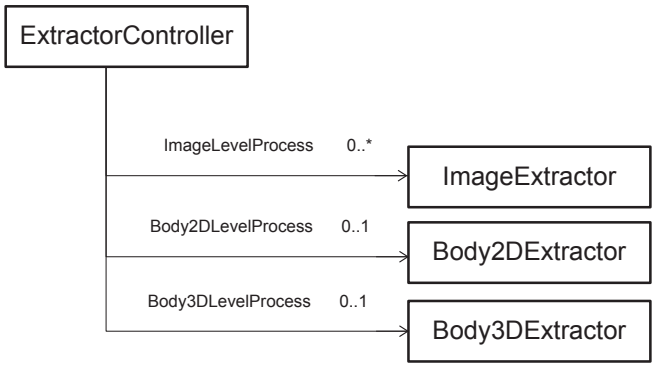


Figure 3.5: The relationship between ExtractorController class and the extractors.

Three levels of extractors are defined, as shown in figure 3.5. The first is the ImageExtractor, where the pixels of a given object are found in an image. If a developer wants to find a red ball, for instance, he can implement an ImageExtractor that extracts all the red pixels from the source image. The next step would be to look for extracted pixels in the shape of a ball. When these pixels are found, the pixels need to be converted into some intelligible entity. The red ball pixels could be replaced by a circle with radius and coordinates. Finding the circle entity from the red pixels would be done by implementing a Body2DExtractor. After using the Body2DExtractor, we would have a virtual 2D representation of a real-world object. To find a 3D representation, the depth of the object has to be found. The Body3DExtractor is used to find the position of the virtual 2D representation in 3D space, and to create a 3D entity

from the circle. For the ball, the Body3DExtractor could create a sphere, for example. The disparity map could be used by the extractor to find the depth of the sphere. This virtual 3D representation can now be placed in the same virtual space as other virtual entities, and can be treated as such.

The ExtractorController abstracts the extractors from the MR application. See figure 3.5. The developer can add extractors to the appropriate level process. The levels from lowest to highest are ImageExtractor, Body2DExtractor, and Body3DExtractor. Once in the main program loop, the controller will automatically process all the extractors. The developer can extend the extractors shown in figure 3.5 according to her needs. As a proof of concept, the framework contains subclasses for each level extractor.

ImageExtractors. Three methods find and return pixels of interest. These methods differentiate between pixels according to the pixel colour. The first extracts foreground pixels from an image by comparing the image pixels to a previously defined background image. The second method finds colour within a specific band in the HSV[2] colour space as defined by the developer. The third is used to supress unwanted shadows that might not be removed by one of the other two methods. These three methods are encapsulated in three extractors, as shown in figure 3.6.

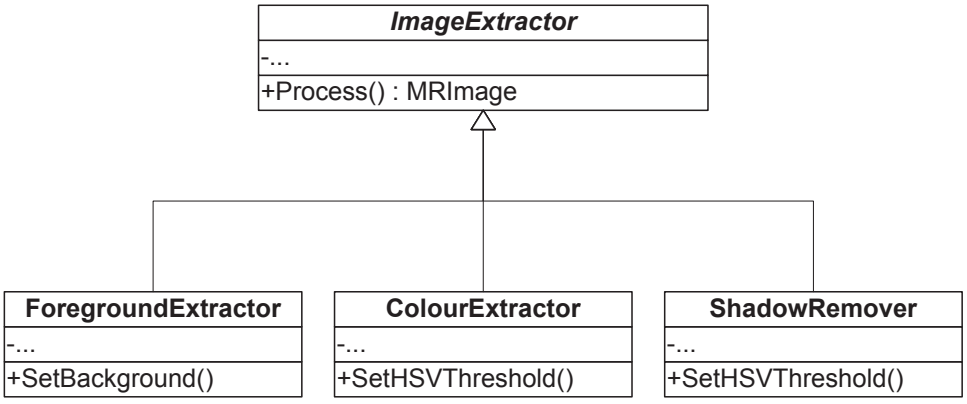


Figure 3.6: The ImageExtractor class diagram.

[2]The HSV colour space is divided into three channels: hue, saturation and value. Hue is the colour, saturation indicates the amount of gray in the colour, and value the brightness. If saturation is 0%, the colour is gray. If value is at 0%, the colour is black.

Body2DExtractors. Body2DExtractors create basic bounding shapes around the extracted pixels passed by the ImageExtractors. Figure 3.7 shows the Body2DExtractor class diagram. Closed contours are created around groups of these pixels. The two Body2DExtractors included in the framework only look for the biggest contour in the image, i.e. the largest object of interest. This means that the framework currently focusses on one object of interest at a time. A bounding box or ellipsoid is then found from the contour, along with a centre point. This bounding box represents the object in 2D screen space for interaction.

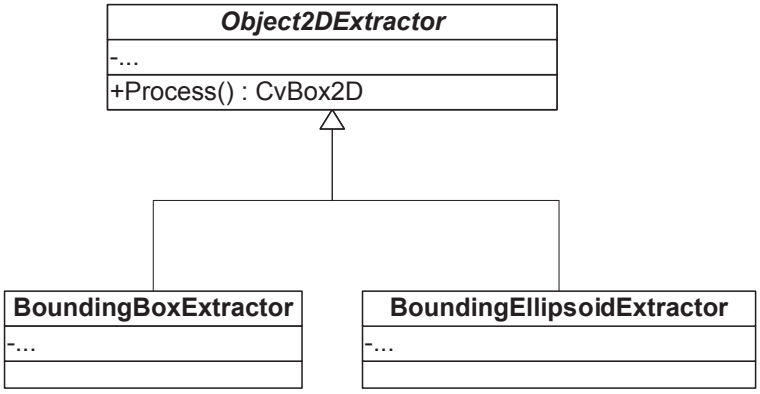


Figure 3.7: The Body2DExtractor class diagram.

Body3DExtractor. This is used when a 3D representation of the 2D virtual entity found by a Body2DExtractor is required. The class diagram is shown in figure 3.8. The bounding box object is transformed into 3D world space, i.e. world space in the DirectX virtual scene. The depth of the bounding ellipsoid is found using stereo vision coordinates, and a bounding sphere representing the real object is returned. The bounding sphere is estimated from the bounding box or ellipsoid. A bounding sphere was chosen because it is the simplest bounding shape for collision detection in 3D space [37].

3.6 Graphics

The Graphics object handles all interaction with the graphics device. The framework currently supports Direct3D from DirectX 9, using the D9Graphics

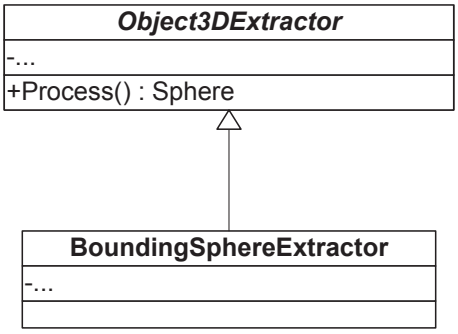


Figure 3.8: The Body3DExtractor class diagram.

class. Figure 3.9 shows possible use-cases for the Graphics object. Since Direct3D is a low level API, the framework includes initialisation functions that set up the scene for typical use. It fulfils the role of a basic game (or simulator) engine with added augmented reality support [57].

A scene can be initialised by a combination of two initialisation functions. The first initialises the graphic device and a 3D scene so that the user can add and render content. This includes the setup of directional and ambient white light, the camera and its projection matrix, as well as automatic depth buffering. The lighting and camera parameters can be changed after setup, so it is convenient to use the initialisation function even if the scene has to be set up differently. A modified version also exists that sets up the back buffer so that windowed mode is supported, along with typical features such as window resizing support.

The second part of initialisation builds on the first one, adding video support for the creation of, for example, typical augmented reality applications. The first option sets up the device for 2D video support, also allowing for video resizing to be handled by the GPU. The Graphics class also includes an initialisation function that allows for full-screen 3D video to be displayed.

Within the main program loop, the framework provides functions to simplify video display, virtual object placement and scene rendering.

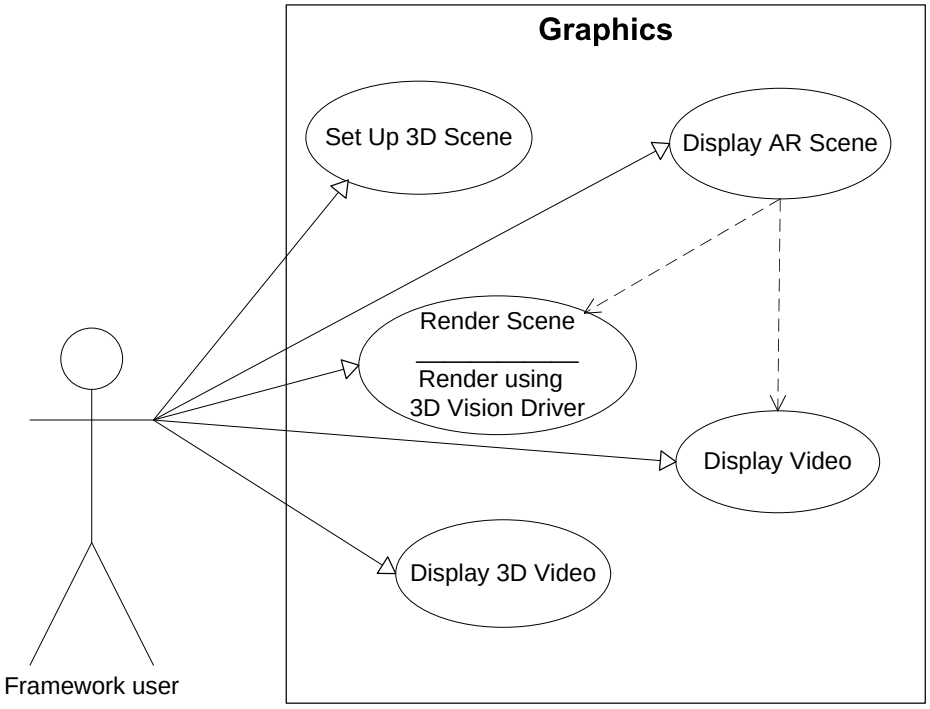


Figure 3.9: Graphics class use case diagram.

3.7 Utilities

The Utility classes are classes with helper functions used throughout the framework. The MRMesh, MRMeshInstance, MRUtility and MRTransform classes were created to handle meshes in the framework environment.

A developer can load a static DirectX UV textured mesh from file into the index buffer with MRMesh. UV Meshes are a standard way of handling textured meshes in 3D applications [36]. It allows for a texture saved in a image file to be mapped to a model. Every vertex is assigned a coordinate relating to a point on the image. In order to avoid confusion with 3D space or screen projection coordinates, the image coordinate system is defined as U in the horizontal direction and V in the vertical direction. The texture is mapped to the mesh according to these UV coordinates.

For memory efficiency, every instance of this mesh object placed within the scene, is created using the MRMeshInstance class, that references a MRMesh object. Figure 3.10 illustrates the use of MRMesh and MRMeshInstance in an application, for the case of a single loaded .x mesh (DirectX format UV mesh

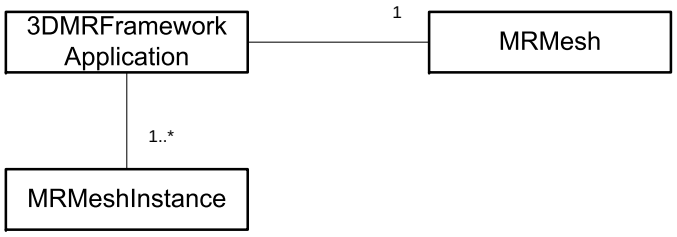


Figure 3.10: Using MRMeshInstance in an application.

file) placed more than once in a scene.

The MRMesh class is used to load in vertex data from a DirectX mesh file into its *Direct3D.Mesh* member. The information contained in a DirectX mesh file differs depending on its source, since some modeling program export functions do not support vertex format information or normals. MRMesh creates this information when the mesh is loaded.

All the polygons of a mesh are not necessarily stored consecutively in the index buffer, and DirectX iterates through the entire index buffer for each set of polygons to ensure that all polygons are correctly rendered. To increase performance, we use textitDirectX.Mesh.OptimizeInPlace to attribute sort the mesh. An attribute is simply a unique texture or material combination to be used with the polygon. All polygons in a mesh have attribute IDs that reference a specific attribute. All polygons with the same attribute ID are called a subset. Attribute sorting the mesh ensures that all polygons in a subset are stored consecutively in memory. If a mesh is known to be attribute sorted, DirectX only has to iterate through the related portion in the index buffer for every subset when rendering. The attribute sorting is done at initialisation time [38].

After sorting, MRMesh iterates through the loaded material list to store the material definitions to a local material array and load in the associated texture files. The diffuse and ambient colour properties are loaded and set up.

A simplified class diagram for the MRMeshInstance class is shown in figure 3.11. MRMeshInstance inherits from the MRTransform helper class. MRTransform assists in the setting up of transform matrices from rotation, translation and scale. When we initiate a MRMeshInstance object, it receives a reference to a MRMesh object. When rendering a MRMeshInstance object,

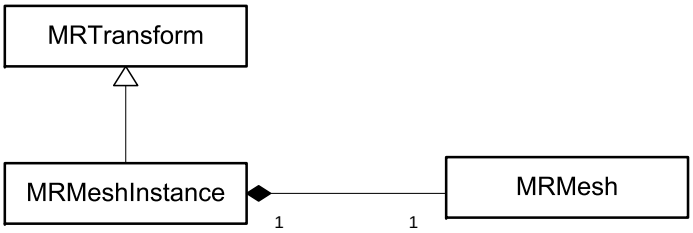


Figure 3.11: Relationship between mesh classes.

the transformation matrix from the current translation, scale and rotation properties is calculated. The mesh is rendered one subset at a time (this has been optimised by attribute sorting the MRMesh object), setting the corresponding material and texture before drawing the subset to the back buffer.

On some versions of Windows, the obtainable timestamps (from system time) can have a maximum resolution of as poor as 15 ms, dependent on underlying hardware. For many processes included in the framework this resolution is inadequate for performance measuring. The performance counter is a high-resolution hardware counter. It can be used for measuring brief periods of time with high precision [58], with a resolution of 10 ns.

The Timer object, once initiated, uses the frequency of the high-resolution performance counter, if one exists. The frequency cannot change while the system is running. It is used to measure performance when using the framework. Once the timer object has been initiated and started, the elapsed time can be checked until the timer is stopped. The average frames per second over a given time can also be found using this class.

The MRImage object is a wrapper for the OpenCV image format IplImage. It contains various commonly occurring actions like pixel-level access and colour space conversion. The pixel-level access functions simplify the task of finding the appropriate pixel in memory, and is used extensively in the framework. The class also offers a function to extract a single colour channel from a multi-channel image.

Chapter 4

Detailed Design

4.1 Introduction

This chapter describes the algorithms included in the framework. The chapter is divided into three independent sections.

Section 4.2 discusses the implemented extractor algorithms used for markerless real-world interpretation. It describes the processes related to gathering and interpreting some real-world object from an image and representing it as 2D and 3D virtual entities. Image filtering is also described in this section.

Section 4.3 presents the design of the stereo vision system. The process of calibrating and rectifying the images is described. The method used for matching pixels between these rectified images is then presented.

Section 4.4 discusses how raw video frames are displayed via the graphics device. The process of displaying 3D video is then discussed.

Two notations to describe a pixel are used in this chapter. The first is matrix notation, as used throughout earlier chapters, where a point $\mathbf{Q}$ with coordinates $(X, Y, Z, 1)$ is:

$$\mathbf{Q} = \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

The second notation represents the value of a pixel on image I with coordinates (x, y) as $I(x, y)$. This notation is used when comparing the colour values of pixels.

4.2 Image Extractors

This section discusses the implemented image extractors included with the framework. The extractors are used toe find pixels in a video frame. The framework includes three implemented image extractors:

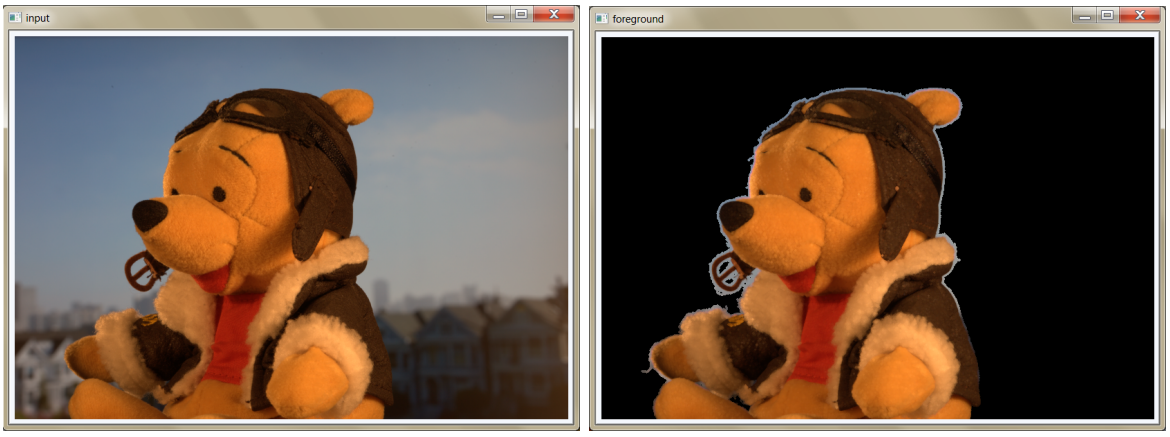
- colour extractor
- foreground extractor
- shadow suppressor

This section describes the algorithms implemented in each one of these extractors and how they can be used in a mixed reality application.

4.2.1 Foreground Extraction

Background subtraction is one of the most elementary problems in computer vision. It is a mature field offering a multitude of algorithms [59, 60, 61], with performance analysis of many of these algorithms also available [62, 63]. The definition of foreground and background can be subjective and therefore the correct separation of foreground and background also varies between cases. For the purposes of the project a point should be passed as foreground if it was not part of the original, static scene defined as background. In [64], a comparison was done between prominent background subtraction methods. The framework uses the method described as basic motion detection. This approach is chosen above the more accurate methods because of processing intensity. Since a mixed reality application built with the framework could consist of various computer vision algorithms, we implemented fast, basic algorithms for each level of extractor.

A basic colour-based method of background subtraction, or matting, is included. This method can be used for purposes including compositing [59] and tracking. Stationarity is defined as that quality of the background that a particular model assumes to be approximately constant [63]. This extractor makes the assumption that the camera and background are stationary. The simplest possible background model is used, where the background model is created from one single user defined frame. Stationarity is independently evaluated at



(a) (b)

Figure 4.1: The background subtraction process. (a) The Input Image. (b) Extracted foreground.

every pixel by comparing the point in the input frame with the corresponding pixel in the background.

During the calibration process, the user chooses the background image. For background subtraction, a basic popular threshold-based algorithm [65] was implemented. For a grey scale image, the pixel value (intensity) at location (x, y) on the background image is described with $B(x, y)$. The pixels of an image are then classified as foreground $F(x, y)$ if it differs from the background with more than a certain threshold τ. $I(x, y)$ is the pixel value in the input image at position (x, y).

$$F(x,y) = \begin{cases} I(x,y) & \text{if} \quad |I(x,y) - B(x,y)| \leq \tau \\ 0 & \text{otherwise} \end{cases} \tag{4.2.1}$$

This method has been expanded for three channel RGB (Red, Green, Blue) colour images. Two variations are provided. The first requires all three channel pixel values to pass the requirement in equation to be classified as foreground. In the framework, this extractor is referred to as ForegroundExtractor1.

A variation on this method is also included that compares the difference over all channels to a threshold to find the foreground. This extractor is called ForegroundExtractor2. The difference between the red channel colour value of $I(x, y)$ and $B(x, y)$ is given by:

$$\Delta R(x,y) = I_R(x,y) - B_R(x,y)$$

For the other channels, $\Delta G(x, y)$ and $\Delta B(x, y)$ are defined similarly. The image pixels are then evaluated according to

$$F(x,y) = \begin{cases} I(x,y) & \text{if } \left|\sqrt{\Delta R^2 + \Delta G^2(x,y) + \Delta B^2(x,y)}\right| \leq \tau \\ 0 & \text{otherwise} \end{cases} \tag{4.2.2}$$

The variation is supplied as the accuracies of the two extractors are dependent on the scene. The method and threshold τ should be chosen by the developer according to the application.

4.2.2 Colour Extraction

The foreground extractor does not distinguish between different parts of the foreground. The colour extractor is included to extract a specific item based on its colour. We wanted to include an extractor capable of finding skin colour so that the user's hand can be identified in a scene. This would mean that the user could interact with the virtual scene without the use of any peripherals. The assumption was made that a framework application would be processing intensive, therefore a rapid classification method was sought. One popular basic method is to build a skin classifier with explicit definitions for skin colour in some colour space [66, 67]. The biggest advantage of this model is that the simplicity of the skin detection rules lead to very rapid classification [68]. The main disadvantage of this method is that high recognition accuracy is dependent on the choice of colour space and that adequate classification rules have to be found empirically [68]. Taking this into account, we implemented an explicit threshold based colour extractor capable of finding human skin.

The term 'skin colour' is not an easily defined physical property of an object, but rather a perceptual phenomenon and therefore a subjective human concept [69]. As the colour extractor uses explicit thresholding, the colour space was chosen as HSV to simplify the threshold choice for the developer. Hue-saturation based colour spaces were created to meet the need for the user to specify colour properties numerically. They describe colour with intuitive values, based on the artist's idea of tint, saturation and tone [68]. Several properties of hue were noted in [70]:

- Hue is invariant to highlights from white light sources.

- Hue is invariant to ambient light and surface orientation relative to the light source for matt[1] surfaces.

This means we can define a threshold in the hue channel to find skin colour in a scene, while accepting a wide range in tone [71]. The extractor receives an RGB image and firstly converts it into HSV colour space. With $I_H(x, y)$ the hue channel value of a pixel in the input image, the pixels in the output image $F(x, y)$ are evaluated according to

$$F(x,y) = \begin{cases} I(x,y) & \text{if} \quad \tau_{H1} \leq I_H(x,y) \leq \tau_{H2} \\ 0 & \text{otherwise} \end{cases} \tag{4.2.3}$$

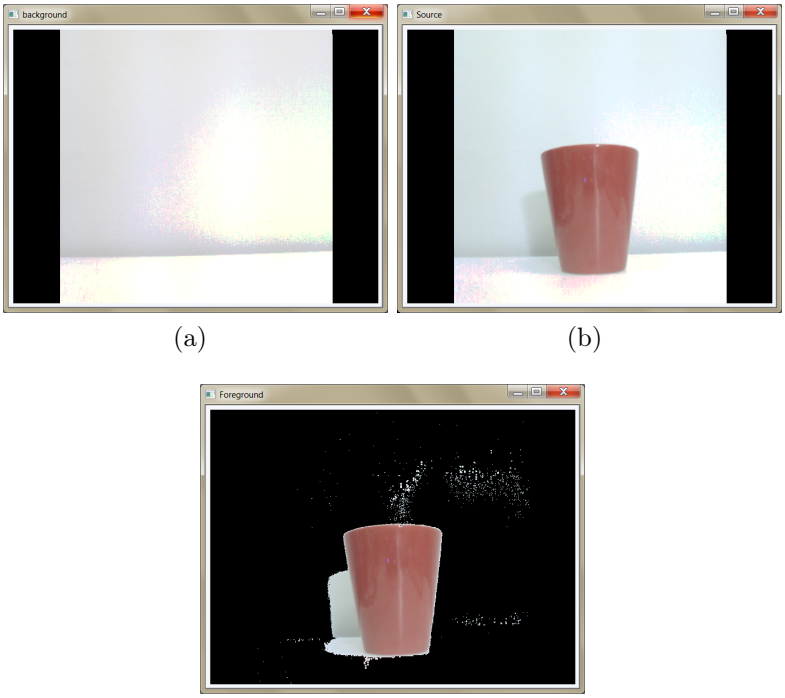
The range τ_{H2} to τ_{H2} is chosen by the developer. To expand the extractor to a more general colour extractor that can be used to extract other objects based on their colour, optional thresholds for the saturation and value channels were added. These thresholds would also assist in removing unwanted noise.

