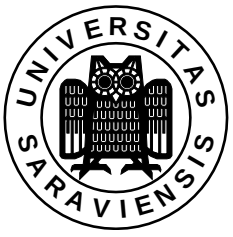
# Interactive Mixed Reality Rendering in a Distributed Ray Tracing Framework

**Andreas Pomi**

**Computer Graphics Group**
**Saarland University**
**Saarbrücken, Germany**

Dissertation zur Erlangung des Grades
*Doktor der Ingenieurwissenschaften (Dr.-Ing.)*
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes



**Betreuender Hochschullehrer / Supervisor:**

Prof. Dr.-Ing. Philipp Slusallek, Universität des Saarlandes, Saarbrücken, Germany

**Gutachter / Reviewers:**

Prof. Dr.-Ing. Philipp Slusallek, Universität des Saarlandes, Saarbrücken, Germany

Dr. Marcus Magnor, MPI Informatik, Saarbrücken, Germany

**Dekan / Dean:**

Prof. Dr. Jörg Eschmeier

**Eingereicht am / Thesis submitted:**

6. Juni 2005 / June 6th, 2005

**Datum des Kolloquiums / Date of defense:**

20. Juli 2005 / July 20th, 2005

**Prüfungskommission / Committee:**

Prof. Hans-Peter Seidel, MPI Saarbrücken
Prof. Philipp Slusallek, Universität des Saarlandes
Dr. Marcus Magnor, MPI Saarbrücken
Dr. Marco Lohse, Universität des Saarlandes

Andreas Pomi
Lehrstuhl für Computergraphik, Geb. 36.1
Universität des Saarlandes
Im Stadtwald, 66123 Saarbrücken
apomi@graphics.cs.uni-sb.de

# Abstract

The recent availability of interactive ray tracing opened the way for new applications and for improving existing ones in terms of quality. Since today CPUs are still too slow for this purpose, the necessary computing power is obtained by connecting a number of machines and using distributed algorithms. *Mixed reality rendering* — the realm of convincingly combining real and virtual parts to a new composite scene — needs a powerful rendering method to obtain a photorealistic result. The ray tracing algorithm thus provides an excellent basis for photorealistic rendering and also advantages over other methods. It is worth to explore its abilities for interactive mixed reality rendering.

This thesis shows the applicability of interactive ray tracing for mixed (MR) and augmented reality (AR) applications on the basis of the OpenRT framework. Two extensions to the OpenRT system are introduced and serve as basic building blocks: *streaming video textures* and *in-shader AR view compositing*. Streaming video textures allow for inclusion of the real world into interactive applications in terms of imagery. The AR view compositing mechanism is needed to fully exploit the advantages of modular shading in a ray tracer.

A number of example applications from the entire spectrum of the Milgram Reality-Virtuality continuum illustrate the practical implications. An implementation of a classic AR scenario, inserting a virtual object into live video, shows how a *differential rendering* method can be used in combination with a custom build real-time lightprobe device to capture the incident light and include it into the rendering process to achieve convincing shading and shadows. Another field of mixed reality rendering is the insertion of real actors into a virtual scene in real-time. Two methods — video billboards and a live 3D visual hull reconstruction — are discussed.

The implementation of live mixed reality systems is based on a number of technologies beside rendering and a comprehensive understanding of related methods and hardware is necessary. Large parts of this thesis hence deal with the discussion of technical implementations and design alternatives. A final summary discusses the benefits and drawbacks of interactive ray tracing for mixed reality rendering.

# Kurzfassung

Die Verfügbarkeit von interaktivem Ray-Tracing ebnet den Weg für neue Anwendungen, aber auch für die Verbesserung der Qualität bestehener Methoden. Da die heute verfügbaren CPUs noch zu langsam sind, ist es notwendig, mehrere Maschinen zu verbinden und verteilte Algorithmen zu verwenden. *Mixed Reality Rendering* — die Technik der überzeugenden Kombination von realen und synthetischen Teilen zu einer neuen Szene — braucht eine leistungsfähige Rendering-Methode um photorealistische Ergebnisse zu erzielen. Der Ray-Tracing-Algorithmus bietet hierfür eine exzellente Basis, aber auch Vorteile gegenüber anderen Methoden. Es ist naheliegend, die Möglichkeiten von Ray-Tracing für Mixed-Reality-Anwendungen zu erforschen.

Diese Arbeit zeigt die Anwendbarkeit von interaktivem Ray-Tracing für Mixed-Reality (MR) und Augmented-Reality (AR) Anwendungen anhand des OpenRT-Systems. Zwei Erweiterungen dienen als Grundbausteine: *Videotexturen* und *In-Shader AR View Compositing*. Videotexturen erlauben die reale Welt in Form von Bildern in den Rendering-Prozess mit einzubeziehen. Der View-Compositing-Mechanismus is notwendig um die Modularität einen Ray-Tracers voll auszunutzen.

Eine Reihe von Beispielanwendungen von beiden Enden des Milgramschen Reality-Virtuality-Kontinuums verdeutlichen die praktischen Aspekte. Eine Implementierung des klassischen AR-Szenarios, das Einfügen eines virtuellen Objektes in eine Live-Übertragung zeigt, wie mittels einer *Differential Rendering* Methode und einem selbstgebauten Gerät zur Erfassung des einfallenden Lichts realistische Beleuchtung und Schatten erzielt werden können. Ein anderer Anwendungsbereich ist das Einfügen einer realen Person in eine künstliche Szene. Hierzu werden zwei Methoden besprochen: Video-Billboards und eine interaktive 3D Rekonstruktion.

Da die Implementierung von Mixed-Reality-Anwendungen Kentnisse und Verständnis einer ganzen Reihe von Technologien nebem dem eigentlichen Rendering voraus setzt, ist eine Diskussion der technischen Grundlagen ein wesentlicher Bestandteil dieser Arbeit. Dies ist notwenig, um die Entscheidungen für bestimmte Designalternativen zu verstehen. Den Abschluss bildet eine Diskussion der Vor- und Nachteile von interaktivem Ray-Tracing für Mixed Reality Anwendungen.

# Acknowledgements

I would like to thank a number of people for their help with the work on this thesis. Working on large software system like distributed interactive ray tracing is teamwork, of course.

First of all, I would like to thank Philipp Slusallek, my supervisor. He guided me in the last five years, pushed me forward and helped me with discussions and ideas.

Another thanks I owe my colleagues, (in alphabetical order) Carsten Benthin, Tim Dahmen (inTrace), Georg Demme, Andreas Dietrich, Heiko Friedrich, Krzysztof Kobus (inTrace), Marco Lohse, Gerd Marmitt, Michael Repplinger, Michael Scherbaum (inTrace), Jörg Schmittler, Ingo Wald, Sven Woop, and Hanna Schilt, the secretary of our computer graphics group. They helped me a lot with ideas, discussion and programming on all my projects.

I also want to thank all our students, in particular those who worked with me on the Mixed Reality projects in the last years: Tim Dahmen, Benjamin Deutsch, Kim Herzig, Simon Hoffmann, Christian Linz, Benjamin Peters, and Stefan Schüffler.

A special thanks goes to Stefan Schüffler for helping me to set up the studio lab and to Simon Hoffmann, who worked a long time with me and helped me a lot in maintaining the studio.

Further I want to thank my colleagues from the Max-Plank-Institute for Computer Science (MPII) at the department AG4, led by Prof. Hans-Peter Seidel. Special thanks also goes to Marcus Magnor at the MPII for reviewing this thesis.

Another thanks goes to the SysAdmin Team (Bonsai) of the computer graphics group: Georg Demme, Rainer Jochem, and Maik Schmidt.

Finally, and most important, I want to thank my parents, Waltraud und Rolf Pomi, who supported me all time with my computer science studies and my best friend Daniel Bach, who always reminds me of what's really important.

# Contents


**1 Introduction** **1**

**2 Interactive Ray Tracing and the OpenRT System** **5**
2.1 The General Ray Tracing Algorithm . . . . . 5
2.1.1 Ray Tracing Based Algorithms . . . . . 7
2.2 Interactive Ray Tracing . . . . . 7
2.2.1 GPU Based Interactive Ray Tracing . . . . . 7
2.2.2 Special Ray Tracing Hardware . . . . . 8
2.2.3 Software Based Interactive (Parallel) Ray Tracing . . . 8
2.3 The OpenRT System . . . . . 9
2.3.1 The OpenRT API . . . . . 10
2.3.2 Programmable Shaders . . . . . 11
2.3.3 The Rendering Object . . . . . 11
2.3.4 OpenRT Application Programs . . . . . 12
2.4 Application Examples for Interactive Ray Tracing . . . . . 13
2.4.1 Virtual Reality . . . . . 13
2.4.2 Augmented Reality and Mixed Reality . . . . . 14
2.4.3 Virtual Television Studios (Actor Insertion) . . . . . 14
2.4.4 Interactive Global Illumination . . . . . 15
2.4.5 Massive Models . . . . . 15
2.4.6 Volume Rendering . . . . . 16
2.4.7 Games . . . . . 16

**3 An Introduction to Mixed Reality Rendering** **19**
3.1 Mixed Reality . . . . . 19
3.1.1 Augmented Reality . . . . . 20
3.1.2 Augmented Virtuality . . . . . 21
3.2 Related Rendering Techniques . . . . . 21
3.2.1 Shadow Generation in MR . . . . . 22
3.2.2 Common Illumination . . . . . 22
3.2.3 Image-Based Lighting . . . . . 23

3.2.4 Sampling of Incident Lightmaps . . . 23
3.2.5 Relighting Methods . . . 24
3.2.6 Inverse Rendering Methods . . . 25
3.2.7 Precomputed Radiance Transfer Methods . . . 25
3.2.8 Environment Matting . . . 25

**4 Streaming Video Textures** **27**
4.1 Video Textures . . . 28
4.2 Video Data Distribution . . . 29
4.2.1 OpenRT Payload . . . 29
4.2.2 A Demand Driven Approach . . . 30
4.2.3 Direct Video Connection . . . 31
4.2.4 Multicast Networking . . . 32
4.3 The OpenRT Video Texture Subsystem . . . 34
4.3.1 The System Architecture . . . 34
4.3.2 Synchronization . . . 34
4.3.3 Packetizing . . . 36
4.3.4 Texture Data Formats . . . 37
4.3.5 Network Packet Loss . . . 37
4.3.6 The OpenRT Video Texture API . . . 39
4.4 A Video Texture Example Application . . . 39
4.4.1 Lighting from Video Textures . . . 40
4.4.2 Results . . . 41
4.5 Conclusion and Future Work . . . 41