4.2.2.1 Shadow Suppression

With either of the discussed extractors, directional light sources might lead to shadows in the scene that might be included in the extractor output. For both extractors this is undesirable, as we would like to find the area in the image relating to the object, and not the object and its shadows. This is especially problematic for foreground extraction. Pixels containing the shadows of foreground objects have different values to the background image, and depending on the threshold, will likely be passed as foreground. These shadows have to be identified and removed.

Like background subtraction, shadow suppression has been quite extensively studied [72, 73, 74]. In comparisons made between various shadow suppression algorithms [72], methods based on HSV colour space are the most generally capable and robust [74]. This is because the HSV colour space corresponds closely to human perception of colour [73]. In [75] Cucchiara presents a deterministic non-model based approach to shadow detection in HSV colour space. This algorithm uses the observation that the saturation is usually lowered by the presence of a shadow at a point in the background. Another

[1] A matt surface refers to a surface that reflects light diffusely.



(a) (b)

(c)

Figure 4.2: Shadows incorrectly classified as foreground. (a) The background image. (b) The object's shadow is visible in the scene. (c) The extracted foreground includes shadows.

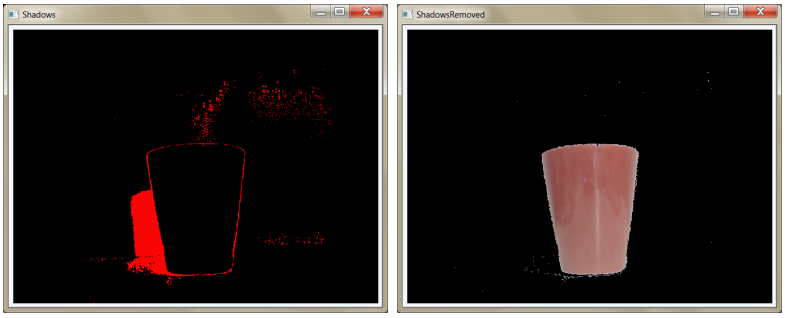
constraint follows because the shadow does not significantly change the hue value. This leads to the decision equation

$$F(x,y) = \begin{cases} 1 \quad \text{if} \quad \alpha \leq I_V(x,y)/B_V(x,y) \leq \beta \\ \qquad\qquad \wedge\, |I_H - B_H| \leq \tau_H \\ \qquad\qquad \wedge [I_S(x,y) - B_S(x,y)] \leq \tau_S \\ 0 \qquad\qquad \text{Otherwise} \end{cases} \tag{4.2.4}$$

The value α takes the intensity of the radiance source into account. As the light source increases, α should be lowered. To avoid the misclassification of foreground as shadow due to noise, β is used. Figure 4.3b shows typical output for this extractor.

4.2.3 Conditioning

Due to noise in images, lighting inconsistency, and other problems associated with practical implementations, it is almost certain that output images from



(a) (b)

Figure 4.3: Shadow suppression. (a) Shadows are removed from figure 4.3a. (b) The resulting foreground (before post-processing).

the image extractors contain noise, as is the case in figure 4.3b. The framework contains a post-processing filter to remove this noise.

Mathematical morphology in digital images is a set of image processing operations that process images based on shape [76, 77]. The VisualCortex wraps two mathematical morphology operations implemented in OpenCV to filter the image: erosion and dilation. The operations are described in detail in [62] and [78]. The OpenCV implementation is described in [2]. This section only gives a basic overview of the operations, and describe how they are used in the framework.

With binary input image I and structuring element (kernel) A, erosion is defined by

$$I \ominus A = \bigcap_{a \in A} I_{-a}.$$

Dilation is defined by

$$I \oplus A = \bigcup_{a \in A} I_a.$$

Erosion can be used to removed unwanted noise (isolated pixels) classified as foreground. Dilating can be used to filter holes in foreground objects classified as background [62].

Using erosion and dilation, this framework implements an operation known as opening [78]. The opening of I by A is found by erosion of I by A, followed by dilation by the same kernel A.

$$I \circ A = [(I \ominus A) \oplus A]$$

The combination of erosion and dilation first removes isolated pixels classified as foreground, and then removes holes caused by isolated pixels classified as background.

4.2.4 2D Body Extractor

In an attempt to extract some intelligible information from the image extractions, we follow the example given in typical physics engines [34] by defining objects in space with bounding shapes (boxes or ellipsoids).

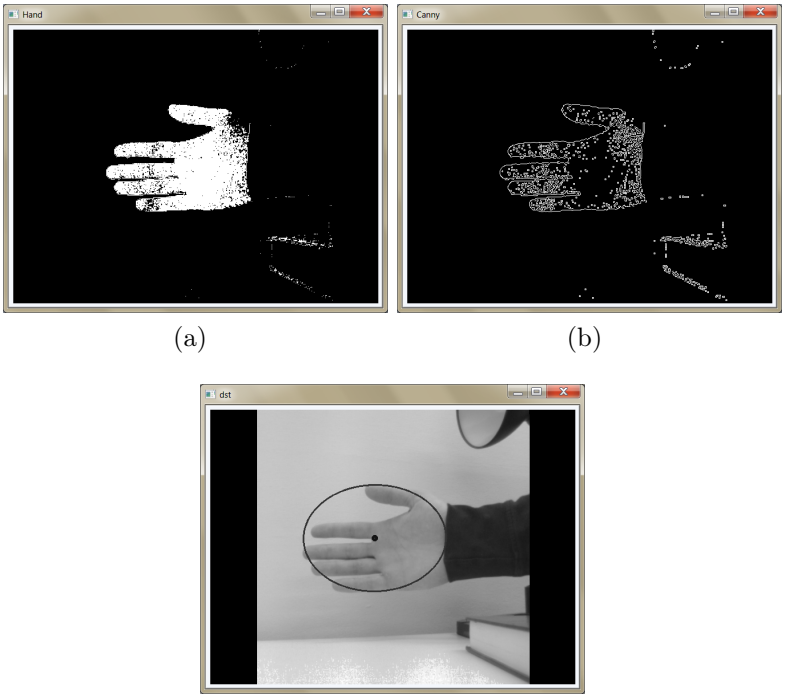
4.2.4.1 Bounding Shape Extractors

The 2D body extractors use a binary version of the output from the image extractors (with the foreground or colour extractor pixels set to 1 and the background to 0). Edges are then found using Canny edge detection [79]. A discrete analogue of the Laplacian operator is used to find the second-order derivatives in the x and y directions as G_x and G_y. The edge gradient G and direction Θ_G can then be found:

$$G = \sqrt{G_x + G_y}$$
$$\Theta_G = \arctan \frac{G_y}{G_x}$$

Edges are then rounded to one of four angles: 0°, 45°, 90° or 135°. These edges are then assembled into contours, using a hysteresis threshold. The threshold consists of a high and low threshold. If the pixel has a value higher than the upper threshold, it is excepted as an edge. If the pixel has a value lower than the lower threshold, it is rejected. It the pixel value is between the thresholds, it is accepted as an edge if one of its neighbouring pixels is above the high threshold. The thresholds should have a ratio of 1:2 to 1:3 [79]. This function returns a binary image containing the edges. OpenCV contours are found from these edges. OpenCV contours are segments of super pixels with knowledge of how they relate to each other, where super pixels refer to perceptually meaningful entities [2].

As shown in figure 4.4, the Canny edge detector may find many edges in an image from the image extractors. As the image extractors should be interchangeable, no assumptions are made about the shape of the edges. The 2D body extractors assume that the input image has been processed by one



(a) (b)

(c)

Figure 4.4: Finding a bounding shape. (a) Skin pixels extracted from an image. (b) The canny edges, with the largest contour surrounding the hand. (c) The resulting bounding box and ellipsoid.

of the the image extractors. Consequently, the assumption is made that the largest closed contour should contain the object that has to be found. The areas contained in the contours in the image are compared, and the largest one is kept to represent the object. A bounding (rectangular) box is found from the contour, and from the center point and size of the bounding box a bounding ellipsoid can be found. OpenCV provides functions to do basic geometric calculations on contours and bounding boxes. A basic, CPU-based collision detection can be done by comparing the depth of two boxes and then using OpenCV's polygon overlap check. This is demonstrated in section 5.4. The bounding box can be used for collision detection in a third-party physics engine. Direct3D supports GPU-based collision detection. The bounding ellipsoid variation can be used if a better fitted bounding shape is needed for some object.

4.2.5 3D Body Extractor

Depending on the application, the designer might want a 3D representation of the 2D body found with the extractors discussed in section 4.2.4. With the currently implemented extractors, the object of interest could be a hand or a human body, which are not rigid objects. As a first proof of concept we have only implemented a 2D body extractor capable of extracting a bounding box or ellipsoid. Similarly, the 3D object extractor will only provide a basic, rigid representation of the object. For the sake of versatility, no prior knowledge of the (3D) shape of the object is taken into account. We combine the 2D bounding shape found as described in the previous section with the disparity map included in the VisualCortex to find a sphere in 3D space as a basic representation the object.

4.2.5.1 3D Picking

The object is defined in screen space by its 2D bounding shape. The VisualCortex includes a disparity map in DepthMapper, containing the disparity of points on the image. The disparity is found by comparing the point on images from the left and right cameras. The method used to find the disparity is discussed in section 2.5.5. The disparity relating to a point on an image, e.g. the origin of the bounding shape, can be directly read from the corresponding point on the disparity map. We are interested in the depth of this image point. The depth Z can be found from the disparity, d. The disparity now has to be interpreted as depth, Z. The normalised single-channel disparity map returned by the framework displays 0 disparity as white (255), and the maximum disparity as 1. Black (0) is reserved for unmatched pixels. Recalling that $d \propto Z$,

$$Z = K_1/(255 - d) + K_2 \tag{4.2.5}$$

where K_1 and K_2 are constant values. From [2] with interaxial distance B and focal length F, the real-world depth can be found with:

$$K_1 = (F - B) \tag{4.2.6}$$

$$K_2 = 0 \tag{4.2.7}$$

In a framework application, there are two sets of 3D coordinates, the coordinates for the real-world, (in which F and B are defined), and the coordinates

for the virtual world. We are interested in relating the position of the real-world object to the virtual space. Consequently, we do not find K_1 and K_2 as shown in equations 4.2.6 and 4.2.7. Instead, it is found using the virtual pose found from the fiducial marker tracker. In calibration, we use the ForegroundExtractor and the BoundingBoxExtractor to find a 2D representation of the marker placed perpendicular to the camera. The disparity, d_E for the center point of the bounding box is then found. At the same position, the marker pose is found with the marker tracker included in the VisualCortex, and from the depth Z_M is read. With two different instances of the disparity from the extractors d_E and the depth from the marker tracker Z_M, we can find values for K_1 and K_2. The calibration process is shown in section 5.2.3.

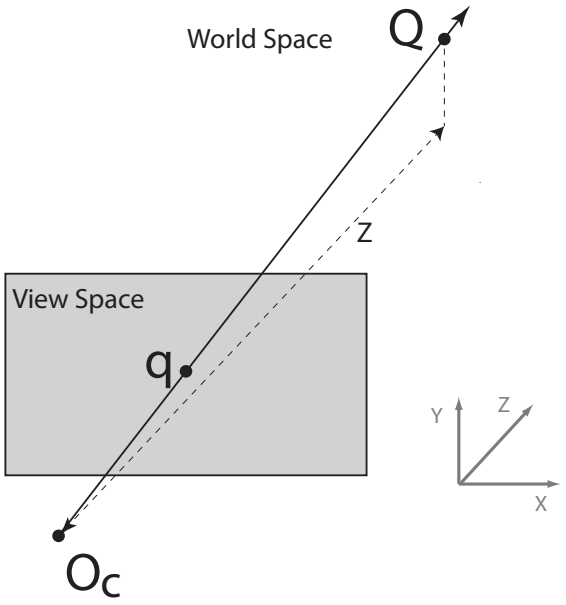


Figure 4.5: Displaying a pair of stereo images.

The x and y points obtained from the 2D bounding box are defined in screen, or view space. To find the 3D X and Y (virtual) position of the relating point, a method based on 3D picking, or ray tracing, [57] is used. The top left corner of the image has the coordinates $(0, 0)$. Point $\mathbf{q}$ has coordinates (x, y) on screen. Adjusting $\mathbf{q}$ according to the Direct3D window and perspective projection $\mathbf{P}$ matrices, yields the new coordinates $\mathbf{q_{vs}}$ in view space.

$$\mathbf{q_{vs}} = \begin{bmatrix} (\frac{2x}{w_B} - 1)/\mathbf{P}_{11} \\ (\frac{2y}{h_B} - 1)/\mathbf{P}_{22} \\ 1 \end{bmatrix} \tag{4.2.8}$$

Where $z = 1$ is used, since it is the defined depth of the screen in virtual space. Now $\mathbf{q_{vs}}$ can be transformed to world space as $\mathbf{q_{ws}}$. World space is the virtual 3D coordinate system in which the developer places the virtual objects.

$$\mathbf{q_{ws}} = \mathbf{Q_{vs}}\mathbf{V^{-1}} \tag{4.2.9}$$

With $\mathbf{q_{ws}}$ we have the screen point position in the virtual world. From $\mathbf{V^{-1}}$ we can also find the camera coordinates $\mathbf{O_c}$.

$$\mathbf{O_c} = \begin{bmatrix} \mathbf{V^{-1}_{41}} \\ \mathbf{V^{-1}_{42}} \\ \mathbf{V^{-1}_{43}} \end{bmatrix} \tag{4.2.10}$$

As shown in figure 4.9 $\mathbf{q_w s}$ gives the direction of the ray R_d from origin $\mathbf{O_c}$ on which the point Q lies. In the framework the camera is defined at $(0, 0, 0)$. Consequently, the point $\mathbf{Q}$ is located in world space at

$$\mathbf{Q_{ws}} = \begin{bmatrix} \mathbf{ZR_{d\ 1}} \\ \mathbf{ZR_{d\ 2}} \\ \mathbf{Z} \end{bmatrix} \tag{4.2.11}$$

With $Z = Z_M$, the coordinates for $\mathbf{Q}$ are found.

4.3 Stereoscopic Vision

The depth of a point found from the 3D body extractor is read from a real-time disparity map included in the VisualCortex. The developer can find the depth of a point in a scene using this map. Section 2.5 explained the steps that are needed to undistort and rectify a stereo rig. The IDepthMapper contains wrapped OpenCV functions to calibrate and rectify the stereo cameras as described in section 2.5. Once the images from the two cameras are rectified, the framework can construct a disparity map. This section describes this process.

4.3.1 Stereo Correspondence and Disparity

With the images row aligned, the depth of features from the two images can be found. With point **Q** found in both images, i.e. $\mathbf{q_l}$ and $\mathbf{q_r}$, we can find the depth of **Q**. For stereo correspondence, the framework implements block matching. It has relatively low processing cost, and has sufficient accuracy [80]. OpenCV contains a fast and effective block-matching stereo algorithm, based on [81]. The algorithm utilises a sum of absolute differences (SAD) window to match points from the left image with those in the right. This is a one-dimensional search along the epipolar lines, since the images are row aligned after rectification. As it is a colour-based SAD, it should be noted that it performs better in texture-rich scenes than than monotonous scenes. The algorithm contains three steps [2]:

1. Prefiltering: normalise image brightness and enhance texture.
2. Correspondence search.
3. Postfiltering: eliminate bad matches.

In prefiltering, the image is normalised to account for light differences and noise. There are two prefilter options, a Laplacian of Gaussian peak detector [82], or a normalised response method [2]. The normalised response method is used by default by the framework as it is computationally less expensive [2].

See figure 4.6. The correspondence is calculated with the sliding SAD window. For a feature in the left image, the best match in the right image along the same horizontal row is found, as the images are row aligned. Because the images are arranged in frontal parallel, the starting point and search direction are chosen to further optimise the function. If a feature is located at pixel $q = (x_0, y_0)$, in the left image, the search starts at position (x_0, y_0) in the right image. The feature will be located to the left of (x_0, y_0) as the images are parallel. This is presumed as the minimum disparity should be 0 in a frontal parallel configured rig.

The horopter is defined as the disparity search field, and is adjusted by changing the minimum and maximum disparities for matching. The horopter can be set to adjust the depth map according to the application.

The OpenCV block matching function typically has the characteristic of a strong central peak surrounded by weak side lobes [83]. Figure 4.6 shows a typ-

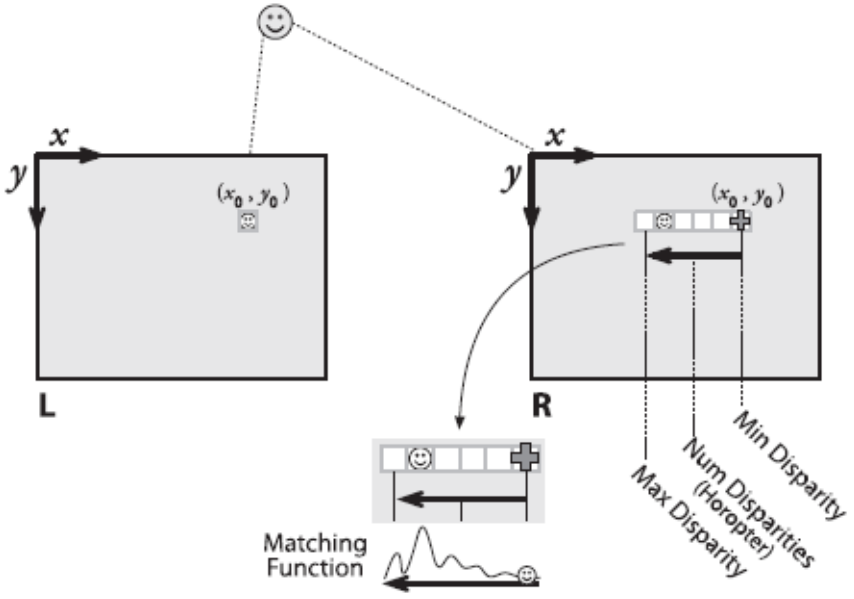


Figure 4.6: Block-matching features with two images. Figure from [2].

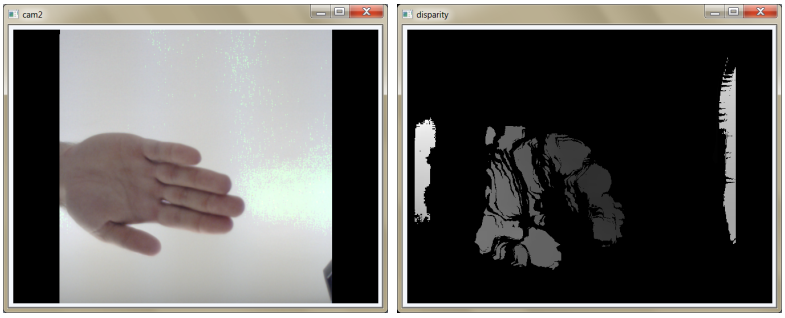
ical function response. The function makes use of this pattern by eliminating incorrect matches in the postfiltering stage as described in [2].

4.3.2 Finding a depth map with MxRFramework.

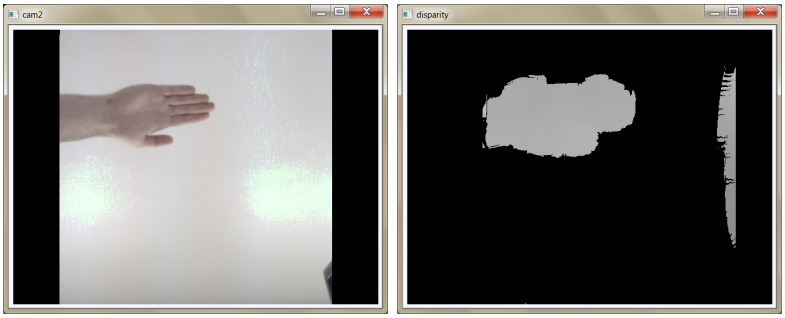
The framework offers the developer stereo vision functionality via a simple interface so that she can concentrate on the content of her mixed reality application. The complexities associated with this process have been hidden from the developer by setting up default parameters, while still making the parameters changeable if needed. All matrices are hidden, and throughout the process the developer only interacts with the raw image, the rectified image and the disparity map. The disparity map can be found with a few steps:

- Add calibration images.
- Calibrate the stereo cameras (or load calibration).
- Query the Visual Cortex for the rectified image or the disparity map.

Once calibrated, the mappings can be exported to an Extensible Markup Language (XML) file, and loaded for later use with the same camera set-up. Listing 4.1 shows all the code needed to show the rectified image pair and find the



(a) (b)



(c) (d)

Figure 4.7: (a) The input image from one camera. (b) The resulting depth map. (c) The Input image with hand held further away. (d) The resulting depth map.

disparity map, where *src* is an array of type MRImage containing images from the cameras, and *disparity* is a single channel *MRImage* entity. The calibration process can be replaced with the *LoadCalibration()* function if calibration has been done previously. Calibration and rectification are handled in the *EndCalibrate()* function, resulting in a pair of rectified maps stored in the VisualCortex object. These maps are used to remap the camera images to frontal parallel configuration, before block matching is applied. The remapping and matching is encapsulated in the *GetDisparity()* function. Figure 4.7 shows the resulting depth maps from *GetDisparity()*.

4.4 Stereoscopic Display

Nvidia's 3D vision drivers are implemented to support stereoscopic display. Two different types of stereo content can be displayed, the stereoscopic rendering of a 3D scene, and the stereoscopic display of two existing stereoscopic

Listing 4.1: Using a stereo rig with MxRFramework.

```
if(stereo)
    while(true)
    {
      ....
      if(calibrateCondition)
        if(!cortex->DepthMapper->AddCalibrateImage(src))
          ...
      else if(calibrateComplete)
      {
        cortex->DepthMapper->EndCalibrate();
        cortex->DepthMapper->SaveCalibration();
      }
      else
      {
        cortex->DepthMapper->ShowRectified(src);
        cortex->DepthMapper->GetDisparity(src,disparity);
      }
    }
```

images. Section 2.6.2 discusses how to render a 3D scene correctly. The driver is capable of displaying existing stereoscopic images, but it is not exposed by the NVidia API (NVAPI). This section presents the method used to display prerendered stereoscopic frames.

4.4.1 Stereoscopic Image Display

For a still stereoscopic frame, the stereo images have to be loaded into Direct3D device memory (the default memory pool) [4]. The frame also has to be recognisable as a stereo entity by the stereoscopic 3D driver. For this, a Direct3D texture surface is created as shown in listing 4.2.

Listing 4.2: The stereo off-screen surface.

```

  d3dDevice->CreateOffscreenPlainSurface
                    (imageWidth*2,      //width: left and right image
                    imageHeight + 1,    //height: height+stereo tag
                    D3DFMT_A8R8G8B8,            //Pixel format
                    D3DPOOL_DEFAULT,            //Memory pool
                    &imgSrc,                    //SurfaceName
                    NULL);
```

On a 3D-vision driver-enabled platform, a surface is recognised as a stereo off-screen surface if the surface is allocated to the default memory pool, and the surface is in the format shown in figure 4.8 [4]. The surface size is defined to hold the two images, and a stereo signature, identifying the surface as stereo.

The stereo signature is to be placed in memory corresponding to the beginning of the last row of pixels on the surface.

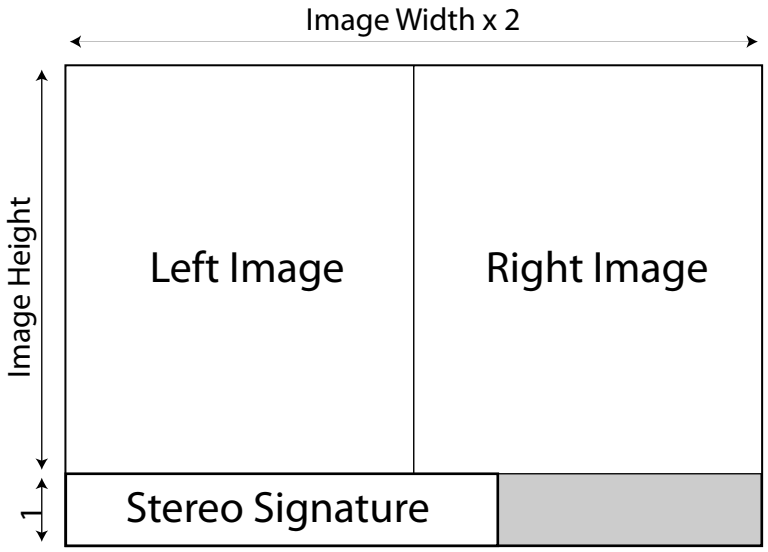


Figure 4.8: Stereo frame format.

The surface is 'locked' [84] before writing the signature to it. Locking the surface memory is a relatively slow algorithm, and not ideal for real time operations [30, 85]. Since the signature is constant, it can be inserted at initialisation time. When editing the surface in real time, care has to be taken not to write to memory allocated for the signature.

In the main program loop, a stereo frame is loaded into the surface from system memory. The two images are allocated to surface memory corresponding to the locations shown in figure 4.8.

It should be noted that the function *D3DXLoadSurfaceFromMemory*, that copies data from system memory to Video RAM stops the 3D accelerator while the data is loaded to the VRAM [84]. At this stage we cannot bypass this limitation with better memory pool use, since the *StretchRect()* function needed limits the memory pools and surfaces that are usable.

To display the surface containing the video, it has to be placed in the back buffer. For the stereo driver to function correctly:

1. The back buffer should match the screen size (and run full screen mode).

Listing 4.3: Recognising a surface as a stereo pair.

```
1   //writing stereo signature to last RAW of surface
2   D3DLOCKED_RECT lr;
3   //lock access to surface
4   imgSrc->LockRect(&lr,NULL,0);
5
6   LPNVSTEREOIMAGEHEADER pSIH =
7                 (LPNVSTEREOIMAGEHEADER)
8                 (((unsigned char *) lr.pBits) +
9                 (lr.Pitch * (imageHeight)));
10
11  //predefined values to set up stereo display
12  pSIH->dwSignature = NVSTEREO_IMAGE_SIGNATURE; //recognise as
13                                                //stereo image pair
14  pSIH->dwBPP = 32;
15  pSIH->dwFlags=  SIH_SCALE_TO_FIT |       //prevents repetition
16                  SIH_SCALE_TO_FIT2;       //& display problems
17
18  imgSrc->UnlockRect();
```

Listing 4.4: Storing the left and right image to the off-screen stereo surface.

```
1  //block corresponding to
2  //left image position
3  resizeRect.left = {0,0,i imageHeight,imageWidth}
4
5  D3DXLoadSurfaceFromMemory
6      (imgSrc,
7      NULL,&resizeRect,
8      leftImagePtr,      //source image
9      D3DFMT_A8R8G8B8, 2560,
10     NULL,&videoRect, //corresponds to image size
11     D3DX_DEFAULT, 0);
12 //block corresponding to
13 //right image position
14 resizeRect.left = {imageWidth, 0, imageHeight, imageWidth*2};
15
16 D3DXLoadSurfaceFromMemory
17     (imgSrc,
18     NULL,&resizeRect,
19     rightImagePtr, //source image
20     D3DFMT_A8R8G8B8, 2560,
21     NULL,&videoRect, //corresponds to image size
22     D3DX_DEFAULT, 0);
```

2. *StretchRect()* should be used to match the stereo surface size to the back buffer size.

With quad buffering, it would be possible to display stereo in windowed mode (not full screen) on the screen. On a standard non-professional graphics device, as used in this project, quad buffering is not supported and therefore only full-screen stereo display is possible. Failure to comply with point 2 leads to failure in the stereo display process.

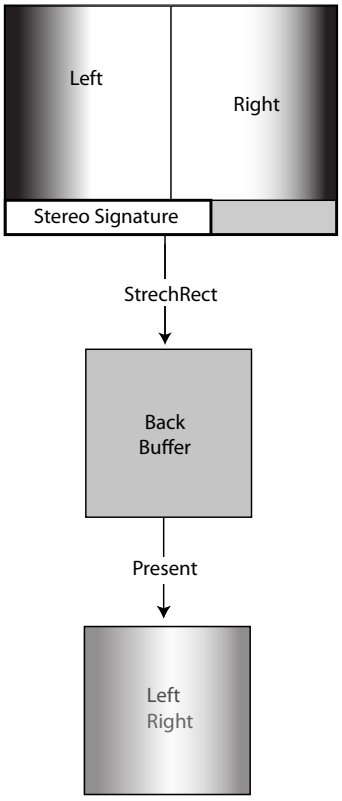


Figure 4.9: Displaying a pair of stereo images.

See figure 4.9. With the stereo frame information in the back buffer, the back buffer is presented to display the images in 3D with a compatible display device. The method used for displaying 2D video is similar, using a normal surface instead of the stereo surface.

Chapter 5

Tests and Implementation

5.1 Introduction

This chapter presents the tests done on the completed framework. The next section describes how the performance and accuracy of framework functionality was tested. All tests were done on a single PC, with specifications as presented in appendix B.

The rest of the chapter is dedicated to the testing of framework implementation. Section 5.3 presents the design of a typical marker-based AR application, and compares it to it with similar applications made with ARToolkit and GoblinXNA. The application is then expanded to use hand-tracking. To demonstrate how the framework can be used to create other mixed-reality applications, a mixed-reality interface using a 3D display is presented in section 5.4. It makes use of the out-of-screen effect to offer users an interface in 3D space, usable with no peripherals.

The framework designed in this project is referred to as the MxRFramework (mixed-reality Framework), in order to avoid confusion when comparing it to other solutions.

5.2 Performance Tests

5.2.0.1 Processing Times

In a mixed-reality application the mixed environment has to be updated fast enough for the user to perceive it as real time. Therefore, knowing the time

it takes to process each video frame is important. This test presents the processing time of all the extractor components contained in the framework. Along with the graphical methods, these are the processes that would take place in the program loop.

5.2.0.2 Method

The processing times of the algorithms were evaluated at three different resolutions, VGA (640×480), QVGA (320×240), and half-QVGA (160×120). The test platform's specifications are given in appendix B.

5.2.1 Results

Table 5.1: Processing time of framework algorithms ([1]OpenCV based functions).

	Processing Time (ms)		
	Half QVGA	**QVGA**	**VGA**
Conditioning			
Erode[1]	0.488	1.653	6.542
Dilate[1]	0.502	1.583	6.519
HSV Conversion[1]	0.337	1.317	5.640
Shadow Suppression	11.027	45.301	203.182
Foreground Extraction			
ForegroundExtractor1 GS	5.312	17.571	100.005
ForegroundExtractor1 RGB	8.641	23.254	148.951
ForegroundExtractor2	13.894	48.645	333.334
Colour Extraction[1]			
	0.504	1.830	7.570
2D Object Extraction[1]			
Find Bounding Shape	68.931	176.186	461.634
Stereo Vision[1]			
Find Disparity Map	0.503	1.684	7.956
3D Object Extraction			
Find 3D Point Coordinates (GPU)	3.153×10^{-3}		

The extent of OpenCV's optimisation becomes apparent in the results shown in table 5.1. Even basic pixel comparisons done without OpenCV functions heavily impact the processing time, as seen in the foreground extractor and the shadow suppression. The erosion algorithm, for example, uses a SAD and yet it outperforms the extractor functions due to OpenCV's optimisation. This indicates that there is much room for improvement in the extractor algorithms. The bounding box algorithm is composed out of canny edge detection, segment creation, contour comparison and the creation of the bounding shapes. Creating segments from edges, and comparing the segments that form closed contours, are processor-intensive operations. Even though the process uses optimised OpenCV functions, it performs poorly at high resolution.

5.2.2 Extractor Accuracy Tests

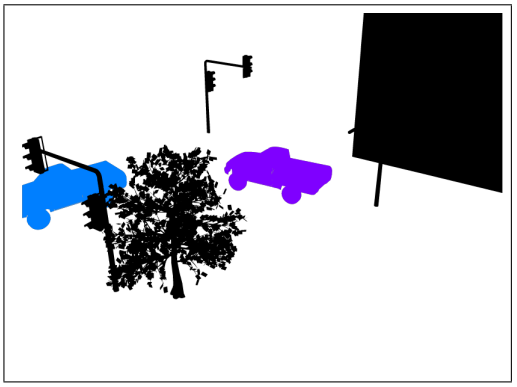
In this test the ability of the framework algorithms to find and classify a object in a video is analysed. The Stuttgart artificial background subtraction dataset [86] is used to evaluate the algorithms.

The dataset contains ground truth masks for every input frame to compare with the foreground found from each algorithm. Appendix C contains the dataset used for testing. Some typical challenges of background extraction are tested with the dataset[86]:

- **Gradual illumination changes.** The ambient light in the scene slowly changes over time.
- **Sudden illumination changes.** Traffic light in the scene produce sudden changes in directional light.
- **Dynamic Background.** The scene contains a tree with moving branches.
- **Reflection.** Window buildings reflect objects moving past.
- **Camouflage.** Foreground objects are obscured by the tree branches.
- **Video noise.** Camera sensor noise is simulated using additive Gaussian noise.

The use of artificial data enables us to separably judge the performance of the background subtraction methods and the impact of the filters. The colour

(a) (b)



(c)

Figure 5.1: Samples from the artificial extractor test set. (a) The background reference image. (b) A sample input image. (c) Ground truth image for the image in 5.1b.

extractor can also be tested using the dataset by searching for foreground items of a certain colour.

5.2.2.1 Method

The 30 test images in appendix C were evaluated against the ground truth foreground for each image. A sample from the set is shown in figure 5.1. All the images were separately processed by the two three-channel foreground extractors and the colour extractor. The foreground extractors compare the image to the background image shown in appendix C. ForegroundExtractor1 classifies foreground according to equation 4.2.3 and ForegroundExtractor2 according to equation 4.2.2. The colour extractor, referred to in this test as ColourExtractor, attempts to find the red-coloured vehicles. The background

subtraction dataset can be used for the colour test since vehicles are classified as foreground in the ground truth images.

The test set was processed by the extractors. The extractor output pixels were then compared against the ground truth for each image. For every frame found the total foreground (or target) pixels correctly identified (T_C), the total background pixels incorrectly identified (T_I), and the number of foreground pixels in the ground truth image T_{GT} are found. The accuracy is then expressed relative to the ground truth foreground.

$$V_C = T_C/T_{GT}$$

$$V_I = T_I/T_{GT}$$

The shadow suppressor and the eroding and dilating filter were added to enhance each extractor. The bounding box extractor was then used to process these images. There are multiple foreground objects in each scene and the bounding box extractor only finds the most prominent object. Because of this we tested whether the bounding box extractor correctly found any one of the foreground objects in a given scene.

The dimensions of a bounding box is compared to the dimensions of the object it encloses. The width of an object, w_O, is defined as the distance between the leftmost pixel and the rightmost pixel found as part of the object in the ground truth image. The height of an object, h_O, is defined as the distance between the highest and lowest pixels of the object in the ground truth image. The height of the object's bounding box is h_B and the width w_B. We define two ratios, α_w and α_h with

$$\alpha_w = \frac{|w_B - w_O|}{w_O} \tag{5.2.1}$$

$$\alpha_h = \frac{|h_B - h_O|}{h_O} \tag{5.2.2}$$

The bounding box extractor is then evaluated with values[2]

$$\alpha_w < 0.25 \tag{5.2.3}$$

$$\alpha_h < 0.25 \tag{5.2.4}$$

A bounding box is accepted as a successful match if conditions 5.2.3 and 5.2.3 are met. Refer to figure 5.1. Note that the wheels and windows are classified

[2]These values where found to be sufficient by trial and error.

part of the ground truth foreground. The colour extractor uses colour thresholding, and rejects these parts. For this reason the colour extractor is only evaluated according to condition 5.2.4.

To standardise the test across the three extractors the thresholds were set using a single calibration image. The threshold values for the extractors were set to achieve the restriction set by 5.2.3 and 5.2.4, while minimising V_I.

5.2.2.2 Results

Graphs and output images relating to this experiment can be found in appendix D. ForegroundExtractor1's accuracy is given in figure D.1. ForegroundExtractor2's accuracy is shown in figure D.2. Setting the thresholds to maximise the number of successful bounding box matches causes the extractor to return high values for V_C at the cost of high values for V_I. The output images are shown in section D.2.1. The peaks in V_I values in the proximity of frames 5 and 22 are due to the low value of V_C, as the foreground object moves behind the moving tree. The shadow suppressor significantly lowers the V_I, but also lowers V_C as the dark tyres and windows are confused with shadows. Dilation is set up with a large (7×7) kernel that removes any holes made in the foreground object by the extractors. The size of the dilation kernel does lead to incorrect classification around the edges of the object. Results for ColourExtractor are shown in figure D.3. Because the ground truth was used to set up the extractor thresholds, the shadow suppressor (named ShadowSuppressor) was emitted from the results. Both ColourExtractor and ShadowSuppressor use HSV thresholding. With thresholds for ColourExtractor set up as discussed in section 5.2.2.1, the addition of ShadowSuppressor could not improve results.

Table 5.2 shows the average image extractor results. Filters refer to shadow subraction and the erosion and dilation filter. Without filters, ForegroundExtractor1 fares significantly worse than ForegroundExtractor2. With the erosion and dilation filter, ForegroundExtractor1 has slightly better results than ForegroundExtractor2. This suggests that a large section of the wrongly classified pixels from ForegroundExtractor1 were isolated.

ColourExtractor's average V_I increases significantly when the filtering is added. This increase is due to the large size of the dilation kernel used. This test set is not ideal for a colour extraction test as the filters have to compensate

Table 5.2: Accuracy average for the dataset.

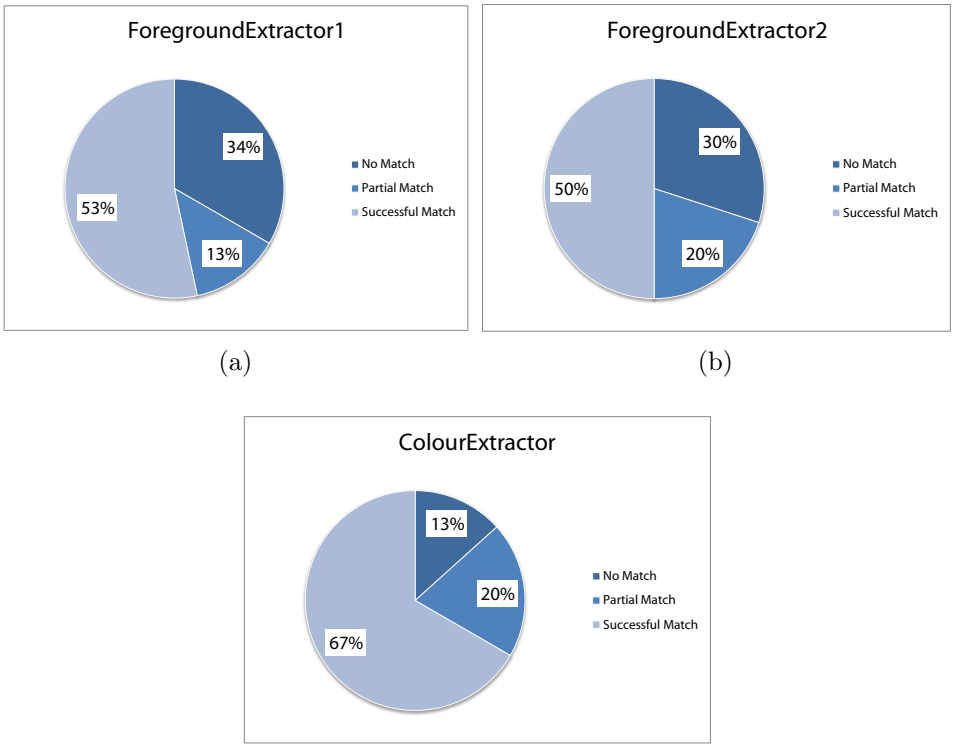
Average over input images (%)		
	Target pixels found (V_C)	**Incorrect pixels found (V_I)**
	Without filters	
ForegroundExtractor1	75.356	178.567
ForegroundExtractor2	88.920	132.523
ColourExtractor	68.211	31.531
	With filters	
ForegroundExtractor1	96.751	94.667
ForegroundExtractor2	94.667	110.467
ColourExtractor	84.210	291.155

for the large windows. This causes the low average V_C for the ColourExtractor without the filters. What this test does indicate is that the filtering can make the extractors more robust in imperfect real world conditions[3].

Section D.2 in appendix D contains the output images showing the bounding box matches, also summed up in figure 5.2. The criteria for a successful match was given in section 5.2.2.1. A partial match occurs when the center point of the bounding box is situated on the target object, but the box does not adhere to the criteria for a successful match.

In all three extractors we see reoccurring reasons for incorrect placement of the bounding box. When a foreground object enters the scene, the contour surrounding it is often cut off by the edge of the screen and therefore not a closed contour. The dynamic background also leads to incorrect placement. The moving tree branches and foreground object reflection in windows are often not removed and cause incorrect placement of the bounding box. The combination of image extractors and the bounding box extractor fares well in scenes where foreground objects are not obscured. This is a positive result, as it is closer to the circumstances that we will have in a controlled augmented reality environment.

[3]Take into account that the main objective here was to find a contour surrounding the object, and not to minimise V_I.



(a) (b)

(c)

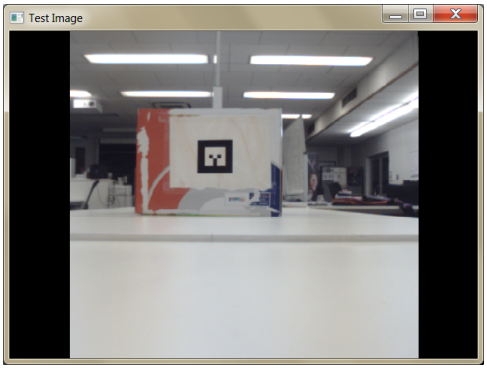
Figure 5.2: The accuracy of BoundingBoxExtractor.

5.2.3 Stereo Vision Tests

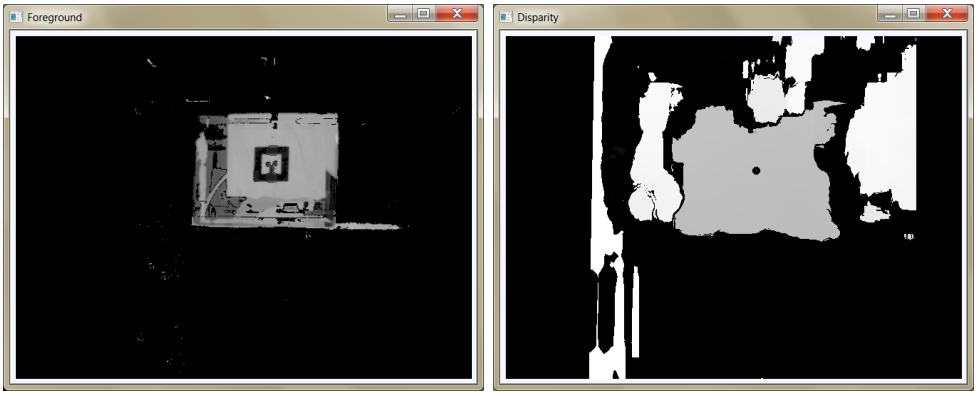
For the depth of a real scene we find a relative depth from the disparity map, and test the accuracy by comparing it to real-world coordinates. The framework does include a high-accuracy third-party fiducial marker tracker [87]. The depth found with the tracker and the disparity map are compared.

5.2.3.1 Method

A marker is placed perpendicular to the camera at a distance of 500 mm. The position of the marker is found using the marker tracker. Foreground extraction is performed on the scene, and the depth is found from the disparity map, as shown in figure 5.3. The disparity map depth, the marker tracker depth, and the physical distance from the camera are then recorded. The marker is moved away from its original position in increments of 100 mm.



(a)



(b) (c)

Figure 5.3: Comparing depth from the marker tracker and the disparity map. (a) The input image contains a marker, the depth is found with the marker tracker. (b) The marker is identified as foreground. (c) The depth of the marker is found from the depth map.

5.2.3.2 Results

Figure 5.4 shows the depth found by the marker tracker and disparity map. The error made relative to the physical position (measured position) is shown in figure 5.5. The population standard deviations(ϵ) for the marker tracker and the disparity map are shown in table 5.3. The values in table 5.3 the measured position of the marker as the expected positions.

While the disparity map fares worse than the marker tracker, the errors are very small relative to the marker's physical depth. This indicates that it is plausible to use the depth obtained from these two sources in the same application.