**5 Video Billboards** **43**
5.1 Virtual Television Studios . . . 43
5.1.1 Video Compositing for Virtual Studios . . . 45
5.1.2 Consistent Lighting . . . 46
5.2 Foreground Segmentation . . . 48
5.2.1 Garbage Matte . . . 48
5.2.2 Chroma Keying . . . 48
5.2.3 Invisible Keying . . . 50
5.2.4 Background Subtraction . . . 51
5.3 Video Billboards . . . 52
5.3.1 The Concept of In-Shader Compositing . . . 53
5.4 An OpenRT Video Billboard Example . . . 55
5.4.1 Hardware Setup . . . 55
5.4.2 OpenRT Setup . . . 56
5.4.3 A Billboard Shader . . . 56
5.4.4 Chroma Keying . . . 57

5.4.5 Results . . . 59
5.5 Drawbacks of Billboards . . . 61
5.6 Conclusion and Future Work . . . 63

**6 Augmented Reality View Compositing 65**
6.1 Video-Based Augmented Reality . . . 65
6.1.1 Camera Tracking . . . 66
6.1.2 AR Compositing . . . 67
6.2 The Concept of In-Shader Compositing for Augmented Reality 68
6.3 Differential Rendering . . . 69
6.3.1 Stand-In Geometry . . . 71
6.4 AR View Compositing in OpenRT . . . 72
6.4.1 AR View Video Streaming . . . 72
6.4.2 Tonemapping . . . 74
6.4.3 A Differential Rendering Example . . . 75
6.4.4 Results . . . 76
6.5 Conclusion and Future Work . . . 76

**7 A Real-Time Lightprobe 79**
7.1 Measuring Incident Light . . . 81
7.1.1 Digital Image Sensors . . . 82
7.1.2 High Dynamic Range Cameras . . . 83
7.1.3 True High Dynamic Range Sensors . . . 84
7.1.4 Spatially Varying Pixel Exposures . . . 84
7.1.5 Spatially Varying Image Exposures . . . 85
7.1.6 Multiple Sensors . . . 85
7.1.7 Sequential Multiple Exposures . . . 86
7.2 Principles of Multiple Exposure High Dynamic Range Imaging 87
7.2.1 Camera System Response Function . . . 89
7.2.2 Image Reconstruction . . . 91
7.3 Panoramic Acquisition . . . 93
7.3.1 Mirror Balls . . . 93
7.3.2 Fish-Eye Lens . . . 94
7.3.3 Moving Cameras . . . 95
7.3.4 Multi-Sensor Rigs . . . 96
7.4 Restrictions of a Single Panoramic Lightprobe . . . 96
7.4.1 Acquiring Incident Lightfields . . . 98
7.5 A Real-Time Lightprobe . . . 99
7.5.1 Building a Simple Video Lightprobe . . . 100
7.5.2 Results . . . 103
7.6 An OpenRT IBL Application Example . . . 104

7.6.1 Hardware Setup . . . 104
7.6.2 OpenRT Setup . . . 105
7.6.3 Light Sample Generation . . . 105
7.6.4 Shadows and Reflections of Virtual Object in the Video Background . . . 107
7.6.5 Ambient Occlusion . . . 109
7.6.6 Lighting the Virtual Objects . . . 110
7.6.7 Results . . . 111
7.7 Conclusion and Future Work . . . 112

**8 In-Shader Image Based Visual Hull Reconstruction 115**
8.1 Interactive 3D Reconstruction Methods . . . 116
8.1.1 Related Work . . . 116
8.2 Towards a Visual Hull Shader . . . 117
8.3 Silhouette Acquisition . . . 119
8.3.1 A Calibrated Multi-Camera Setup . . . 120
8.3.2 Foreground Segmentation . . . 121
8.3.3 Silhouette Data Compression . . . 121
8.4 An OpenRT Visual Hull Shader Example . . . 122
8.4.1 Data Acquisition . . . 122
8.4.2 The Compression Method . . . 123
8.4.3 Image Based Ray Traversal . . . 124
8.4.4 An OpenRT Visual Hull Shader . . . 126
8.4.5 View Dependent Texturing . . . 129
8.4.6 Surface Normals . . . 131
8.4.7 Results . . . 132
8.5 Conclusion and Future Work . . . 133