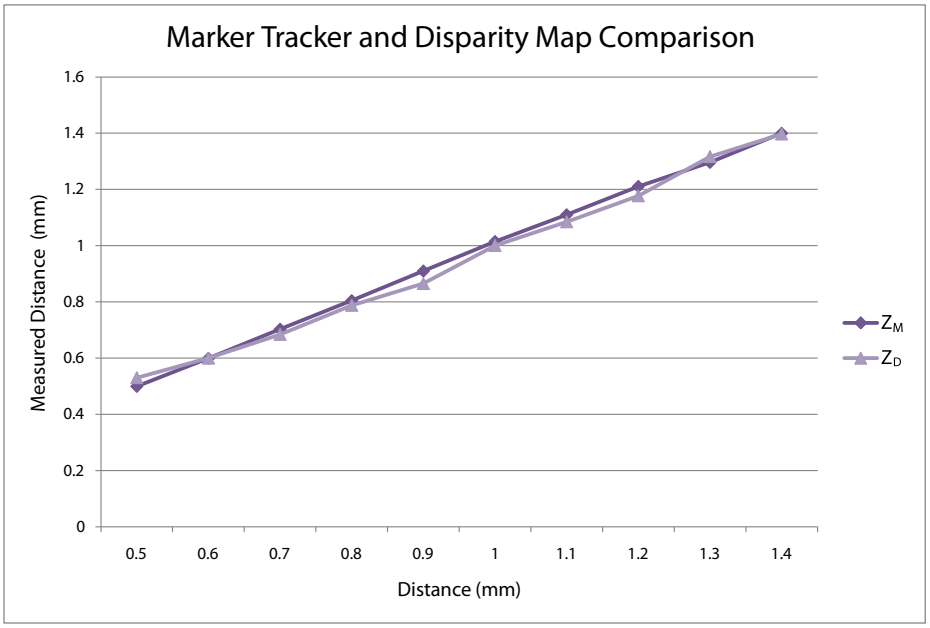


Figure 5.4: Comparison between the distance from the marker tracker and disparity map.

Table 5.3: Error in depth.

	ϵ (mm)	Average error (%)
Marker tracker	7.718	0.583
Disparity map	18.627	1.847

5.3 Implementing an Augmented Reality Application

One of the main objectives of this project is to create a framework for the rapid development of augmented reality applications. This section describes the design process of a typical marker-based AR application. The performance of an AR application made with this framework, referred to as MxRFramework, is then compared to the most prominent solutions at the time of writing, ARToolkit and GoblinXNA. In section 5.5 the AR application is expanded to use some of the other tools made available by the MxRFramework.

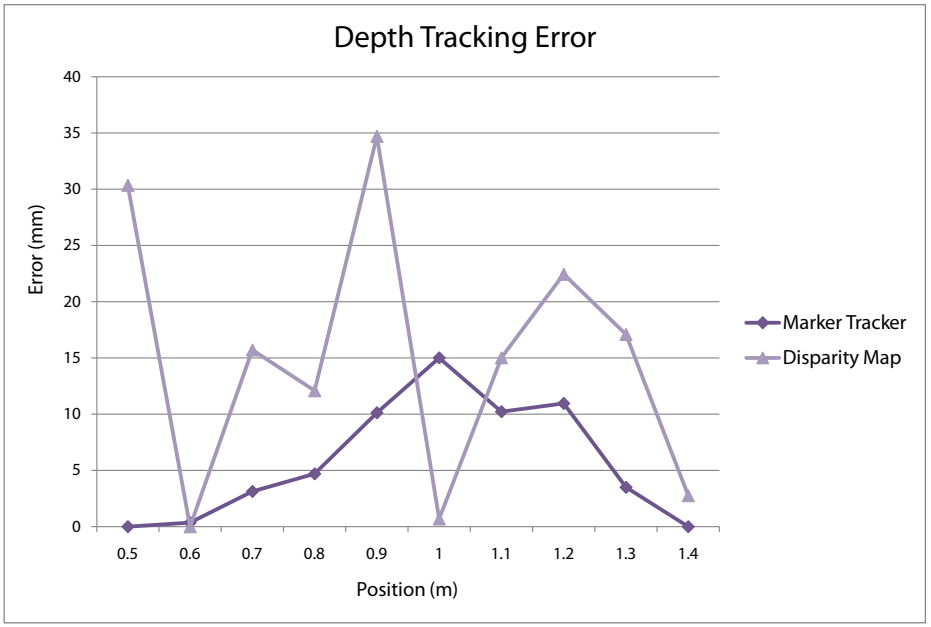


Figure 5.5: Depth error from the marker tracker and disparity map.

5.3.1 Application Design

As stated in chapter 3, the framework was designed so that the program flow would be similar to the existing solutions. This simplifies the learning process for users of other AR development solutions.

The program flow is shown in figure 5.6. The Graphics class (and DirectX) used for display is initialised, and the 3D model used in the application is loaded from file into GPU memory. The camera interface is initialised, before a instance of the Visual Cortex is created. The Visual Cortex contains the marker tracker, which is then initialised.

At this point the main program loop can be entered. In the loop the frame is grabbed from the camera, and processed by the Visual Cortex marker tracker. If a marker is found, the 3D model's transformation matrix is then set up according to the marker pose. The video frame is then written to the back buffer, before the model is rendered onto the video image. The resulting application is shown in figure 5.7.

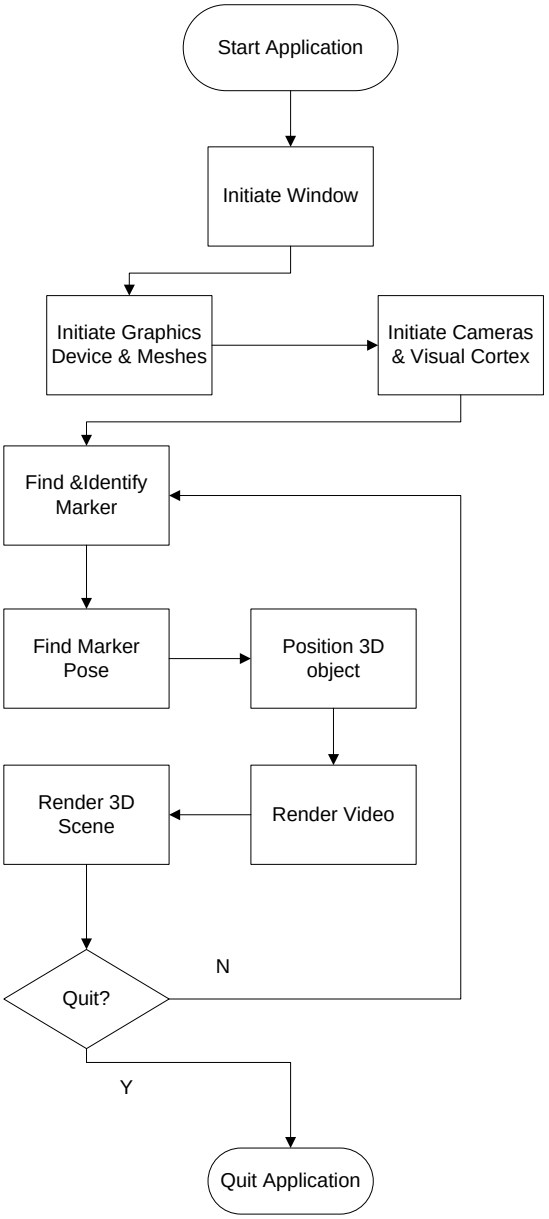


Figure 5.6: AR application flow chart.

5.3.2 Application Comparison

For the framework to be a valid alternative to existing solutions, it has to have comparable performance. The framework is compared to ARToolkit, the most basic solution, and GoblinXNA, the solution that offers the most features. For the comparison, a basic application was built using the three solutions. The application superimposes a cube onto a single marker in a VGA video with a frame rate of 30 Hz. The ARToolkit application was developed in C and uses

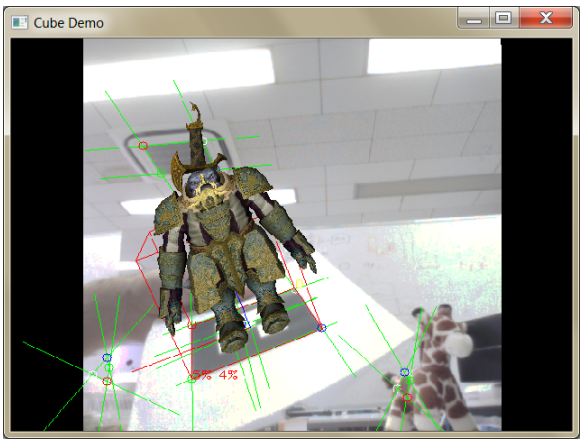
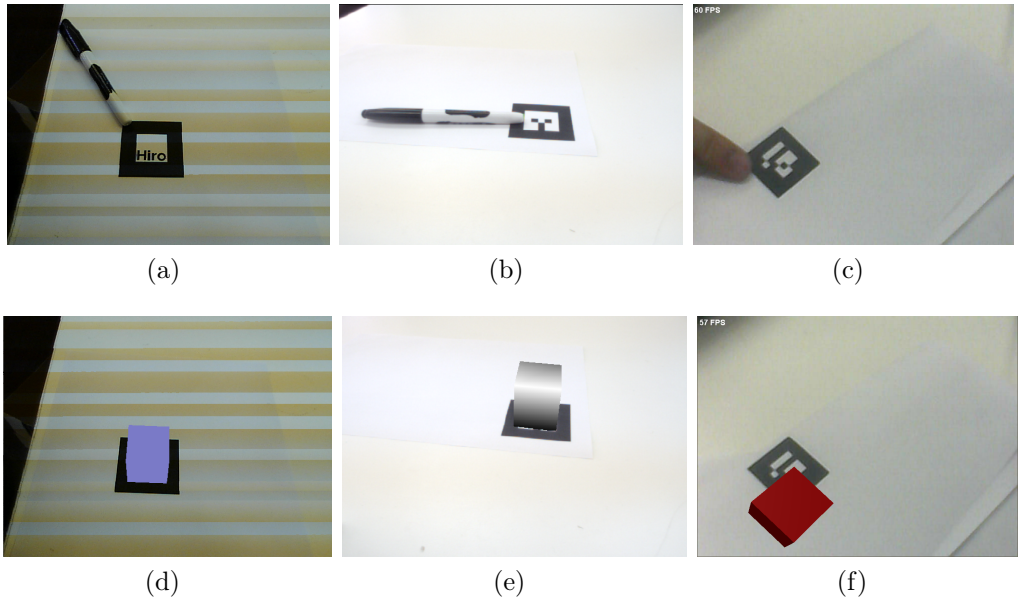


Figure 5.7: An AR application written with the framework, showing marker tracker debug information.



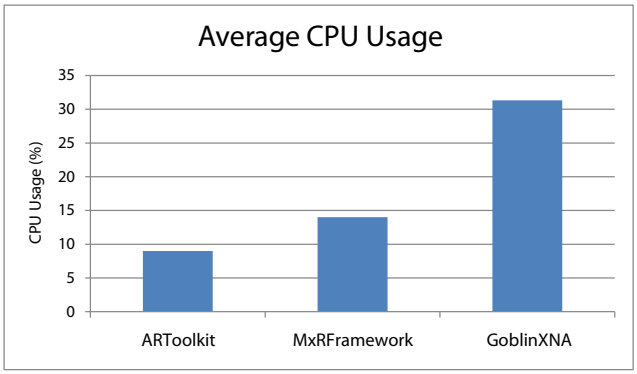
(a) (b) (c)

(d) (e) (f)

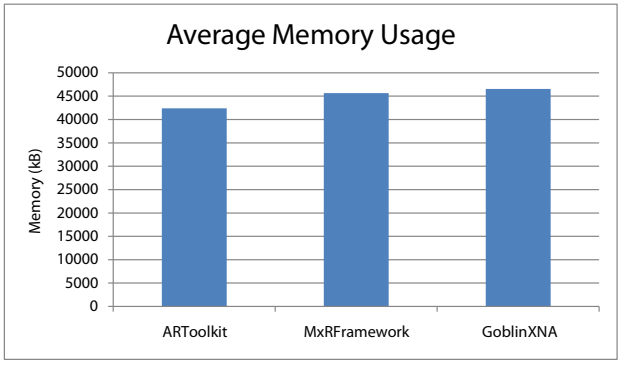
Figure 5.8: Basic Augmented Reality Application Comparison. (a) and (d) ARToolkit. (b) and (e) MxRFramework. (c) and (f) GoblinXNA.

OpenGL for rendering. The GoblinXNA application was developed in C#, and uses DirectX for rendering. The three applications are shown in figure 5.8.

The graphs shown are the average measurements made over a one minute timespan, while the applications were in the main loop unless otherwise stated. The average memory and CPU usage are shown in image 5.10. The CPU processing for ARToolkit and MxRFramework are comparable, while GoblinXNA



(a)



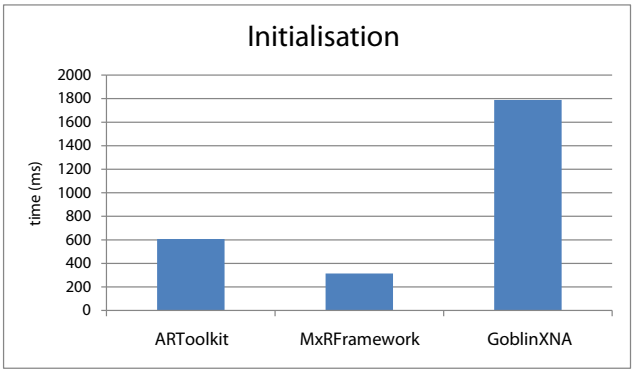
(b)

Figure 5.9: (a) Comparing the average CPU usage and (b) average memory usage of the three AR solutions.

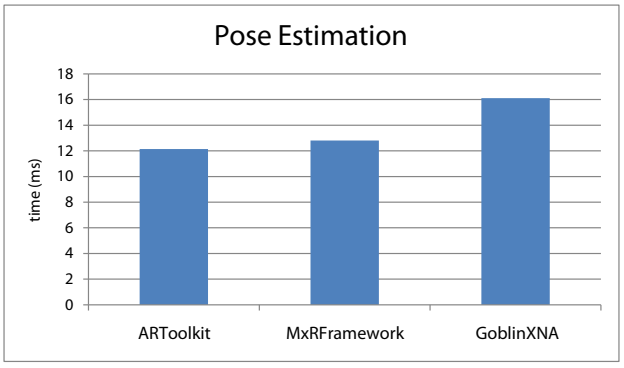
is more processing intensive. The application memory usage for all three cases are similar.

The processing times for the applications are shown in figure 5.10. The initialisation time is the average of 20 samples. The MxRFramework initialises significantly faster than the other solutions, although the initialisation consists of similar processes in all three solutions. In figure 5.10b, the shown pose estimation includes the marker tracking and pose estimation processes. The rendering times includes drawing the video frame and the virtual object. The rendering time for each application differs significantly dependent on the virtual object size and camera movement. For this reason all measurements made were done with stationary markers and cameras.

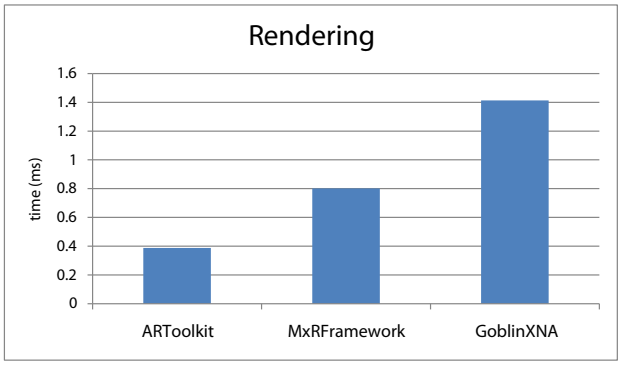
These results show that ARToolkit performs significantly better than GoblinXNA. In the design of GoblinXNA performance was sacrificed to create a



(a)



(b)



(c)

Figure 5.10: AR solutions processing time comparison.

higher level development environment. As illustrated in section 2.2.3.1, GoblinXNA greatly simplifies the creation of intricate virtual scenes for use in AR.

For this project, the results are favourable, as MxRFramework only performs slightly worse than the purpose built ARToolkit, while MxRFramework

offers a more versatile development environment, as illustrated in sections 5.5 and 5.4.

5.4 Implementing a 3D mixed-reality interface

Section 5.3 showed that MxRFramework can be used to create a marker-based augmented reality application comparable to an ARToolkit application. In this section we will illustrate how the expanded abilities of MxRFramework can be used to create a different kind of mixed-reality application. We will discuss the design of an application that makes use of the depth map and 3D display to create a basic 3D peripheral-less interface.

5.4.1 Overview

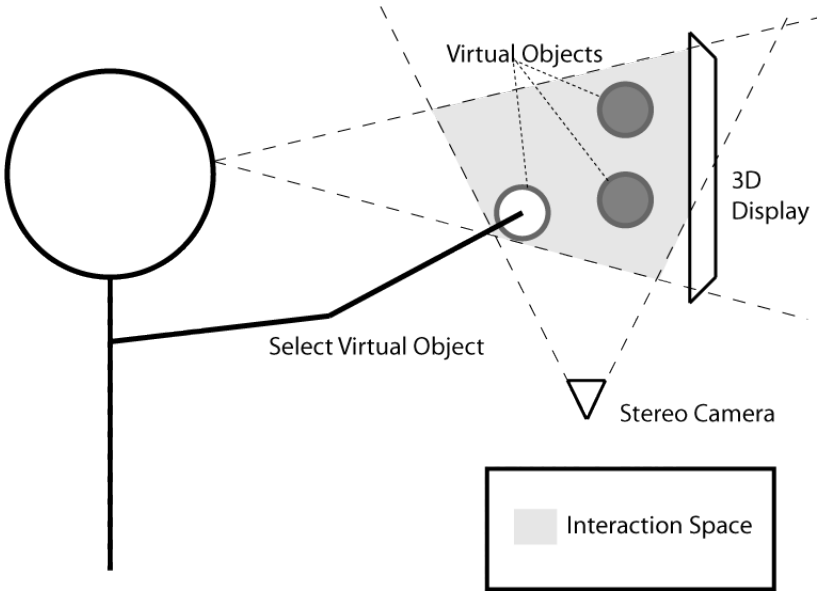


Figure 5.11: 3D Vision Based UI

In this application the "out-of-screen" effect is utilised with a 3D display to create a UI with elements "floating" in front of the screen. The idea is to create a 3D space in front of the monitor in which interaction between real and virtual objects can take place. Interaction would be comparable to a touch screen, but in three dimensions instead of two, and without any tactile

feedback. Figure 5.11 shows the basic setup for the UI. Three main objectives have to be realised in order to create this interactive space:

1. Create the visual illusion of "floating" virtual objects in space outside of the 3D display.
2. Find real world data in the same space, and create virtual representations of this data.
3. Detect collisions between objects in the mixed-reality space.

The first objective can be completed by rendering virtual objects in a 3D scene so that they fall within the virtual camera's visual frustum, nearer than screen depth. This space is dependent on the placement of the virtual camera, but is easily found by inspection. Since 3D displays use an aberrant method for 3D vision, care must be taken to create a believable illusion of floating objects. A few techniques are used to aid the human brain in interpreting the visuals correctly [46].

- Virtual objects should not be clipped.
- Objects should start within the screen, moving out slowly to give eyes time to adapt.
- When rendering the object, opt for realistic (lifelike) rendering[4].

The next step is finding the real world information. Stereo cameras and the dense block matching algorithm are used to find the disparity map of one of the camera planes. The application then monitors the relative positions of known virtual objects on the disparity map for collisions.

5.4.2 Design

All of the computer vision algorithms in the Visual Cortex subclass, are calculated on the CPU. Since the "out-of-screen" effect is utilised in this application, it is imperative that the frame rate is high enough to look natural and avoid eye strain and diplopia.

[4]As opposed to using cartoon shading for example. Create and render 3D objects with lifelike qualities and lighting.

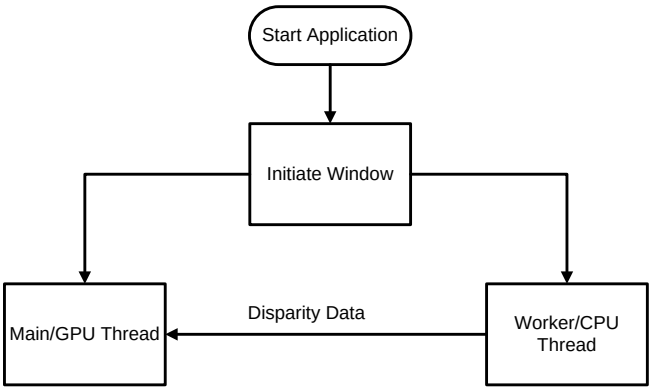


Figure 5.12: MR application flowchart.

This application does not display the video feed. We can separate the computer algorithms calculated on the CPU with the rendering process on the GPU, thereby optimising both processes. This is done by applying a multi-threaded approach, as shown in figure 5.12. The worker thread, with no access to the graphics device, handles the cameras, stereo vision calculations and pixel level collision detection. The main thread is then tasked with creating and rendering the virtual scene at the desired constant frame rate in sync with the refresh rate.

Commands are sent from the worker thread to the main thread when a collision with a UI element is detected. Sections 5.4.2.1 and 5.4.2.2 discuss the design of the two threads in more detail.

5.4.2.1 Worker Thread

As shown in figure 5.12, the worker thread is created and entered as soon as the basic window has been initialised. Figure 5.15 represents the flow of the worker thread.

Firstly, the object instances relating to the CPU are initialised, i.e. two camera instances, and one instance of a Visual Cortex. Since this application uses the disparity map, the program checks for a calibration file. If it is not found, the cameras are calibrated. The capture loop is then entered. The first action taken within the loop is the capturing of video frames. These frames are then remapped to their rectified and undistorted equivalents. The disparity map can then be created from the two frames. The framework was created with simplicity of use in mind. This design philosophy is exemplified in the

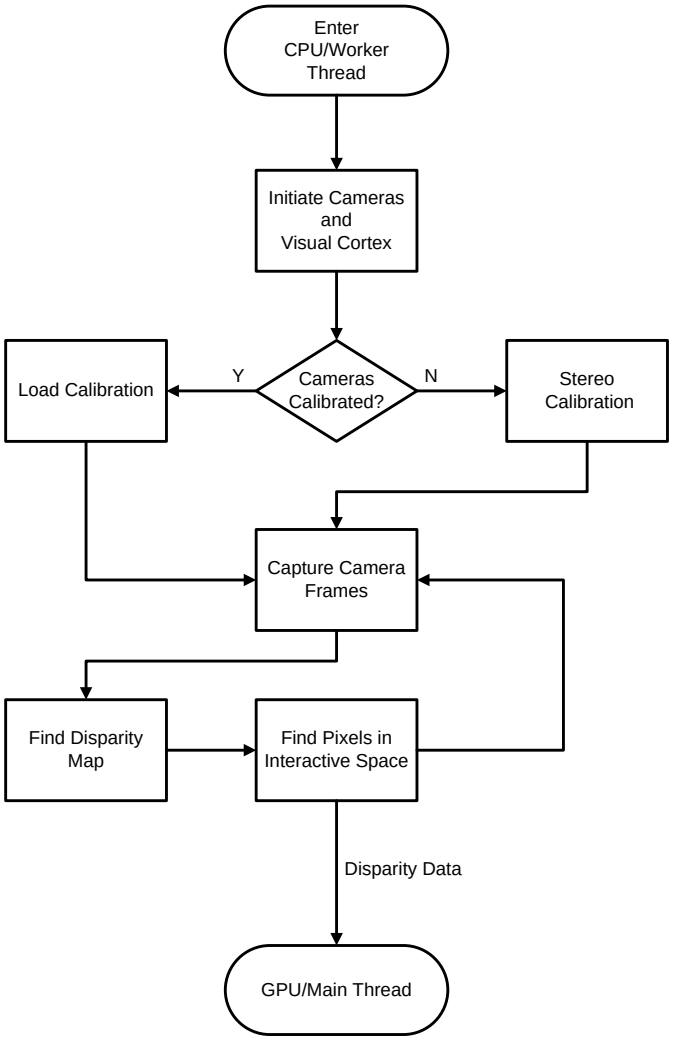


Figure 5.13: Flowchart for the worker thread.

code needed to find the disparity map, shown in listing 5.1.

The thread then monitors the positions of the virtual objects on the disparity map. These virtual spheres, shown in figure 5.14, are used as selectable menu items. If the depth of any pixels from the disparity map falls within the defined depth of the virtual UI spheres a collision test is done. We will refer to the depth in which the menu spheres are found as the menu depth threshold. The test involves the following steps:

1. Extract foreground from input image.

Listing 5.1: Calibrating cameras and finding the disparity map.

```
1
2       cortex->depthMapper->StartCalibration();
3
4       while(calibrate)
5       {
6          ...
7          cortex->depthMapper->AddCalibrationFrame(src);
8          ...
9       }
10      cortex->depthMapper->EndCalibration();
11      while(true)                 //enter main loop
12      {
13         ...
14         cortex->depthMapper->Find(src,&disparity);     //remap find disparity
15         ...
```

Figure 5.14: MR application screen shot.

2. Extract pixels from foreground that fall within the menu depth threshold on the disparity map.

3. Erode and dilate the output from step 2.

4. Find the bounding box.

5. Detect 2D collisions between the bounding box and menu items.

Any occurring collisions are signalled to the main thread. The main thread can then react accordingly before the next draw cycle.

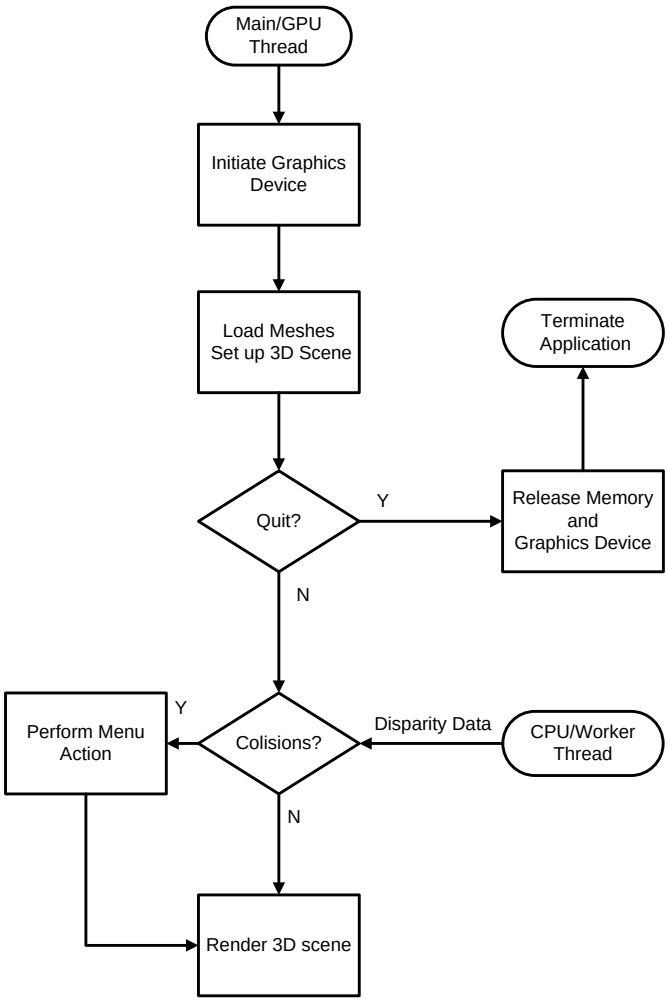


Figure 5.15: Flowchart for the main thread.

5.4.2.2 Main Thread

All interaction with the graphics device happens from the main thread, as shown in figure 5.15. The graphics device and scene are initialised with the DX9Graphics interface and UV-textured meshes are loaded and initialised using the MeshWrapper class in MRUtility. The program then enters the main draw loop, where a check occurs for any commands from the worker thread. Figure 5.14 shows a screen shot from the MR application. The spheres are the selectable objects in the UI, and appear to float outside the scene. The selectable objects all demonstrate tasks that have been simplified within the

framework. The character model and road appear to lie within the screen. These are placed as points of reference in order to exaggerate the floating effect.

5.5 Expanded Augmented Reality Example

Following a similar multi-threaded approach as the design presented in section 5.4, the basic AR application shown in section 5.3 is expanded to include a markerless UI. In this application, a user uses her hand to select virtual menu items placed on a marker. The program detects collisions, but not hand gestures.

Since the design is a combination of the previous two design examples, marker-based augmented reality and multithreaded program approach will not be discussed again. Instead of revisiting program structure, the design section focuses the discussion on code excerpts to demonstrate key aspects of the framework. Please note that the code excerpts demonstrate framework functionality, and that code unrelated to the framework will not be given nor discussed.

5.5.1 Design

The application uses the extractors and depth map to detect a user's hand. If the hand is found, a virtual bounding sphere is created to represent the hand in 3D space. Collisions between the hand and menu items on the marker is then monitored. If the hand intersects a menu item, the menu item reacts. For this example, there are two menu items, the two arrows in figure 5.16. Selecting an arrow will cause the model to rotate in the desired direction.

The marker-based AR scene shown in figure 5.16 has to be constructed. To create an application that could display the scene, the developer would need:

- A live video feed from a camera.
- 3D meshes to build the virtual component of the screen.
- A marker detector.
- A way to display the scene on screen.

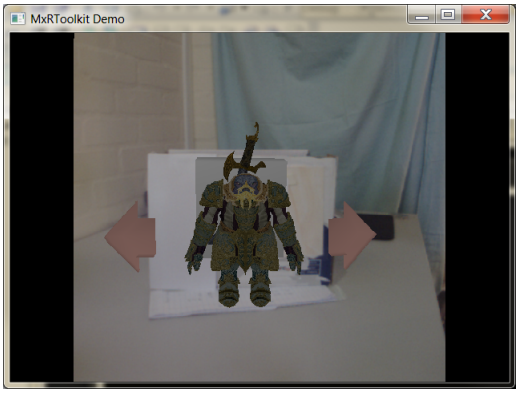


Figure 5.16: AR scene placed on the marker.

To illustrate the usability of the framework, initialisation code is given in listing 5.2. This is the complete code necessary to initialize a basic AR application capable of displaying the scene in figure 5.16.

Listing 5.2: Initializing a basic marker-based AR application.

```
1  //Init Camera and display image
2  eye[0] = new PS3Eye(RESOLUTION,FRAMERATE, RGBA);
3  src[0] = new MRImage(WIDTH,HEIGHT,RGBA);
4  eye[0]->StartCapture();
5
6  //Init tracker with default marker
7  cortex->tracker = new AlvarTrackerTracker(WIDTH,HEIGHT);
8  cortex->tracker->Init(markerID);
9
10 //Init Graphics
11 graphics = new DX9Graphics();
12 graphics->InitWindow(...);
13 graphics->Init2DVideo();
14
15 //Load meshes from file
16 dwarfMesh.Load( "dwarf.x" );
17 if ( dwarfInstance )
18 {
19   dwarfInstance[0].Release();
20 }
21 // create 1 instance
22 dwarfInstance = new MRMeshInstance[1];
23 dwarfInstance[0].SetMesh( &dwarfMesh );
```

Similarly, the run-time framework calls, shown in listing 5.3, are uncomplicated. The camera frame is grabbed and stored in *src[0]*. This image is passed to the tracker for marker detection. If the marker is found, the dwarf mesh's transformation matrix is adjusted accordingly. The video and dwarf mesh are then rendered and displayed on screen.

Listing 5.3: Main-loop code for a basic marker-based AR application.

```
1  eye[0]->GetFrame(src[0]);
2  markerFound = cortex->tracker->GetPose(src[0],poseMatrix);
3
4  if(markerFound)
5    dwarfInstance[0].SetTransform(poseMatrix);
6  dwarfInstance[0].Translate(SOME_OFFSET);
7
8  graphics->RenderSceneStart();
9  graphics->Display2DImage(src[0]);
10 if(markerFound)
11   dwarfInstance[0].Render();
12 graphics->RenderSceneEnd();
```

We will now discuss how the application is expanded to incorporate markerless interaction. The application should react if a user touches one of the arrows with her hand. To detect these collisions between the hand and the arrows, the application should:

- Use the colour extractor to find hand coloured pixels.
- Remove any unwanted shadows and noise pixels.
- Find a 2D virtual representation of the hand.
- Find the depth of the hand relative to the cameras using a disparity map.
- Find a 3D virtual representation of the hand.

These steps can be accomplished with a combination of extractors and a depth mapper. The application uses a combination of two image extractors: the colour extractor to find hand pixels and the shadow remover. 2D and 3D body extractors are used to find 2D and 3D representations of the object. The application uses two cameras to create the disparity map.

Listing 5.4 gives the code relating the extractors. Please note that the extractor threshold and constants have to be found and the stereo rig needs to be calibrated.

The method $MathMorph()$ enables opening as discussed in 4.2.3. The extractor is set up with default kernels. The kernel shapes and sizes can be changed if needed.

When setting up the controller, the order in which the extractors are added is important. The controller will finish processing all extractors on a level before advancing to the next, but the order in which the extractors on the

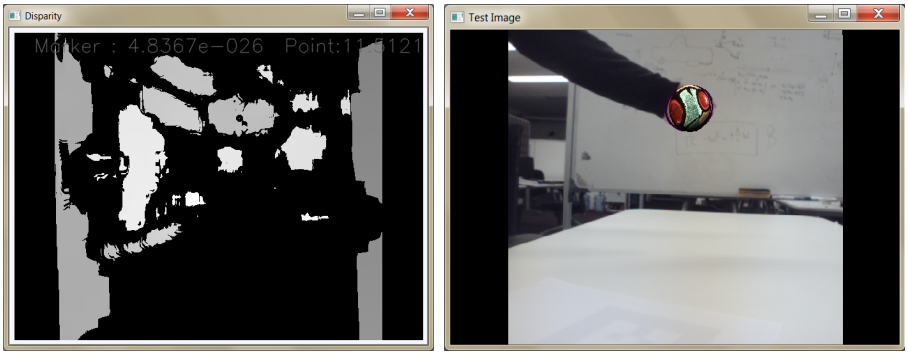
same level are processed is determined by the order in which they are added. In this example it means that *colourExtractor* and *shadowSuppressor* will be processed in the order they are added, but they will always be followed by *ellipseExtractor* and then *sphereExtractor*. In this example we are only interested in the highest level controller output: a bounding sphere. Lower level output (*imageLevel* or *body2DLevel*) could also be accessed if needed.

Listing 5.4: Extractor and depth map code example.

```
1  /////  initialisation TIME
2  ImageExtractor* colourExtractor = new ColourExtractor();
3  ImageExtractor* shadowSuppressor = new ShadowSuppressor();
4  Body2DExtractor* ellipseExtractor = new BoundingEllipsiodExtractor();
5  Body3DExtractor* sphereExtractor = new BoundingSphereExtractor();
6  cortex->extractorController->Init();
7  cortex->depthMapper->Init();
8
9  //Initialize, calibrate and set thresholds
10 .
11 .
12 .
13 ///
14 ellipseExtractor->AddDisparityMap(cortex->depthMapper);
15 //Set up controller
16 //imageLevel
17 cortex->extractorController->Add(colourExtractor);
18 cortex->extractorController->Add(shadowSuppressor);
19 //body2DLevel
20 cortex->extractorController->Add(ellipseExtractor);
21 //body3DLevel3
22 cortex->extractorController->Add(sphereExtractor);
23 cortex->extractorController->MathMorph(true);
24
25 //IN PROGRAM LOOP
26 cortex->depthMapper->Find();
27 cortex->extractorController->Process(&3DSphere);
```

In figure 5.17 the 3D sphere found by the extractor controller is visualised with a rendered sphere. At the moment the sphere's size is constant.

The extractor controller outputs the hand sphere in the same coordinate system as the virtual UI arrows. The program monitors collisions between the hand sphere and the UI arrows. When the user selects an arrow, the dwarf mesh rotates in the selected direction. A screenshot from the completed application is shown in figure 5.18, and videos of the application is included in appendix E.



(a) (b)

Figure 5.17: Finding the bounding sphere using the disparity map to find depth. (b) The bounding sphere is illustrated with a rendered sphere. (a) The depth of the bounding sphere is found from the disparity map.

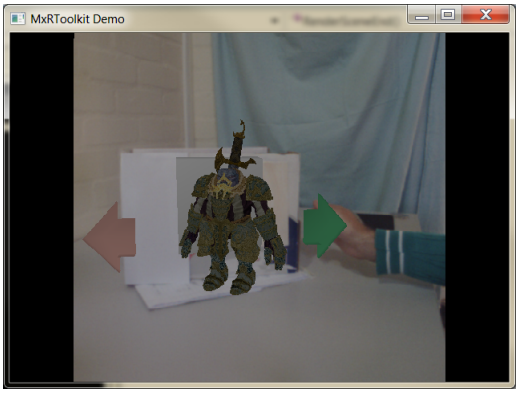


Figure 5.18: Markerless AR user interaction.

5.5.2 Results

The framework was successfully used to create a marker-based application, and could also be used to create expanded mixed reality applications. We have shown that the framework design gives the developer freedom. The computer vision algorithms were also shown to be fast and accurate enough to be used in a real-time application. The structure of the extractors and extractor controller seemed to be especially valuable during development. The framework effectively abstracts its various elements, and compared to existing solutions, offers a wider scope of possible applications. Unfortunately the image extractors and the 3d sphere extractor needs to be calibrated before use, and this still proved to be time consuming as the framework does not automate

or simplify this process.

5.6 Implementation Issues

In this section we will look at issues that had an impact on framework development and implementation.

5.6.1 3D Video

The method of moving the video frames data from CPU local RAM to memory local to the graphics device is slow. Displaying two 640×480 images in this way causes the framerate to slow to under ten frames per second. The problem is that the method found in [4] for displaying pre-rendered stereo content assumes that the the images can be loaded at initialisation time as the process is not ideal for real-time application. The stretchRect method used to place the stereo image surface in the back buffer is not part of the standard DirectX pipeline, and causes a short pause in processes on the GPU while it waits for a CPU clock cycle [84]. The strechRect method needed for this process does not allow transferring surfaces from the default memory to non-render target surfaces in any other memory pool than default memory [46].

5.6.2 3D Augmented Reality

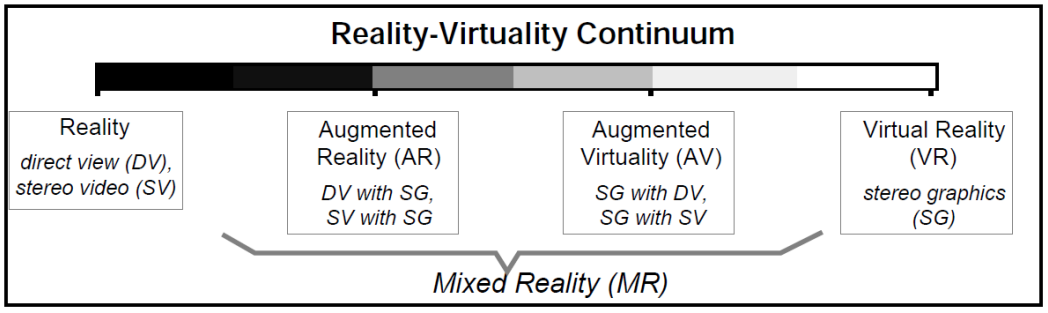


Figure 5.19: Perception along the Reality-Virtuality Continuum. Figure from [5].

The project includes 3D video support for the purpose of making 3D augmented reality. Perceiving a virtual object's position and size correctly in a

2D augmented reality scene can be problematic [5]. One method to remove the ambiguity is to cast shadows onto real world surfaces, as done by [28] and [31]. This approach is sufficient if the object is close to the surface.

To remove the ambiguity completely, the framework is capable of rendering the virtual part of the augmented scene for 3D display, superimposing the scene on the corresponding 3D video from two cameras. To accurately display the video the projection of the virtual camera has to include the physical camera intrinsics $\mathbf{M}$. The second virtual camera's projection cannot include the second real camera's intrinsics, as the virtual camera is duplicated automatically in the rendering process. The virtual cameras have to be aligned according to the real world cameras by editing the parallax (vergence) and inter-ocular distance. This process proved to be highly volatile. The maximum parallax of the virtual cameras would typically not be enough to fit the the convergence of the physical rig, with the rig set so that the 3D video could be viewed comfortably. The physical rig also needs to be near perfectly parallel, as the virtual cameras are in the same Y-plane. As mentioned, the display only creates the illusion of 3D sight. Slight differences between the real world and the virtual world cameras would cause diplopia due to the brain not interpreting the scene as 3D.

A more stable approach would be to render the virtual objects to each of the two video frames separately, and then display the two frames in stereo as described in section 4.4. The position of the virtual object is calculated with the marker tracker, and inserted into each scene accordingly. As a test, the pose was calculated, and 3D content was rendered to the marker position using the CPU before integrating it with the image. This was done for both images before the images were displayed using the graphics device. With this setup, the scene displayed correctly, even when the cameras move (or even move relative to each other). However, this cannot be performed in real-time.

We were unable to reproduce this process using the GPU. As the stereoscopic driver automatically renders all virtual objects stereoscopically, the idea was to create mono off-screen render targets, render the two augmented reality scenes to the surfaces and then combine them as 3D video. The process described by NVidia in [46] to create mono render targets while the stereo driver is active proved unsuccessful. Nvidia would make implicit stereo rendering possible with the driver [4], at the time of writing this has not realised. The only possible solution found was to use a professional Quadro graphics device

with quad buffering capabilities [85], however this has not yet been tested.

Chapter 6

Conclusions and Recommendations

6.1 Summary

To truly be a test bed for new ideas, MxRFramework had to have an expandable approach to computer vision. Marker tracking was included as it offers a known trusted reference. Support for single image processing techniques was added with the addition of image and 2D body extractors. A interface to a disparity map was added to expand the use of the image and 2D body extractors to 3D space. Extractors and the extractor controller gives the developer a way to include a variety of algorithms in the framework, and experiment with them in mixed reality environments.

Fast, basic algorithms were implemented for the extractors and depth mapper, and a wrapper to an existing marker tracker library was included. We showed that with this combination of tools, various interesting applications could be built, while the performance of MxRFramework was comparable to other solutions. In the following section, we will show that our initial project goals were met during the research, design and implementation of MxRFramework

6.1.1 Meeting Project Goals

This project had three main objectives. The first was the theoretical study of augmented reality, and a review of development solutions. In chapter 2 this

objective was met, as it gave an overview of available AR design solutions.

The development of a platform for the rapid design of augmented reality applications was the first goal that arose out of this study. The framework was designed as discussed in 3, and augmented reality applications built with the framework were presented: The second goal, to develop a platform for the rapid design of augmented reality applications, was realised.

Chapters 3 and 4 discussed the addition of tools to interpret the physical world, and chapter 5 presented two applications with markerless interaction. These tools can be used separately or together depending on the application. The final objective, expanding the framework capabilities beyond marker based augmented reality was met. Additionally, section 5.4 presented a 3D, mixed reality interface, exemplifying that the framework can be used to create a wide variety of mixed reality applications. The project was successful, providing a platform for further research and development and proving the concept of markerless interaction in augmented reality.

6.2 Further Research

The framework contains extractors focussing on minimising processing time. These algorithms were chosen so that multiple extractors could be used in the same application, while satisfying the augmented reality condition that the application should run in real time. We experimented with the inclusion of more intricate image extractors, but the processing times quickly increased to unacceptable levels. The framework in it's current state offers the possibility of various new applications, but to truly reach its potential, a GPU-based Visual Cortex is needed. This will allow the inclusion of more intricate image and 2D object extraction methods. Ideally, a dedicated GPU could be used for computer vision with a different graphics device used for rendering.

As explained in chapter 5.6, the possibilities of 3D augmented reality with 3D video could also be explored further with a Quadro graphics device.

The implemented 3D object extractor creates a virtual sphere to represent the real object extracted with the image extractors. The depth of the sphere is then found by finding the bounding shape centre point on the disparity map. A sphere was chosen because collision detection is easily done with spheres, while the concept is still proven. A 2D object extractor could be implemented

to better approximate the real world object, and with this information a better virtual 3D object could be created. The disparity map is under utilised by the current extractors.

During the timespan of this project, public interest in 3D displays and stereoscopic computer vision has increased significantly. 3D television has become commercially available [88]. Another significant development was the release of the Microsoft Kinect [89]. The Kinect has successfully been used in a number of projects [90], and it could be a viable replacement for the current stereo camera system. This growing interest and advances suggests merit in the initial idea to merge these technologies, and provides growing new options for further research.

Bibliography

[1] P. Milgram and F. Kishino, "A Taxonomy of Mixed Reality Visual Displays," *IEICE Transactions on Information Systems*, vol. E77-D, pp. 1321–1329, 1994.

[2] G. Bradski and A. Kaehler, *Learning OpenCV*, 1st ed., M. Loukides, Ed. OReilly Media, September 2008.

[3] P. Webb, Ed., *Bioastronautics Data Book*. NASA: Washington D. C., 1964.

[4] S. Gateau, "The In and Out: Making Games Play Right with Stereoscopic 3D Technologies," 2009, nvidia.

[5] D. Drascic and P. Milgram, "Perceptual issues in augmented reality," in *Proceedings, Stereoscopic Displays and Virtual Reality Systems III,*. SPIE, February 1996, pp. 123–134.

[6] "Artag homepage." [Online]. Available: http://www.artag.net/

[7] M. Fiala, "Artag, an improved marker system based on artoolkit," National Research Council of Canada, Tech. Rep., 2004.

[8] "Artoolkit homepage." [Online]. Available: http://www.hitl.washington.edu/artoolkit/

[9] R. Raskar, Ed., *Spacial Augmented Reality: Merhing Reale and Virtual Worlds*.

[10] R. T. Azuma, "A survey of augmented reality," *Teleoperators and Virtual Environments*, vol. 6, pp. 355 – 385, 1997.

[11] M. Billinghurst, H. Kato, I. Poupyrev, H. Regenbrecht, D. S. Tan, and N. Tetsutani, "Developing a generic augmented-reality interface," *IEEE Computer Graphics (SIGGRAPH 84)*, vol. 35 No. 3, pp. 44–50, 2002.