**9 Final Summary 137**

**A The CTools Suite 145**

**B A Triggering Interface for Sony DFW Cameras 147**

**C The OpenRT Video Texture API 149**

**D An OpenRT Video Billboard Shader Example 151**

**E The Studio Lab 155**

**Bibliography 157**

# List of Figures


2.1 The General Ray Tracing Algorithm . . . . . 6
2.2 SaarCOR Application Example . . . . . 8
2.3 Plug-and-Play Shading . . . . . 9
2.4 The OpenRT System Hardware Setup . . . . . 10
2.5 Example Applications of an OpenRT VRML Scenegraph . . . 12
2.6 Virtual Reality Example Applications . . . . . 13
2.7 Actor Insertion Example Applications . . . . . 14
2.8 Interactive Global Illumination Example Applications . . . . . 15
2.9 Massive Model Example Applications . . . . . 16
2.10 Volume Rendering Example Applications . . . . . 17
2.11 Game Example Applications . . . . . 17

3.1 The Milgram RV Continuum . . . . . 20

4.1 Video Texture Application Examples . . . . . 28
4.2 Video Texture Distribution: OpenRT Payload . . . . . 30
4.3 Video Texture Distribution: Demand Driven . . . . . 31
4.4 Video Texture Distribution: Direct Connection . . . . . 32
4.5 Video Texture Distribution: Multicast . . . . . 33
4.6 The OpenRT Video Texture Subsystem . . . . . 35
4.7 A Video Texture Packet . . . . . 36
4.8 A Video Texture Example Application . . . . . 39
4.9 Lighting from Video Textures . . . . . 40

5.1 Virtual Studio Hardware Setup . . . . . 45
5.2 Lighting for the BBC Truematte Technology . . . . . 49
5.3 Invisible Infrared Keying . . . . . 51
5.4 Traditional vs. In-Shader Compositing . . . . . 54
5.5 Video Billboard Example Hardware Setup . . . . . 55
5.6 A Billboard Application . . . . . 56
5.7 A Billboard Shader . . . . . 57

5.8 Practical Chroma Keying . . . 58
5.9 Chroma Keying Scale Factor . . . 59
5.10 Billboard Results . . . 60
5.11 Billboard Distortion . . . 61
5.12 Billboard Floating . . . 62
5.13 Billboard Mirror . . . 62
5.14 Billboard Shadows . . . 63

6.1 A Video-Based Augmented Reality System . . . 66
6.2 OpenRT AR Compositing Data Flow . . . 72
6.3 Differential Rendering 1 . . . 74
6.4 Differential Rendering 2 . . . 75

7.1 An AR Example with OpenRT . . . 79
7.2 Incident Light . . . 80
7.3 The Photographers Incident Lightmeter . . . 81
7.4 A CCD Sensor and Vertical Smear . . . 82
7.5 A Spatially Varying Pixel Exposure Sensor . . . 84
7.6 A Spatially Varying Image Exposures System . . . 85
7.7 A Beam-Splitter HDR Camera . . . 86
7.8 An Exposure Sequence . . . 88
7.9 The Camera Response Curve . . . 89
7.10 The HDR Compositing Principle . . . 89
7.11 The Response Curve of a JAI CV-S3300 Camera . . . 92
7.12 Mirror Ball Shots . . . 94
7.13 Moving Panorama Cameras . . . 95
7.14 The Point Grey Research Ladybug Camera . . . 97
7.15 Failure of The Environment Map Assumption . . . 98
7.16 Incident Lightfield Acquisition . . . 99
7.17 The ICT Real-Time Lightprobe . . . 100
7.18 Components of our Real-Time Lightprobe . . . 101
7.19 The Real-Time Lightprobe . . . 102
7.20 IBL Example Hardware Setup . . . 105
7.21 Light Sample Generation . . . 106
7.22 Lightprobe Warping . . . 107
7.23 An OpenRT IBL Example . . . 111
7.24 A Virtual Car in a Real Background . . . 112

8.1 A Visual Hull Shader in a VR Scene . . . 115
8.2 Visual Hull Shader System Principle . . . 118
8.3 An Intersection Example . . . 119

8.4 The Input Views . . . . . 123
8.5 Image Compression . . . . . 124
8.6 Image-Based Visual Hull Intersection . . . . . 125
8.7 Image-Based Visual Hull Intersection Flowchart . . . . . 126
8.8 Pixel Exact Occlusion . . . . . 127
8.9 Visual Hull Shader Flowchart . . . . . 128
8.10 View Occlusion Artefacts . . . . . 130
8.11 Visual Hull Normals . . . . . 132
8.12 Visual Hull Shader Examples . . . . . 134

A.1 The CEdit System . . . . . 145

B.1 A Triggering Interface for Sony DFW Cameras . . . . . 147
B.2 Triggering Interface Schematics . . . . . 148

E.1 The Studio Lab . . . . . 155
E.2 The Studio Sphere . . . . . 156

# Chapter 1
# Introduction

Photorealistic rendering methods are required for convincing combinations of real and virtual scenes. High-Quality *Mixed Reality* (MR) rendering is the art of providing a seamless visual integration of both worlds, the digital and the analog one. This is a difficult task, however, since a number of means are needed to transfer the necessary information to provide consistency between these two very different representations.

The *ray tracing* algorithm, improved with a number of additions like *global illumination* simulation over the years, has proven its ability to provide realistic images for a long time. When it comes to interactive applications with an output of several frames per second, hardware (GPU, *Graphics Processing Unit*) based rendering has usually been preferred. The raster graphics algorithms, even on todays GPUs, are *brute force* like on the first days of computer graphics. Hardware resources are limited and force the programmer to many trade-offs in quality and speed. Applications are hard to port from one hardware platform to another due to differences in GPU design. [OpenGL] provides an industry standard API and manufacturers of modern graphics boards advertise *programmable shading* features, but in reality the possibilities are often limited due to hardware restrictions like code memory. Application of global effects like global illumination simulation on GPUs is difficult.