[12] S. Cawood and M. Fiala, *Augmented Reality: A Practical Guide*. Pragmatic Programmers, LLC, 2007.

[13] H. Park, "Invisible marker tracking for ar," in *Third IEEE and ACM International Symposium on Mixed and Augmented Reality*, 2004, pp. 72–273.

[14] F. Chaumette, A. Comport, and E. Marchand, "A real-time tracker for markerless augmented reality, mixed and augmented reality," in *Proceedings, International Symposium on Mixed and Augmented Reality, ACM*, 2003.

[15] M. Armstrong and A. Zisserman., "Robust object tracking," in *Asian Conference on Computer Vision*, 1995.

[16] A. Davison, "Real-time simultaneous localisation and mapping with a single camera," in *International Conference of Computer Vision*, 2003.

[17] A. Behzadan, V. Kamat, and B. Timm, "General-purpose modular hardware and software framework for mobile outdoor augmented reality applications in engineering," *Adv. Eng. Inform.*, vol. 22 no. 1, pp. 90–105, 2008.

[18] O. Oda, L. Lister, S. White, and S. Feiner, "Developing an augmented reality racing game," in *INTETAIN '08*, January 2008.

[19] G. Klein, "Visual tracking for augmented reality," Ph.D. dissertation, University of Cambridge, 2006.

[20] "Hitlab, washington, artoolkit computer vision algorithm,." [Online]. Available: www.hitl.washington.edu/artoolkit/documentation/vision.htm

[21] M. Billinghurst, R. Grasset, J. Loosener, and H. Seichter, "Osgart a pragmatic approach to mr," hitlab NZ.

[22] "Buildar, hitlab nz." [Online]. Available: www.hitlabnz.org/wiki/BuildAR

[23] "Metaio homepage." [Online]. Available: http://www.metaio.com/products/

[24] "Mars homepage." [Online]. Available: https://gna.org/projects/mars/

[25] B. MacIntyre, M. Gandy, S. Dow, and J. Bolter, "Dart: A toolkit for rapid design exploration of augmented reality experiences," College of Computing, Interactive Media Technology Center, School of Literature, Communication and Culture, GVU Center, Georgia Institute of Technology, Atlanta, USA.

[26] "Macromedia director, macromedia inc,." [Online]. Available: www.macromedia.com/director

[27] S. Guven, "Authoring 3d hypermedia for wearable augmented and virtual reality," in *7th IEEE International Symposium on Wearable Computers*, 2003.

[28] "Osgart homepage." [Online]. Available: http://www.artoolworks.com/osgART_-_Home.html

[29] "Opensg homepage." [Online]. Available: http://www.opensg.org/

[30] K. Hawkins and D. Astle, *OpenGL Game Programming*, A. LaMothe, Ed. Course Technology PTR, 2002.

[31] "Goblinxna homepage." [Online]. Available: http://graphics.cs.columbia.edu/projects/goblin/index.htm

[32] "Newton game dynamics home page." [Online]. Available: http://newtondynamics.com/

[33] O. Oda and S. Feiner, *Goblin XNA User Manual Version 3.3*, 2009.

[34] F. D. Luna, *Introduction to 3D Game Programming with DirectX9.0*, R. Lopez, Ed. Worldware Publishing Inc., 2003.

[35] R. A. Thomson, "Direct3d graphics pipeline," preliminary Draft.

[36] W. Jones, *Beginning DirectX 9.* Premier Press, 2004.

[37] F. D. Luna, *Introduction to 3DGame Programming with DirectX- 9.0c:A Shader Approach*, R. Lopez, Ed. Jones & Bartlett Learning, 2010.

[38] "Msdn library, direct3d 9." [Online]. Available: http://msdn.microsoft.com/en-us/library/bb219837(VS.85).aspx

[39] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2003.

[40] J. Fryer and D. D.C. Brown, "Lens distortion for close-range photogrammetry," *Photogrammetric Engineering and Remote Sensing*, vol. 52, pp. 51–58, 1986.

[41] D. Brown, "Decentering distortion of lenses," *Photogrammetric Engineering*, vol. 32(3), pp. 444–462, 1966.

[42] Z. Zang, "A flexible new technique for camera calibration," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, pp. 1330–1334, 2000.

[43] D. C. Brown, "Close-range camera calibration," *Photogrammetric Engineering*, vol. 37, pp. 855–866, 1971.

[44] A. Bhatti, *Stereo Vision*. InTech, 2008.

[45] J.-Y. Bouguet, "Camera calibration toolbox for matlab." [Online]. Available: http://www.vision.caltech.edu/bouguetj/calib_doc/htmls/ref.html

[46] S. Gateau, "3d vision technology: Develop, design, play in 3d stereo," 2009, nvidia.

[47] "Maya homepage." [Online]. Available: http://usa.autodesk.com/maya/

[48] "Blender homepage." [Online]. Available: http://www.blender.org/

[49] "Unity homepage." [Online]. Available: http://unity3d.com/

[50] L. Prechelt, "An empirical comparison of c, c++, java, perl, python, rexx, and tcl," *IEEE Computer*, vol. 33, pp. 23–29, Oct 2000.

[51] "http://www.nvidia.com/object/3d-vision-requirements.html," nvidia 3D vision compatible devices.

[52] "Cimg library homepage." [Online]. Available: http://cimg.sourceforge.net/reference/group__cimg__faq.html

[53] "Vxl homepage." [Online]. Available: http://vxl.sourceforge.net/

[54] "Integrating vision toolkit homepage." [Online]. Available: http://ivt.sourceforge.net/

[55] S. C. Entertainment, "Playstation eye brings next-generation communication to playstation3," April 2007. [Online]. Available: us.playstation.com

[56] T. Speech, "Playstation eye - q+a," April 2007. [Online]. Available: ThreeSpeech.com

[57] J. Adams, *Programming Role Playing Games with DirectX*. Premier Press, 2005.

[58] "High-performance counter support." [Online]. Available: http://msdn.microsoft.com/en-us/library/ms901807.aspx

[59] T. Porter and T. Duff, "Compositing digital images," *Computer Graphics (SIGGRAPH 84)*, vol. 18(3), pp. 253–259, 1984.

[60] A. Frery, J. Gomes, and L. Velho, *Image Processing for Computer Graphics and Vision*, D. Gries and F. Schneider, Eds. Springer Science + Business Media, 2009.

[61] Y.-Y. Chaung, B. Curles, D. H. Salesin, and R. Szeliski, "A bayesian approach to digital matting," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2001.

[62] R. Szeliski, "Computer vision: Algorithms and applications," 2010, preliminary Draft.

[63] K. Toyama, J. Krumm, B. Brumitt, and B. Meyers, "Wallflower: Principles and practice of background maintenance," in *International Conference on Computer Vision*, 1999.

[64] Y. Benezeth, P. Jodoin, B. Emile, H. Laurent, and C. Rosenberger, "Review and evaluation of commonly-implemented background subtraction algorithms," in *ICPR 2008. 19th International Conference on Pattern Recognition*, Dec 2008.

[65] S. Cheung and C. Kamath, "Robust techniques for background subtraction in urban traffic video," *Visual Communications and Image Processing*, vol. 5308, pp. 881 – 892, 2004.

[66] P. PEER, J. KOVAC, and F. SOLINA, "Human skin colour clustering for face detection," in *EUROCON*, 2003.

[67] M. FLECK, D. FORSYTH, and C. BREGLER, "Finding naked people," in *ECCV*, vol. 2, 2002, pp. 592 – 602.

[68] V. Vezhnevets, V. Sazonov, and A. Andreeva, "A survey on pixel-based skin color detection techniques," in *GRAPHICON*, 2003.

[69] B. D. Zarit, B. Super, and F. Quek, "Comparison of five color models in skin pixel classification," in *ICCV*, 1999.

[70] W. Skarbek and A. Koschan, "Colour image segmentation: a survey," Technical University of Berlin, Tech. Rep., 1994.

[71] A. X. Li, "Researcher in human-computer interaction, university of huddersfield." [Online]. Available: http://www.andol.info/research

[72] A. Prati, M. T. I. Mikic, and R. Cucchiara., "Detecting moving shadows: Algorithms and evaluation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, pp. 918–923, 2003.

[73] R. Cucchiara, C. Grana, M. Piccardi, A. Prati, and S. Sirotti, "Improving shadow suppression in moving object detection with hsv color information," in *IEEE International Conference of Intelligent Transportation Systems*, 2001.

[74] A. Doshi and M. Trivedi, "Hybrid cone-cylinder codebook model for foreground detection with shadow and highlight suppression," in *IEEE International Conference on Advanced Video and Signal Based Surveillance*, 2006.

[75] R. Cucchiara, C. Grana, G. Neri, M. Piccardi, and A. Prati, "The sakbot system for moving object detection and tracking," *Video-Based Surveillance Systems-Computer Vision and Distributed Processing*, vol. August, pp. 145–157, 2001.

[76] G. Matheron and J. Serra, "The birth of mathematical morphology," presented at the international symposium on mathematical morphology, June 2000.

[77] J. Serra and P. Soille, "Mathematical morphology and its applications to image processing," in *proceedings of the 2nd international symposium on mathematical morphology*, 1998.

[78] L. Najman and H. Talbot, *Mathematical morphology: from theory to applications*. ISTE-Wiley, 2010.

[79] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8, pp. 679–714, 1986.

[80] D. Scharstein and R. Szeliski, "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms," *International Journal of Computer Vision*, vol. 47(1/2/3):, pp. 7–42, 2002.

[81] K. Konolige, "Small vision system: Hardware and implementation," in *Proceedings of the International Symposium on Robotics Research*, 1997, pp. 111–116.

[82] G. Sotak and K. Boyer, "The laplacian-of-gaussian kernel: A formal analysis and design procedure for fast, accurate convolution and full-frame output," *Computer Vision, Graphics, and Image Processing*, vol. 48, pp. 147–189, 1989.

[83] "Opencv homepage." [Online]. Available: http://opencv.willowgarage.com/wiki/

[84] W. Bahnassi, "Copying managed texture from system memory to real video memory, questions answered by microsoft directx mvp lead programmer." [Online]. Available: http://us.generation-nt.com/

[85] "Meant to be seen homepage." [Online]. Available: http://www.mtbs3d.com/

[86] S. Brutzer, B. Hoferlin, and G. Heidemann, "Evaluation of background subtraction techniques for video surveillance," *Computer Vision and Pattern Recognition*, vol. S, pp. 1937 – 1944, 2011.

[87] "Alvar homepage." [Online]. Available: http://virtual.vtt.fi/virtual/proj2/multimedia/alvar.html

[88] T. N. Group, "Media fact sheet," 2011.

[89] "Kinect homepage." [Online]. Available: http://www.xbox.com/en-US/kinect

[90] "Creative applications network." [Online]. Available: http://www.creativeapplications.net/kinect/

Appendices

Appendix A

Comparison between AR Solutions

A.1 ARTag and ARToolkit

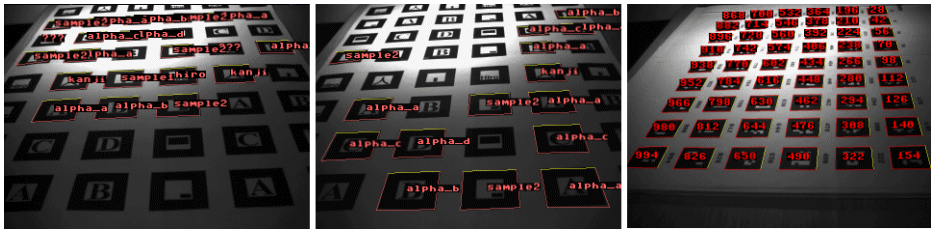
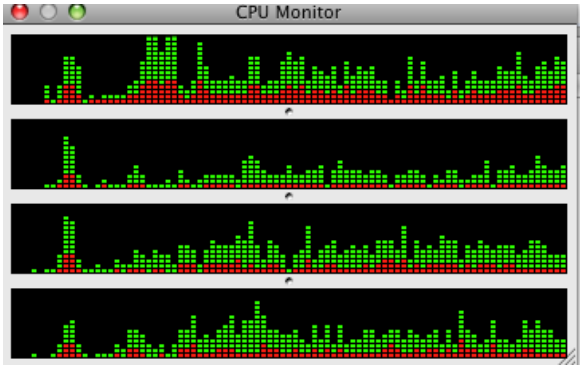


Figure A.1: Marker tracker comparison in varying light conditions: ARToolkit performance with low and high threshold values are shown in the left and middle images. The right image shows ARTag performance under the same conditions. Image courtesy of [6].

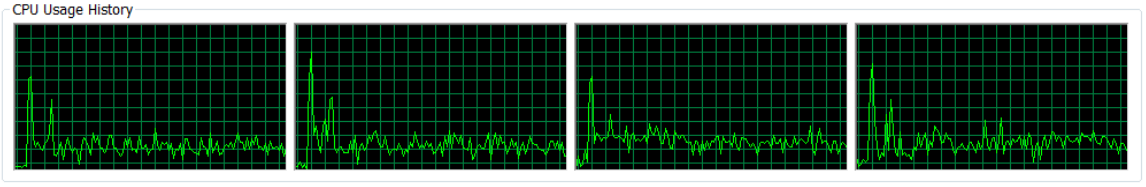
Table A.1: ARTag and ARToolkit processing times using a Pentium 4 3.0 GHz PC. Table reproduced from [7].

		ARToolkit				**ARTag**			
Camera		**Visible Markers**				**Visible Markers**			
Type	**Resolution**	0	24	32	48	0	24	32	48
Intel CS120	320 × 240	3 ms	17 ms	22 ms	22 ms	2 ms	7 ms	8 ms	9 ms
IEEE 1394 Dragon-fly	640 × 240	0–15 ms	0–15 ms	15–31 ms	15–31 ms	0–15 ms	0–15 ms	0–15 ms	0–15 ms

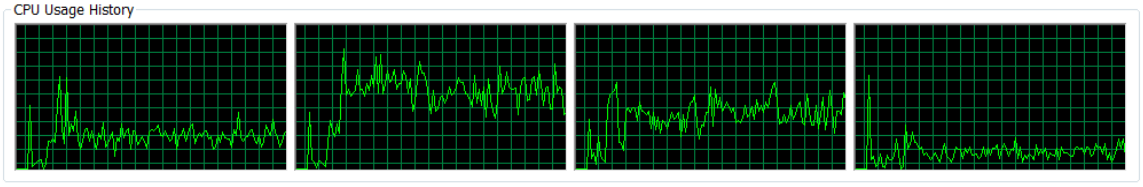
A.2 CPU Usage for ARToolkit, GoblinXNA and MxRFramework



(a)



(b)



(c)

Figure A.2: Comparing Depth from the marker tracker and the disparity map. Initial peaks are at initiation time. (a) ARToolkit. (b) MxRFramework. (c) GoblinXNA.

Appendix B

Development Computer Specifications

PC Specifications	
CPU	Inter Core 2 Quad CPU Q9300 2.5 GHz
Operating System	System Windows 7 64 bit
Memory	8 GB DDR2 RAM
Memory Speeds	833 MHz
Graphics Card Specifications	
Graphics Card	ASUS GTS 250 1GB
Connection	PCI-E
Core Speed	740 MHz
Memory	1 GB DDR3
Memory Speed	2000 MHz
Shader Clock Speed	2000 MHz
Memory Bus	740 MHz

Appendix C

Artificial Background Subtraction Test Data

C.1 Dataset

C.1.1 Background Reference Image

Figure C.1: Background image.

C.1.2 Test Images

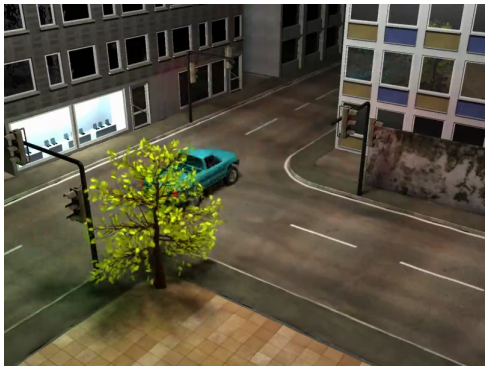
(1)

(2)

(3)

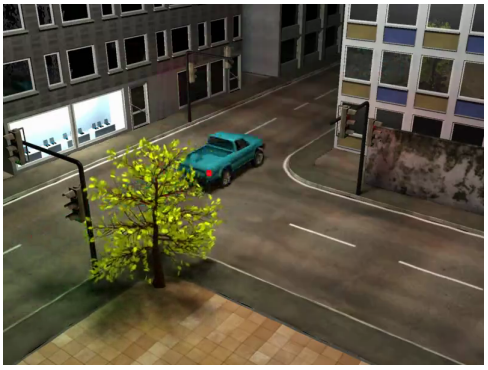
(4)

(5)



(6)

Figure C.2: Test images.



(7)

(8)

(9)

(10)

(11)

(12)

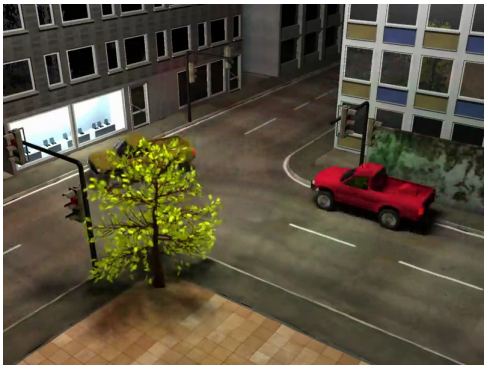
(13)

(14)

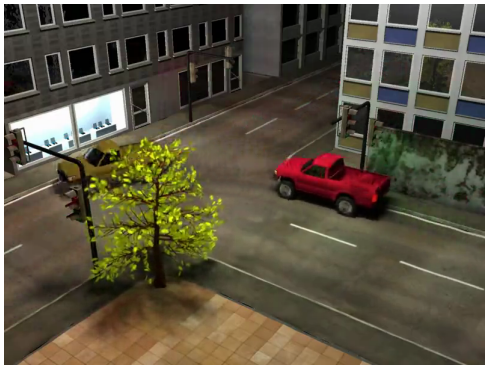
Figure C.3: Test images (continued).

(15)

(16)

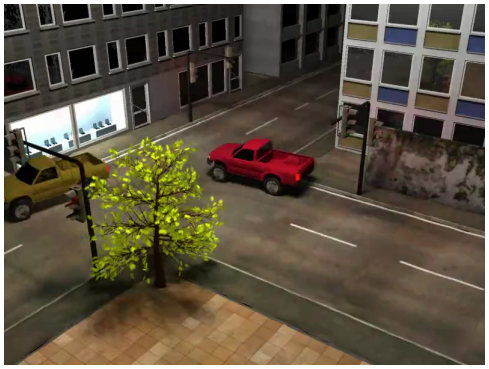


(17)



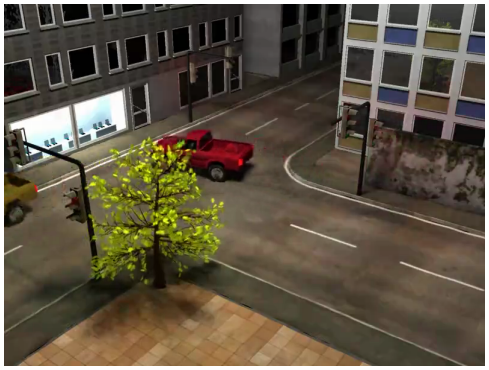
(18)

(19)



(20)

(21)

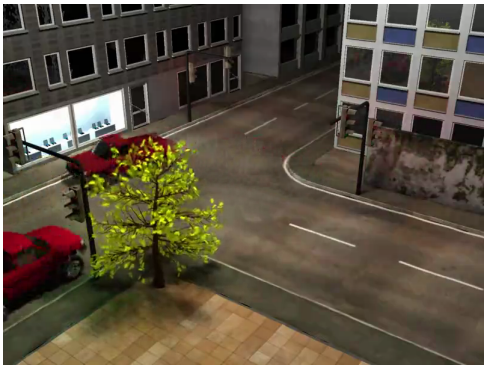


(22)

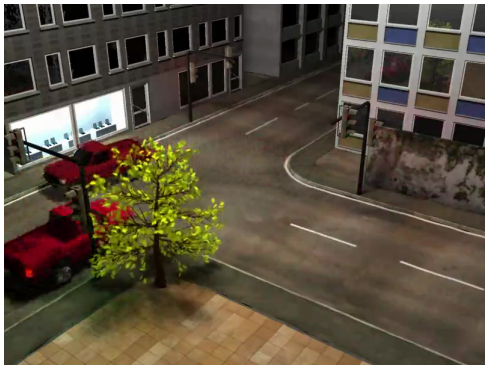
Figure C.4: Test images (continued).