The recent availability of *interactive* ray tracing technology has opened the field for new applications and also for improvement of existing ones. The high image quality, that can be obtained with the ray tracing algorithm, can provide a new, yet unreached level of photorealism to applications. The modular properties of the algorithm help in simplification of the software design compared to traditional raster graphics methods.

Interactive ray tracing systems, like the OpenRT [OpenRT, Wald04a] framework, achieve the necessary rendering power by parallelization on a number of CPUs and by algorithmic improvements. Code optimization for modern CPU designs, supporting multiple numerical operations at the same time (SIMD), can provide an additional speedup. This results in interactive frame rates up to video rates. A system API, familiar to OpenGL programmers, allows for easy implementation or porting of application programs.

Since shared memory machines with a high number of CPUs are even today very seldom and expensive, it is necessary to connect a number of machines via a network. Algorithms need to be adapted, since maintaining global data structures in a distributed system is often not possible due to latency and network bandwidth limitations.

It is worth to explore the capabilities of modern, interactive ray tracing systems in the mixed reality field. Here, the availability of a higher realism can provide a new quality for interactive applications.

In this thesis, I give an overview of interactive mixed reality and augmented reality (AR) rendering based on the OpenRT framework. I provide the necessary background information and a number of application examples that I developed over the last years. These applications prove the applicability of interactive ray tracing for mixed reality rendering but also show up a huge field of future research possibilities.

The main contributions of this thesis to the field of ray tracing based interactive mixed reality rendering are

- the concept of *in-shader compositing* to combine shading and compositing in mixed reality applications,
- the extension of the OpenRT framework with streaming video textures to provide a means for representing the real world inside the rendering process,
- an AR view compositing mechanism for OpenRT suited to the special needs of distributed rendering,
- a 3D compositing method for actor insertion based on *in-shader* visual hull reconstruction and allowing for a seamless integration of a real person into the distributed ray tracing process at interactive frame rates,
- and the design and construction of a real-time lightprobe device based on a low cost video camera for including a real lighting situation into the rendering process.

## Outline of this thesis

Chapter 2 will provide an overview over interactive ray tracing technology with focus on distributed, cluster based systems. The OpenRT system will be discussed in some detail to provide the necessary background for the following chapters.

In Chapter 3, I will give a brief overview over the rendering techniques related to (traditional) mixed reality and related research work. The focus will remain on rendering and compositing aspects .

To provide support for mixed reality rendering in OpenRT, I introduce the concept of streaming video textures in Chapter 4. The necessary background for implementation in a distributed real-time rendering system is provided and an advanced application example is given.

Chapter 5 comes up with a description of the *actor insertion* problem related to virtual television studios. I describe the use of video billboards in a ray tracer to achieve rendering effects like shadows and reflections.

For the application of video-based augmented reality, Chapter 6 provides an extension to the OpenRT framework to allow the necessary compositing to be done inside the ray tracers shading process.

A more advanced AR application is described in Chapter 7: a virtual car is rendered into the live video backplate of a hall. Consistent lighting is achieved by a real-time lightprobe device. In this chapter, I provide the necessary technological background for building the lightprobe device and describe the image-based lighting method for the car.

To overcome the drawbacks of the video billboards from Chapter 5, a full 3D solution is introduced in Chapter 8. Here, an actor is reconstructed at interactive frame rates within an OpenRT shader. Correct interaction with the environment is the result.

I will close with a final summary over the benefits of using interactive ray tracing for mixed reality rendering. A number of short appendices provide additional technical informations to some of the projects.

# Chapter 2

# Interactive Ray Tracing and the OpenRT System

Even though known for decades, the ray tracing algorithm was never considered for interactive work until the recent years. Its flexibility due to the modular approach of intersection and shading makes it an interesting alternative to the raster graphics algorithms widely used for interactive graphics today.

This chapter provides a brief overview over (interactive) ray tracing and a description of the OpenRT ray tracing framework in some detail. It closes with a short discussion of typical application examples for interactive ray tracing.

## 2.1 The General Ray Tracing Algorithm

The idea of ray shooting was first used in [Appel68] for hidden surface rendering. Rays are traced from a virtual camera into the scene. After determining the closest intersection with a ray, the color for the related pixel in the image plane of the camera is calculated (*shading*). The same ray shooting concept can be used to determine the visibility of light sources from a point in the scene to test if a surface point lies in shadow (Figure 2.1).

In the shading process for a surface point, *secondary rays* for reflection, refraction and shadow test can be spawn. This can happen recursively. A single primary ray can thus result in a whole recursion tree of rays. This concept of *recursive ray tracing* was introduced by Whitted in [Whitted80].