(23)

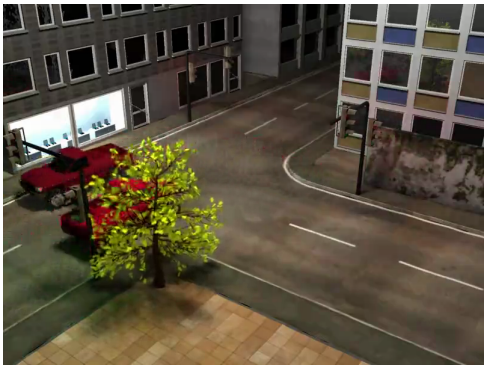
(24)



(25)

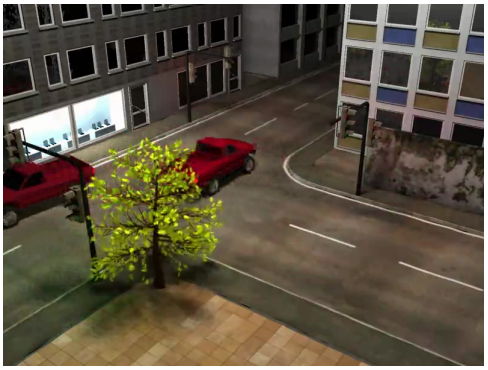


(26)



(27)

(28)

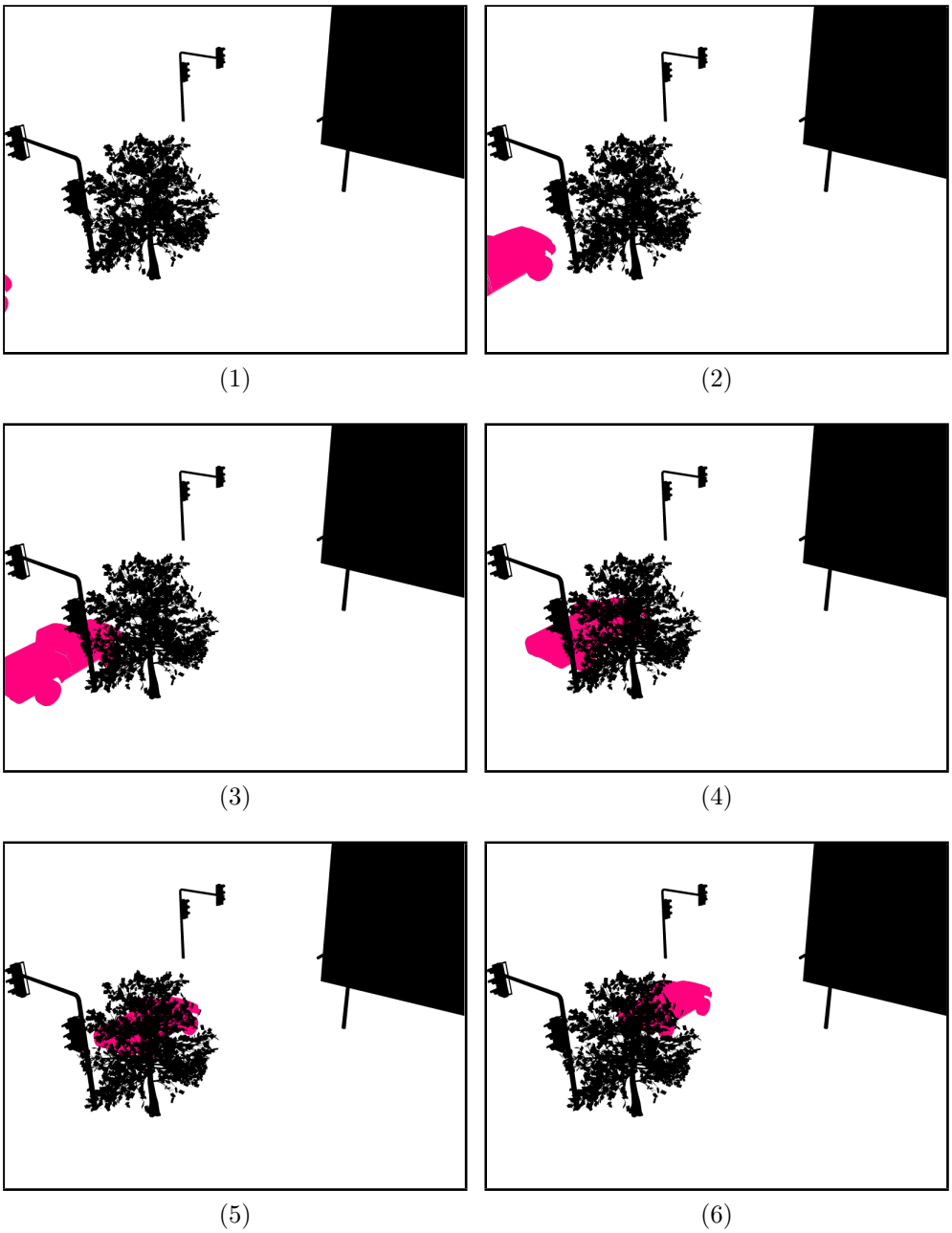


(29)

(30)

Figure C.5: Test images (continued).

C.1.3 Ground Truth Images

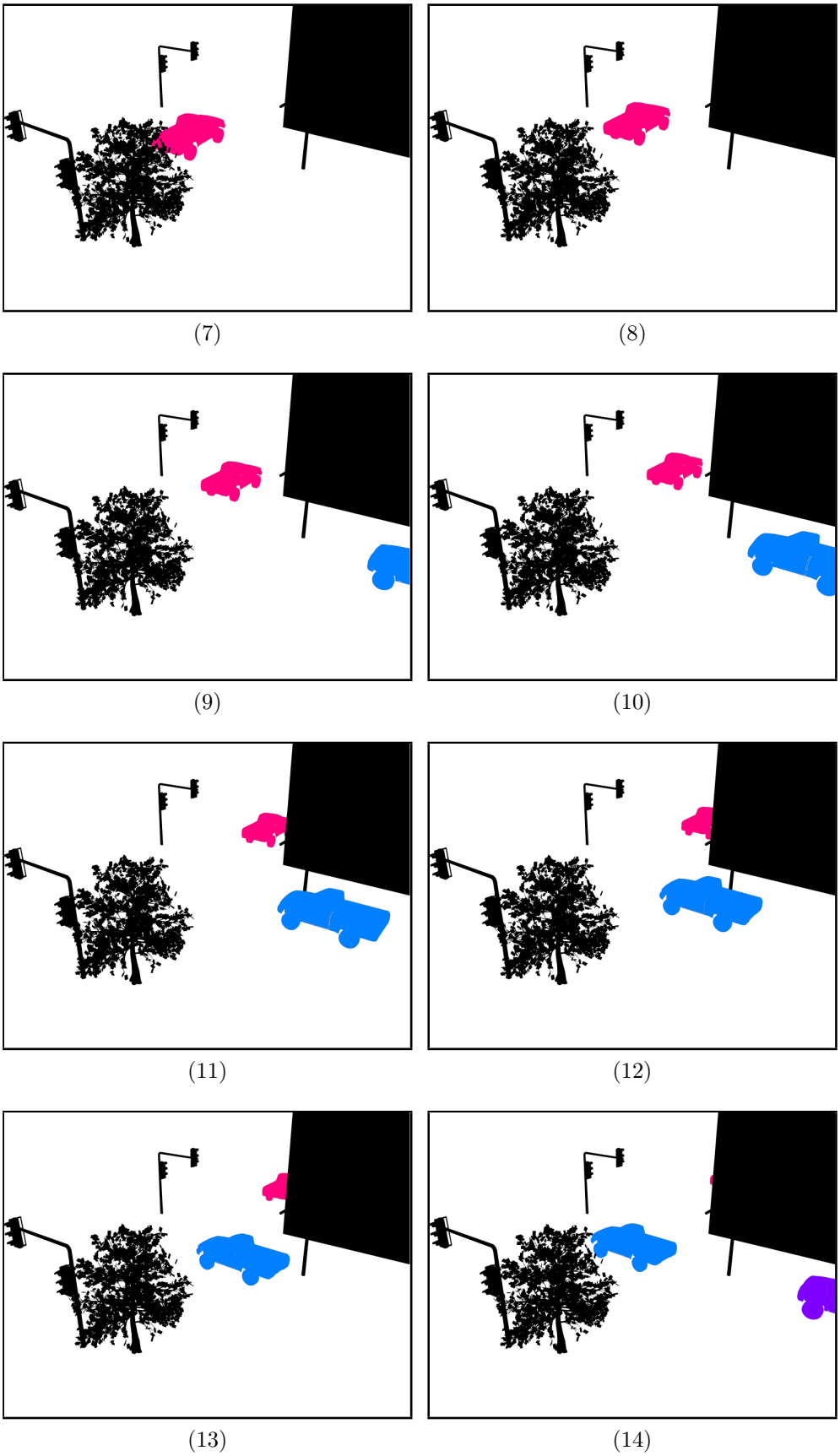


(1) (2)

(3) (4)

(5) (6)

Figure C.6: Ground truth foreground and background.



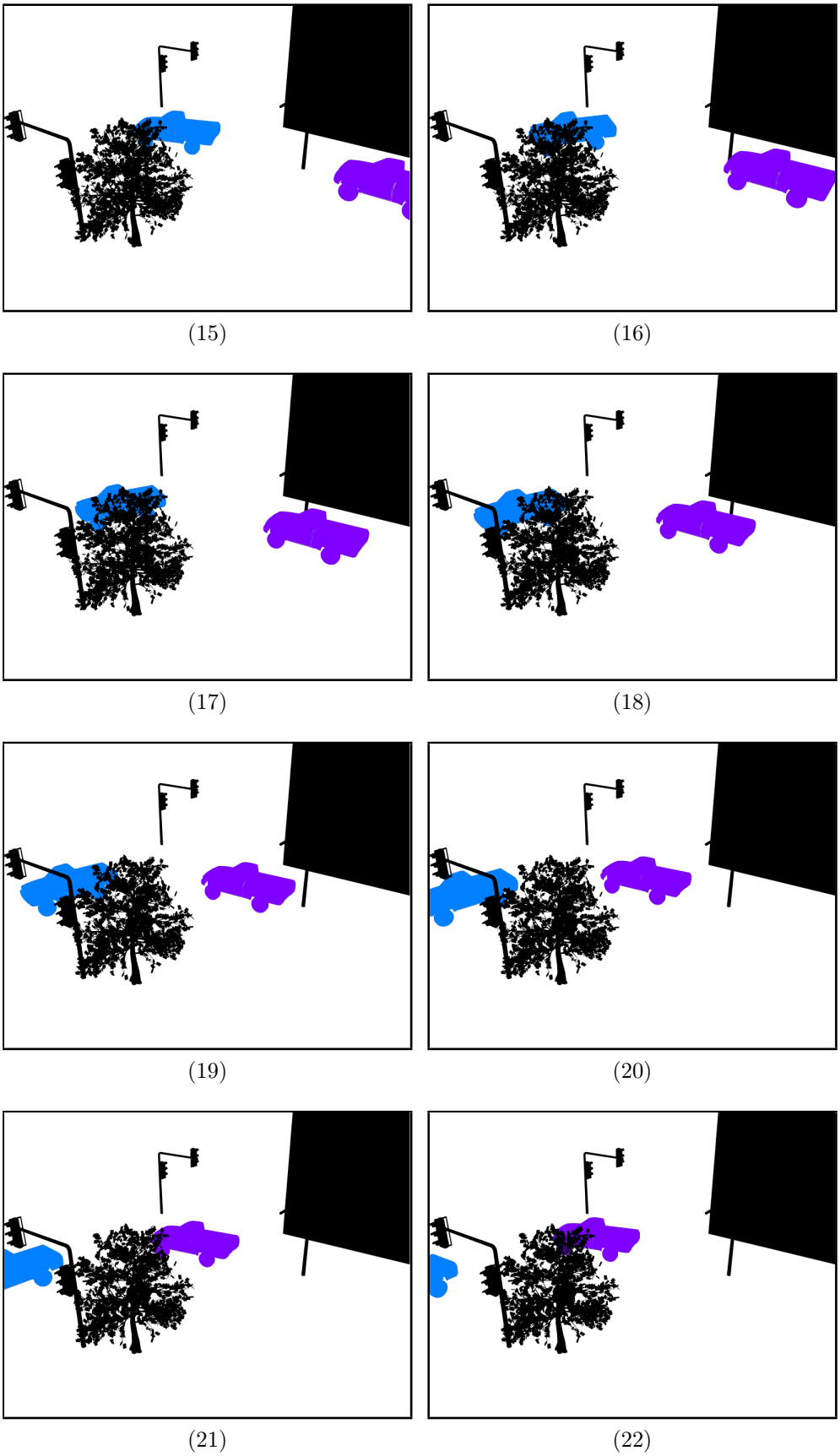
(7) (8)

(9) (10)

(11) (12)

(13) (14)

Figure C.7: Ground truth foreground and background (continued).



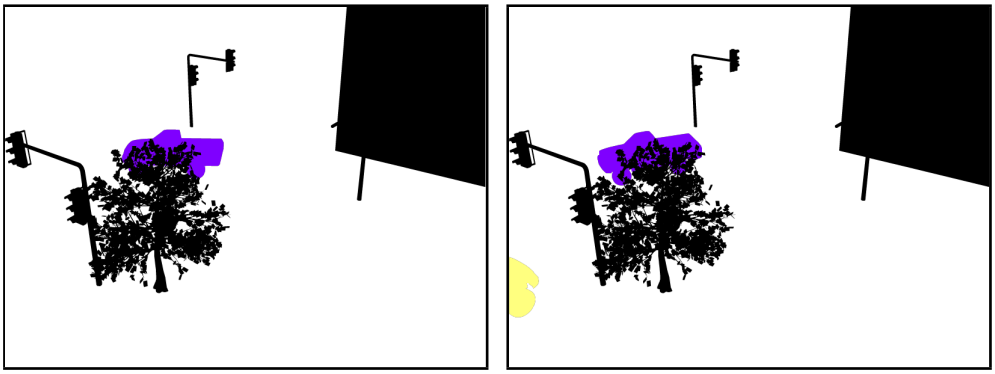
(15) (16)

(17) (18)

(19) (20)

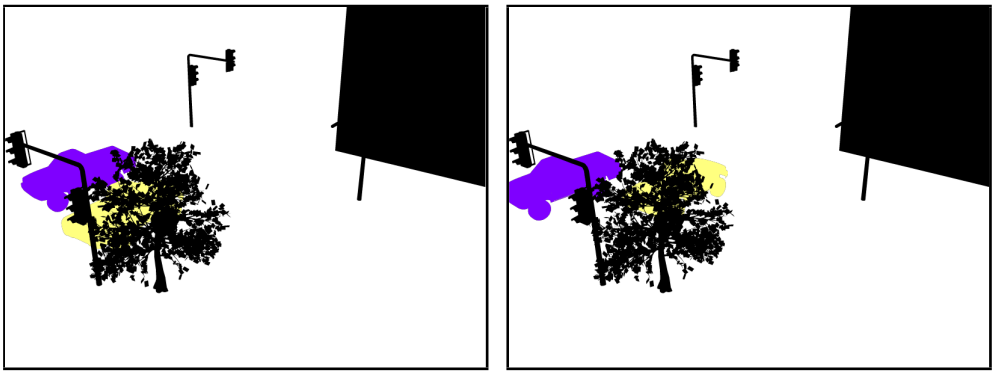
(21) (22)

Figure C.8: Ground truth foreground and background (continued).

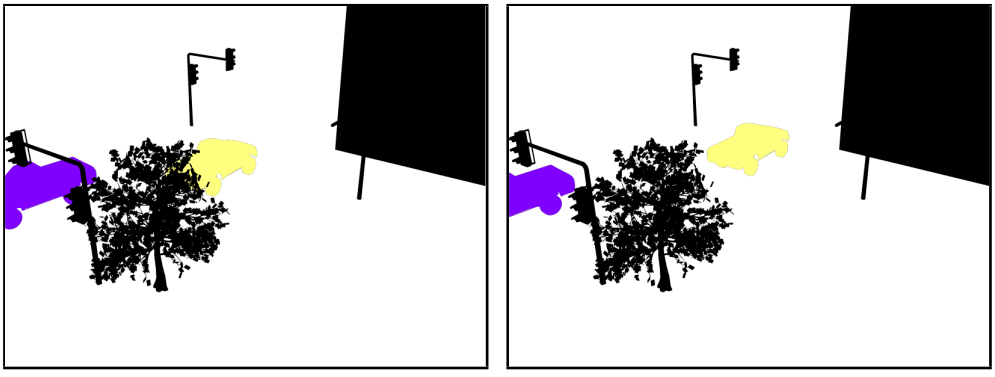


(23) (24)

(25) (26)



(27) (28)



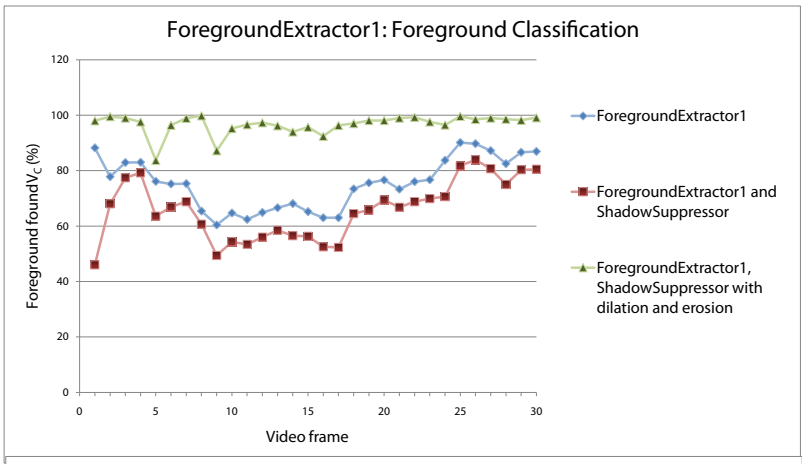
(29) (30)

Figure C.9: Ground truth foreground and background (continued).

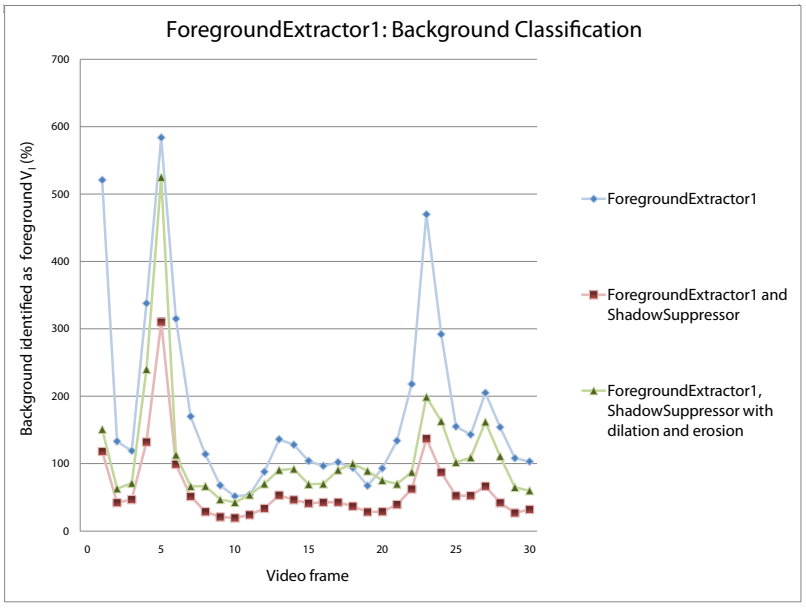
Appendix D

Artificial Background Subtraction Test Results

D.1 Image Extractor Performance

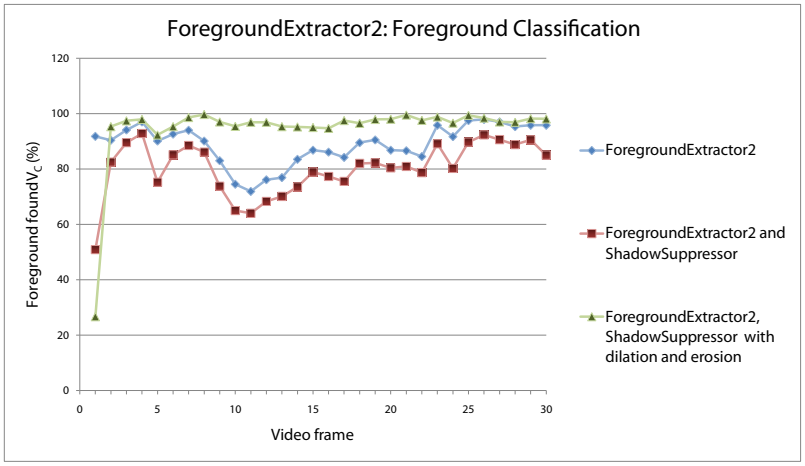


(1)

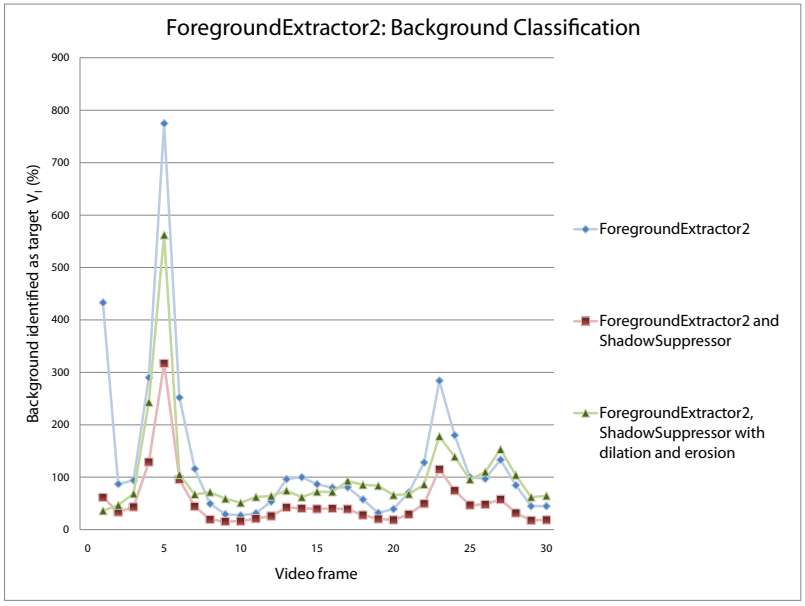


(2)

Figure D.1: The accuracy of ForegroundExtractor1.

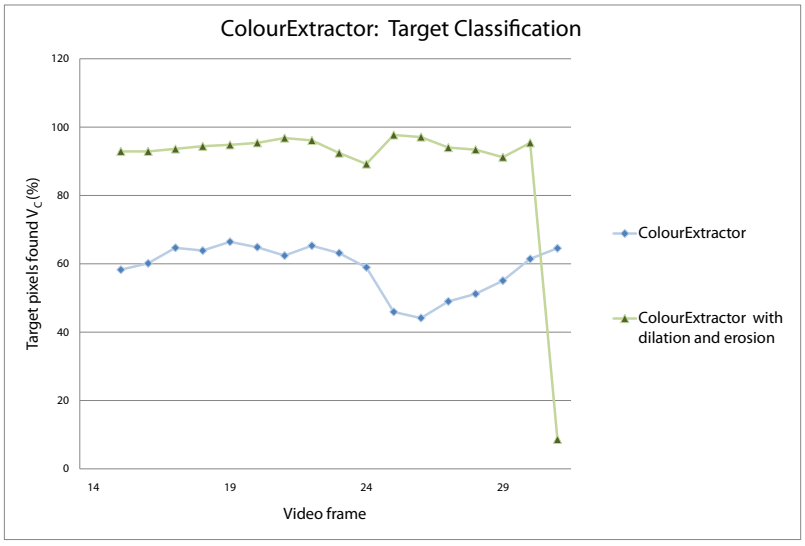


(1)

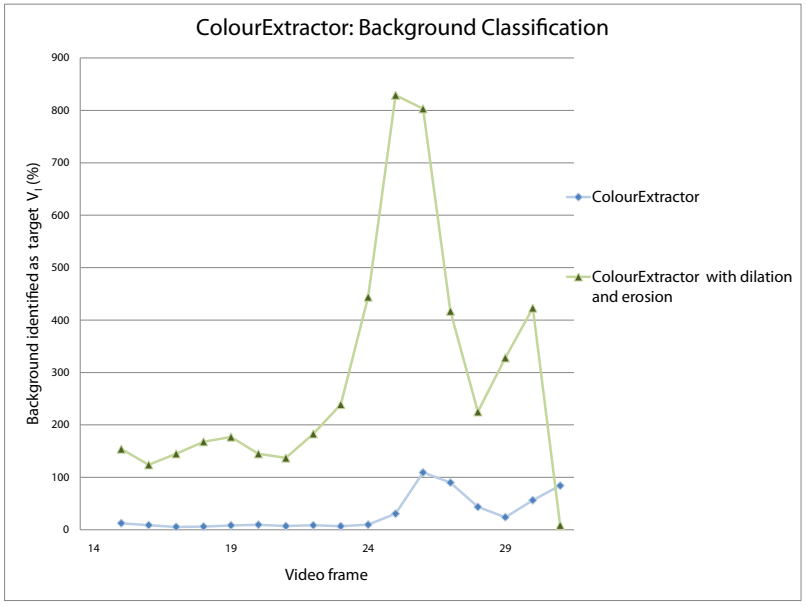


(2)

Figure D.2: The accuracy of ForegroundExtractor2.



(1)

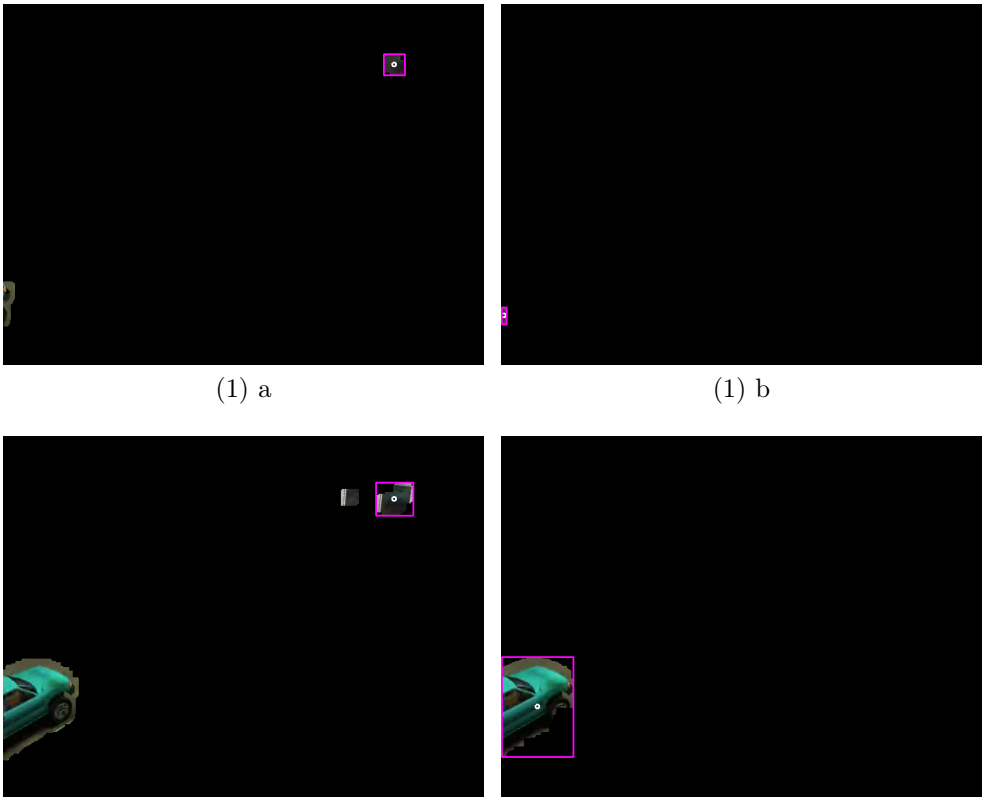


(2)

Figure D.3: The accuracy of ColourExtractor.

D.2 Extractor Output Images

D.2.1 Foreground Extractors with Filters



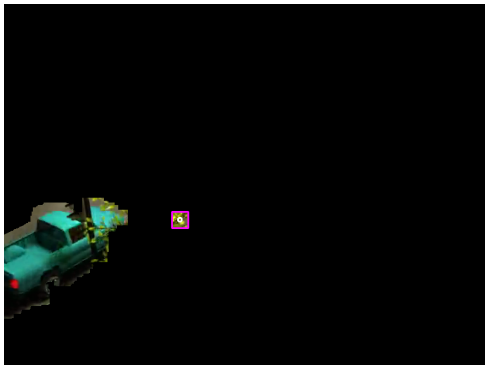
(1) a (1) b

(2) b (2) b

Figure D.4: Foreground extractor output images.

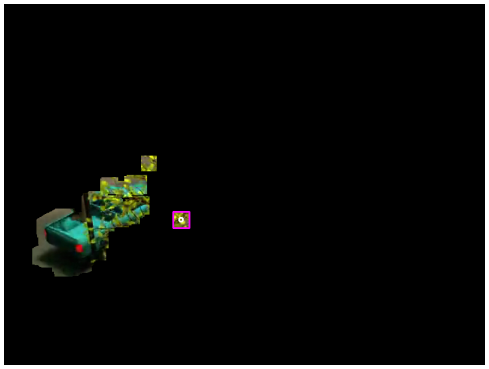
D.2.2 Colour Extractors with Filters

(3) a

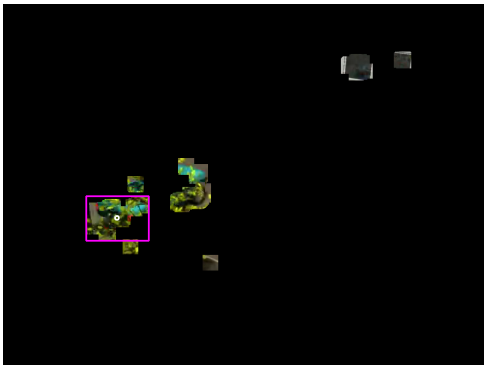


(3) b

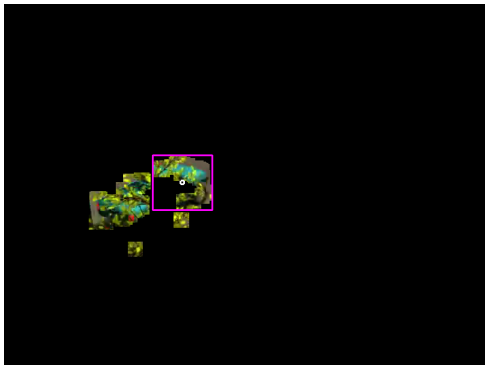
(4) a



(4) b

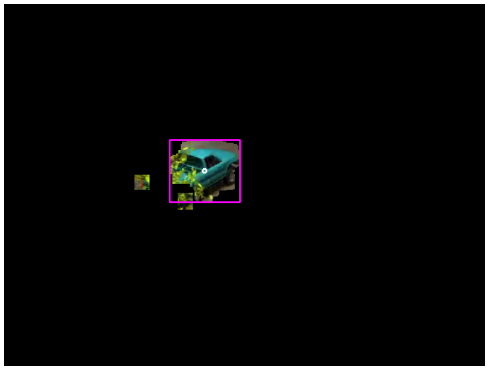


(5) a



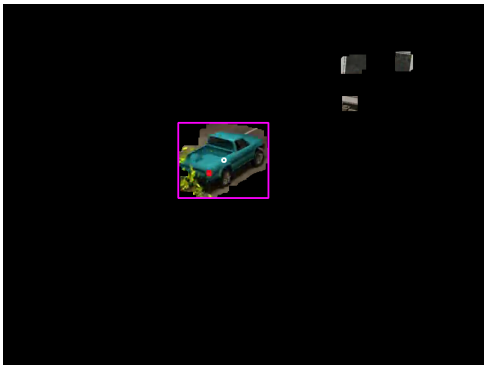
(5) b

(6) a

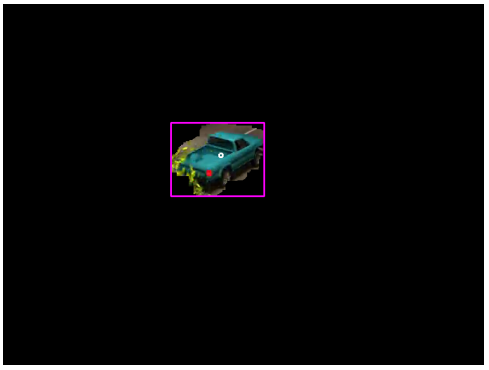


(6) b

Figure D.5: Foreground extractor output images (continued).



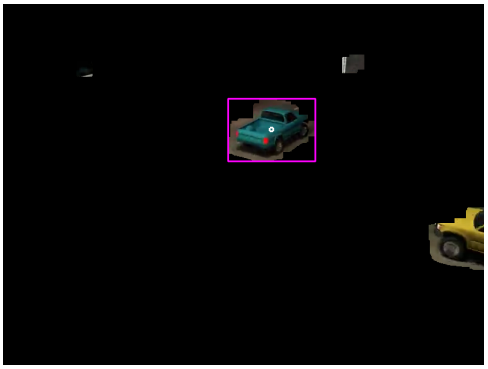
(7) a



(7) b

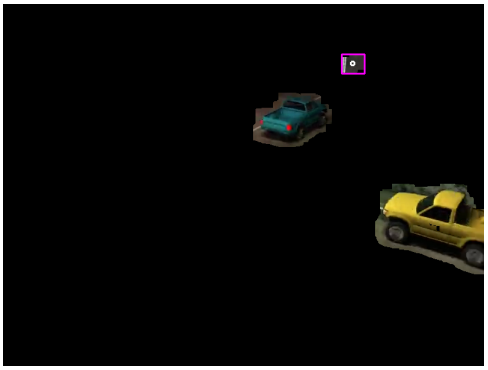
(8) a

(8) b



(9) a

(9) b



(10) a

(10) b

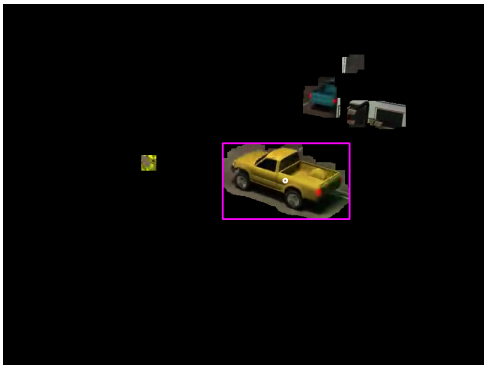
Figure D.6: Foreground extractor output images (continued).

(11) a

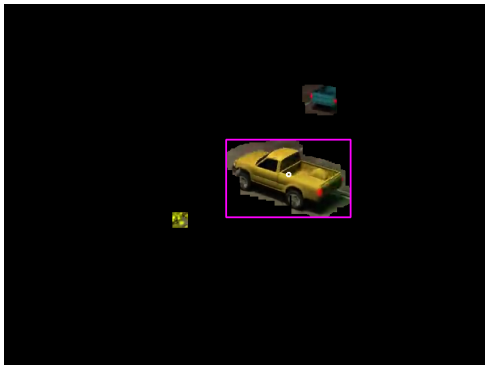
(11) b

(12) a

(12) b



(13) a



(13) b

(14) a

(14) b

Figure D.7: Foreground extractor output images (continued).

(15) a

(15) b

(16) a

(16) b

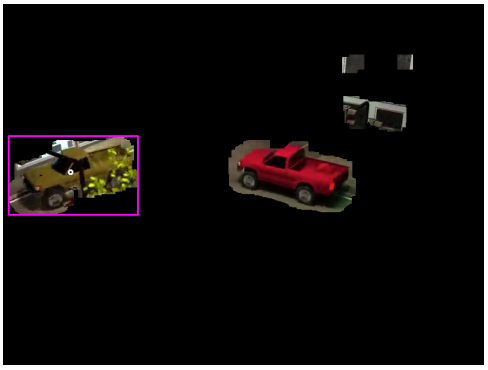
(17) a

(17) b

(18) a

(18) b

Figure D.8: Foreground extractor output images (continued).



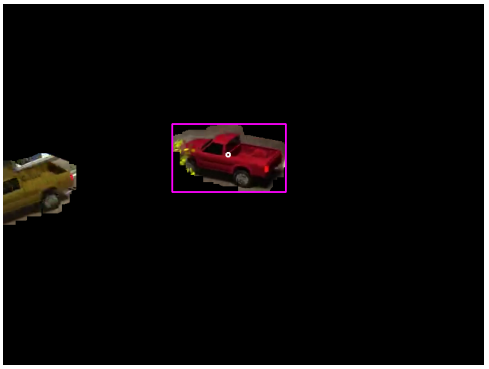
(19) a

(19) b

(20) a

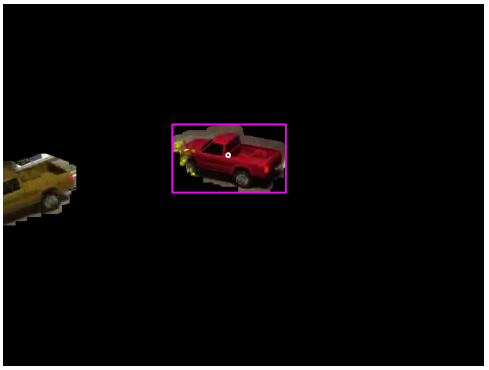
(20) b

(21) a



(21) b

(22) a

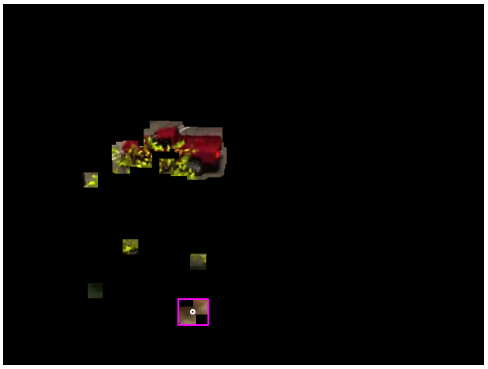


(21) b

Figure D.9: Foreground extractor output images (continued).

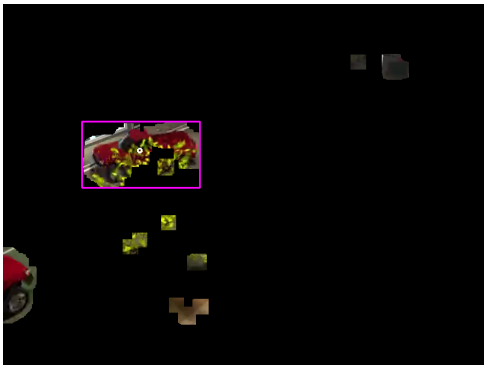
(22) a

(22) b



(23) a

(23) b



(24) a

(24) b

(25) a

(25) b

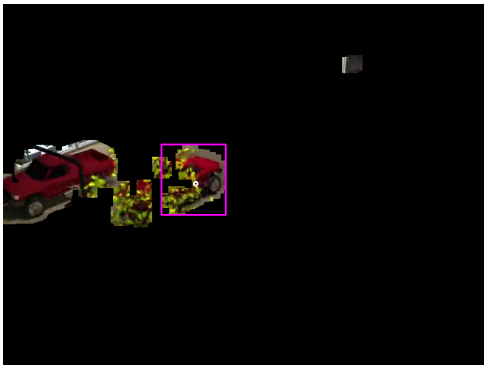
Figure D.10: Foreground extractor output images (continued).

(26) a

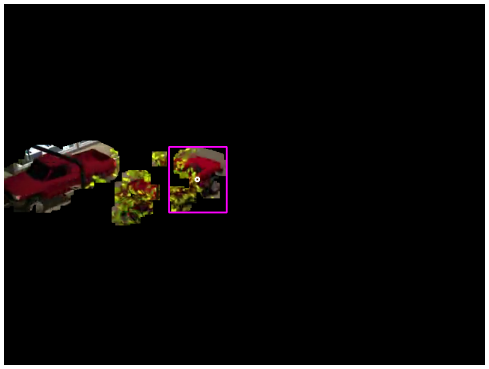
(26) b

(27) a

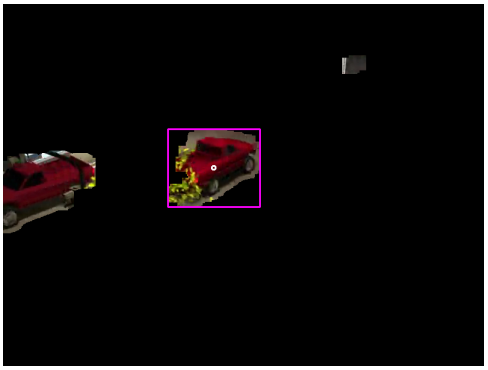
(27) b



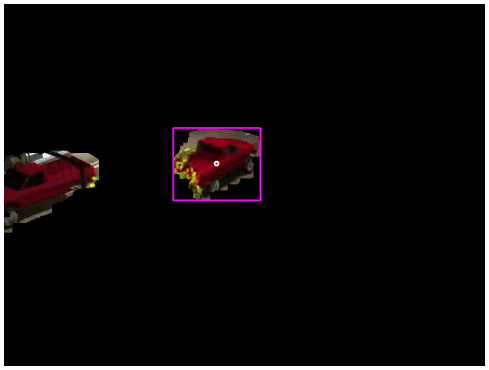
(28) a



(28) b

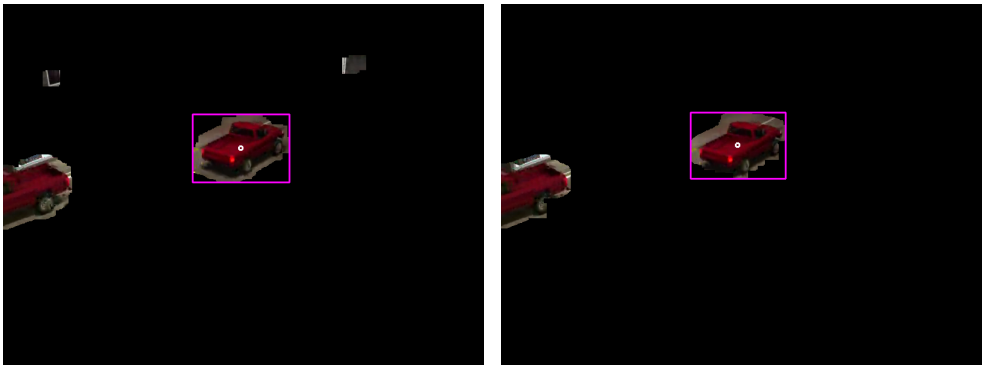


(29) a



(29) b

Figure D.11: Foreground extractor output images (continued).

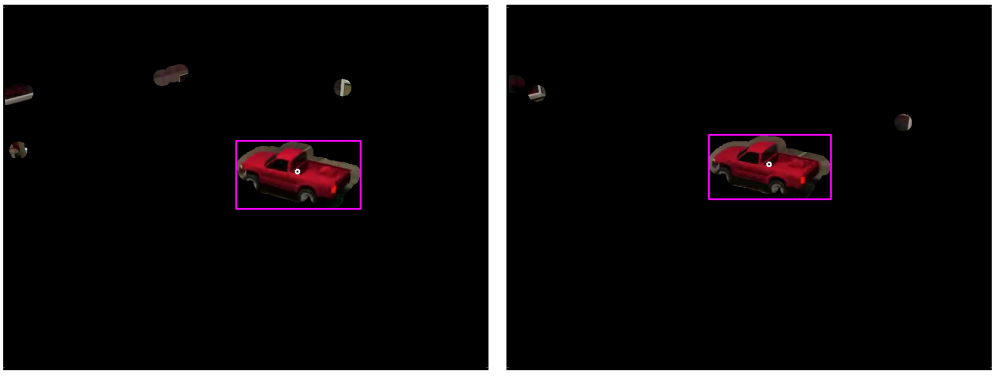


(30) a (30) b

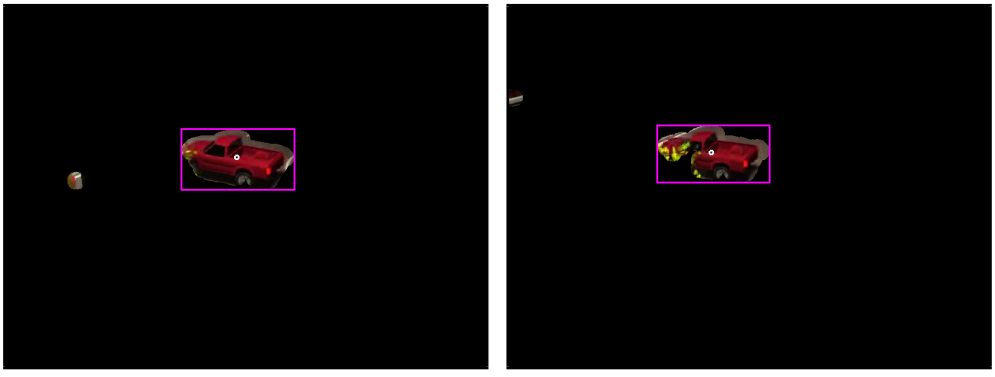
Figure D.12: Foreground extractor output images (continued).

(15) (16)

(17) (18)



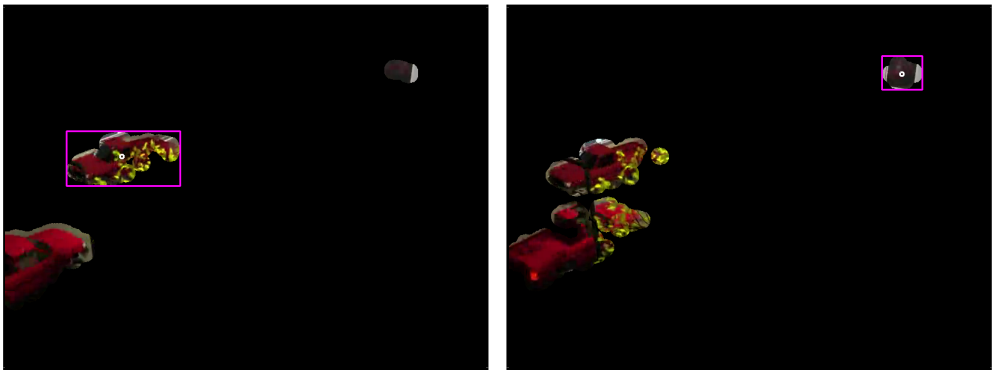
(19) (20)



(21) (22)

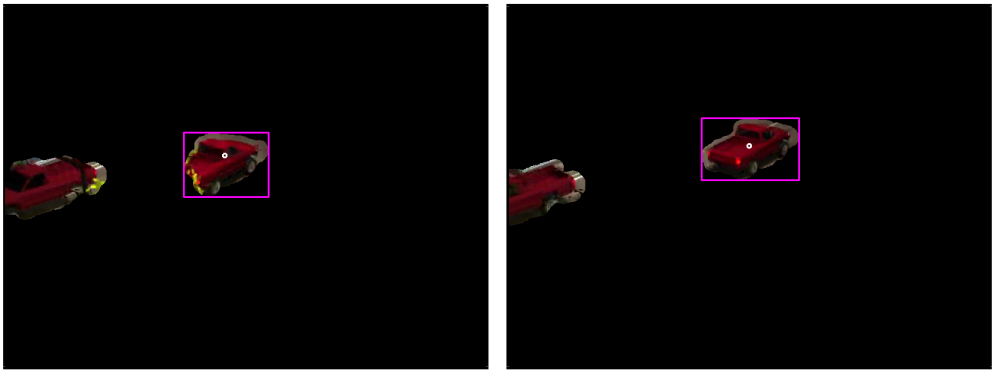
Figure D.13: Colour extractor output images.

(23) (24)



(25) (26)

(27) (28)



(29) (30)

Figure D.14: Colour extractor output images (continued).

Appendix E

Framework demonstration application

E.1 DVD content

- **HandTracking.avi**. A sphere is rendered over the hand to show how the object is tracked.
- **Collision.avi**. This video shows how collisions between the hand and the virtual objects are monitored.
- **Final.avi**. A demonstrational video of the application in its final form.

The videos are recorded using a DIVX compatible codec.