For efficient intersection of a ray with the scene description an *acceleration data structure* is needed (e.g. BSP, kd-tree). The primitives need to

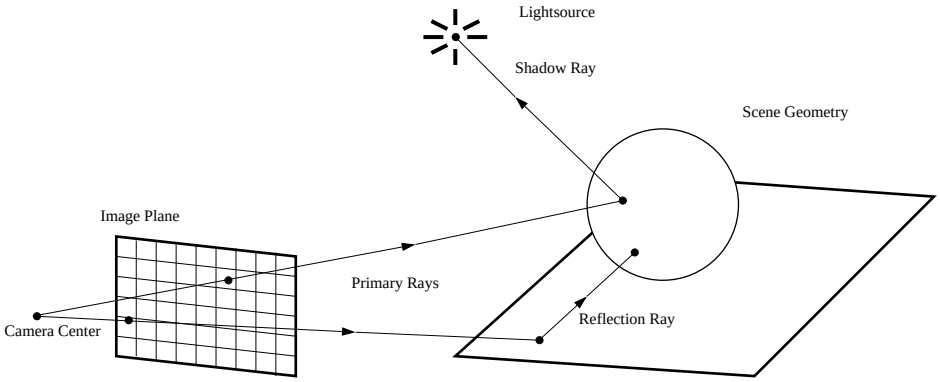



*Figure 2.1: The General Ray Tracing Algorithm. The camera on the left side is described by a center point and an image plane subdivided into pixel cells. Rays for each cell are intersected with the scene geometry. Two ray cases are shown: a primary ray hitting the sphere spawning a shadow ray for shading and another primary ray reflected on the floor plane.*

be spatially sorted in the data structure in a pre-process. Due to the logarithmic complexity of database searching, the ray tracing algorithm is better suited for complex scenes and models with a huge number of primitives compared to raster graphics using a *Z-Buffer* method, which has in general linear complexity.

The shading process in these early systems was limited to (ideal) mirror reflections and hard shadows and yielded artifical looking images. [Cook84] presents a system for glossy reflections (*distribution ray tracing*). Multiple primary rays per pixel and secondary rays per surface reflection in combination with spatial probability distributions allow for more realistic images featuring soft shadows, glossy reflections, depth-of field, and even motion blur.

Ray tracing is a *demand-driven* algorithm. Shading calculation (and thus spawning of secondary rays) is only performed when needed. The computational effort for the output image depends on the parts of the scene visible with the actual camera settings. The ray tracing algorithm is thus said to be *output sensitive*. This makes it a ideal method for complex scenes with large occluded parts.

Compared to raster graphics methods [Moeller99], ray tracing provides a more *straightforward and modular* way for implementing rendering systems. Complex multi-pass methods, like used with raster graphics for overcoming limited system resources, are not needed. Ray tracing also serves as a powerful frame work for more complex rendering algorithms.

### 2.1.1 Ray Tracing Based Algorithms

Most of the algorithms that are built on top of a ray tracing framework break with the classical concept of recursive ray tracing and use only the ray/scene intersection mechanism (*ray shooting*). A number of *Global Illumination* algorithms are based on ray shooting. Extending the concepts of [Cook84], algorithms like *bi-directional path tracing* [Lafortune93] and *photon mapping* [Jensen96] use a ray tracing framework for providing realistic and physically-correct lighting simulations.

There are a number of applications of ray shooting outside computer graphics. The basic ray tracing framework can, for example, be used for physical simulation of radio-wave propagation for mobile phone networks considering complex reflections at the walls of buildings. Another applications include particle tracing in nuclear processes or simulation of audio wave propagation.

## 2.2 Interactive Ray Tracing

With the availability of faster machines, larger main memory and complex GPUs, more and more research aiming towards interactive ray tracing came up in the last years (see [Wald01a] for an overview). There are several ways to achieve interactive frame rates: using GPUs, using special designed ray tracing hardware or just using plain CPU power.

### 2.2.1 GPU Based Interactive Ray Tracing

In [Purcell02] Purcell et al. showed that ray tracing at interactive frame rates is possible using commodity PC graphics boards. The scene data is stored in the on-board texture memory (which is very limited). Ray intersection is done via a multi-pass state machine over the ray state for each pixel. Shading is performed using the programmable *pixel shaders* of current GPU hardware.

Some parts of the algorithm like the state machine control has to be done by the CPU because the used GPU lacked branching and looping. A number of special hardware features like GPU support for texture indirection is needed. Purcell et al. [Purcell02] showed that it is even possible to implement Global Illumination algorithms like path tracing on GPU hardware.

Implementing interactive ray tracing on a GPU is limited by the current available hardware features. Design variations of the various manufacturers make porting of individual algorithms difficult.

Note that there are a number of publications that mix raster graphics and ray tracing effects by using rasterization for the 'simple' parts of the

scene and add ray tracing effects afterwards by tracing rays for a low number of pixels. This should not be confused with real, modular interactive ray tracing.

### 2.2.2 Special Ray Tracing Hardware

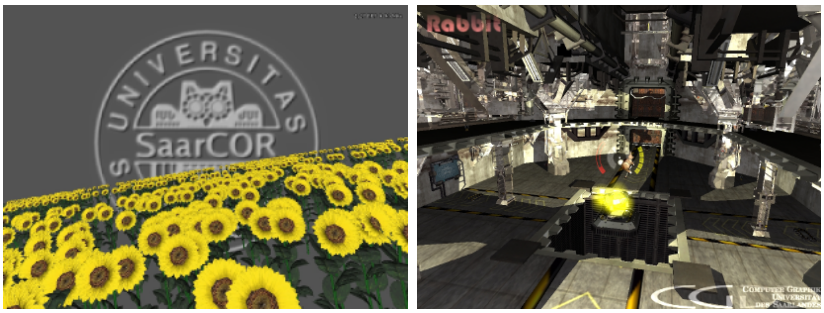



*Figure 2.2: Some screenshots rendered with the SaarCOR ray tracing hardware built at Saarland University by Schmittler, Woop et al. [Schmittler02].*

A special designed graphics hardware optimized to ray tracing seems a more promising approach than the GPU based one. Schmittler et al. [Schmittler02, Schmittler03, Schmittler04b, Woop05] recently came up with a modular and scalable FPGA[1] architecture (SaarCOR) for a ray tracing based graphics board. A running prototype impressively shows the possibilities and there's still room for orders of magnitude of speed using more sophisticates hardware (ASIC) that allows higher clock rates and provides more chip space.

### 2.2.3 Software Based Interactive (Parallel) Ray Tracing

A pure software implementation of a ray tracer is of course the classic way. It is not limited by the narrow resources of specialized hardware and thus allows for complex shading algorithms (*plug-and-play shading*, see Figure 2.3).

Since a single CPU is still too slow to achieve real-time (video) frame rates at a moderate image resolution, pure software implementations need the power of multiple CPUs. There are two alternatives for multi CPU machines: shared memory machines or clusters with per client memory connected by a network. Both ways need slightly different approaches to balance the computation load on the CPUs.

[1]FPGA: Field Programmable Gate Array.

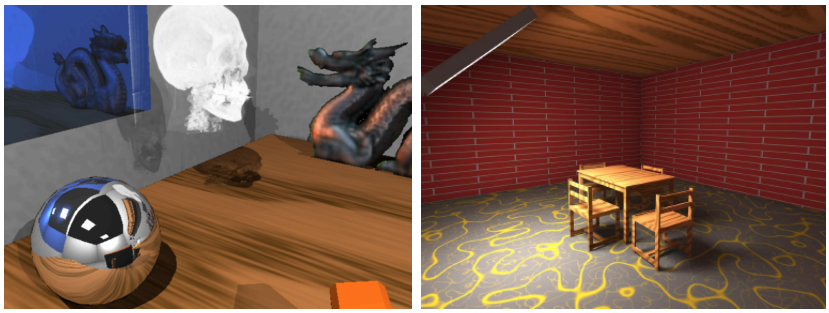


*Figure 2.3: Examples of plug-and-play shading in software ray tracers. The left image shows a number of shader types like a bump mapped mirror on the wall, a procedural wood texture on the desk, a light-field shader (the dragon) or a volumetric shader applied to a box (the skull) The right image shows the possibilities of using special shaders for global illumination algorithms. Those shaders are too complex for implementing them on the limited resources of programmable raster graphics hardware (GPU).*

Load balancing in a parallel ray tracing system is typically done by a screen space subdivision of the output image. Different parts of the frame are scheduled for rendering on the different CPUs. Since the computational effort is not equal for each tile and depends on the visible parts of the scene, a proper load balancing scheme is vital for an interactive parallel ray tracer. This is especially true for the network distributed cluster solution.

The OpenRT system is an example of a distributed, software based interactive ray tracing system.

## 2.3 The OpenRT System

The OpenRT ray tracing system was created at Saarland University by Ingo Wald, Carsten Benthin, and others [Wald01c, Wald04a, Wald02a]. The intention was to provide a framework for network distributed, interactive ray tracing on a cluster of commodity PCs. Figure 2.4 shows a typical hardware setup for an OpenRT based system.

Each rendering frame is subdivided into a regular grid of *tiles*. These tiles are scheduled on the individual clients for rendering. A load balancing mechanism sends job descriptions of the tiles over the network and thus keeps the clients busy on rendering.

Once a client finished a tile job, the resulting sub-image is sent back. A frame is displayed when all tiles are sent back. Of course the server can start

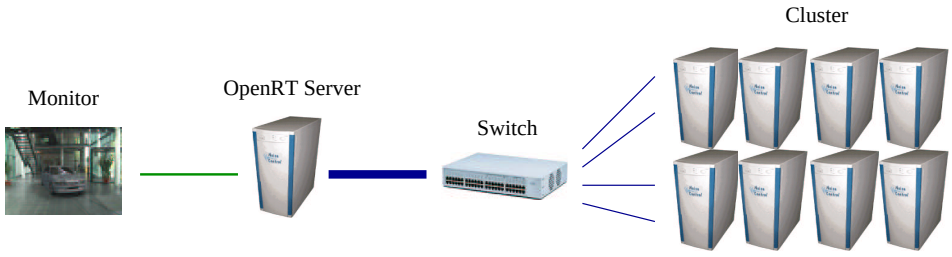



*Figure 2.4: A typical OpenRT based ray tracing system. A number of rendering clients and a rendering server with monitor output are connected to a central switch. To overcome network bottlenecks, the server has a higher bandwidth connection (Gigabit, 1000Mbit/s) compared to the clients (FastEthernet, 100Mbit/s).*

to generate jobs for tiles of the following frame as soon as all jobs of the current frame have been distributed.

### 2.3.1 The OpenRT API

To facilitate programming of new OpenRT applications and porting of existing OpenGL applications, an OpenRT-API, very much in the spirit of the OpenGL API [Woo97], was created [Dietrich03]. Some of the OpenGL concepts do not apply for ray tracing (framebuffer and fragment operations, etc). One important difference with the OpenRT system is that the full geometry needs to be specified in before. Here a mechanism very similar to OpenGL display lists is used. Besides the OpenRT application API, a second API for shader programming is used (see next section).

The OpenRT reference manual [OpenRT] gives the details of the used data structures and functions. An OpenRT tutorial document [Wald03b] provides the necessary introduction in writing OpenRT applications and shaders.

OpenRT currently supports only triangles as geometric primitives. An experimental implementation of *geometry shaders* allows for plugins for arbitrary primitives like free-form surfaces [Benthin04] or volumetric data.

At the time of writing, the OpenRT API is still evolving. For the example MR OpenRT applications in the following chapters, I refer to the current API status at the time when the programming work was done. Newer API versions may provide more efficient ways to performs the necessary tasks.

### 2.3.2 Programmable Shaders

OpenRT features programmable shaders in the spirit of the RenderMan API [Apodaca90]. While RenderMan uses a custom shading language, OpenRT shaders are written as C++ classes based on a generic OpenRT shader class. One or more shaders can be compiled to a *shared object* file, a library that is loaded at run-time when the shader is referenced by the applications.

A shader can have a number of parameters that can be edited at run-time. Local class attributes are bound to parameter names using special functions of the OpenRT shader API. The access of the shader environment data (texture coordinates, normals, etc.) is done on demand using API calls. An OpenRT shader provides a *shade* method that calculates the color and eventually spawns secondary rays. A second method (called *light transparency* in OpenRT) is used for shadow rays allowing to skip the color calculation if only an occlusion information is needed. Utility methods for initialization of local data and *per frame* calculations complete the shader class.

Besides *surface shaders*, OpenRT also provides *lightsource shaders* bound to the lightsources specified in the scene description. In addition to the traditional lightsources like point light or spot light, an *area lightsource* shader facilitates sampling of area lights specified as geometry in the scene. Note that there is no high level support of sampling like the RenderMan `gather` function in OpenRT.

Other examples of OpenRT shaders are *camera shaders* to create primary rays (useful e.g. to provide a virtual camera that is calibrated to match a real camera for mixed reality rendering) and *environment shaders* used for shading calculation of rays that do not intersect the scene geometry (to simulate a distant environment).

Appendix D provides OpenRT example shader. It is used for the video billboard application introduced in Chapter 5 is.

### 2.3.3 The Rendering Object

The OpenRT *rendering object* is a special kind of shader. It controls the communication between the OpenRT server and the clients. The user can specify his own rendering object for an application.

The rendering object is implemented as a C++ class that is instantiated on both, the server and the clients. Parts of the method set are executed only on the server, the other methods on each client. Besides initialization and *per frame* methods, a rendering object can provide a method for subdividing the screen space into tiles and a method called on each clients that contains the rendering loop over the pixels of a tile. Custom data can be added to

the tile data structure by a user pointer and is automatically transmitted to the clients.

### 2.3.4 OpenRT Application Programs

The main OpenRT application used for the examples in this thesis is a VRML97 viewer. It was originally written by Wagner [Wagner02] and is based on the XRML library by Bekaert [Bekaert01]. In recent years the viewer evolved to a commercial product ('inView') sold by inTrace GmbH. The VRML97 viewer is used as an application for all mixed reality application examples implemented with OpenRT in the work on this thesis.

In [Dietrich04b], a scenegraph library, optimized for the use with OpenRT API, is introduced. Conventional scenegraph systems (e.g. SGI Performer or OpenSG) are optimized for complex OpenGL multi-pass rendering methods and the use of OpenGL display lists and therefore not suited for ray tracing systems.

A scenegraph API is the key to user interaction with a scene like (constrained) moving of objects, e.g. opening a hinged car door. The VRML97 based scenegraph from [Dietrich04b] also provides the VRML scripting interface for easy interconnection of OpenRT applications to external programs, like the chess program used for a 3D chess game in Figure 2.5.

*Figure 2.5: Two applications of an OpenRT scenegraph library [Dietrich04b]. The left image shows a desk lamp illuminating a room. A VRML97 animation path [Carey97] is used to move the lamp. A global illumination simulation is updated each frame. In the right image a chess program is connected to a rendered chess board using the VRML97 scripting mechanism.*

## 2.4 Application Examples for Interactive Ray Tracing

Having a modular, interactive ray tracing system like OpenRT, it is worth to explore the space of potential applications. Besides new implementation possibilities of existing application fields, a number of new applications are only possible because of the availability of interactive ray tracing. In the following, the most important applications fields of an interactive ray tracing system are described.

### 2.4.1 Virtual Reality

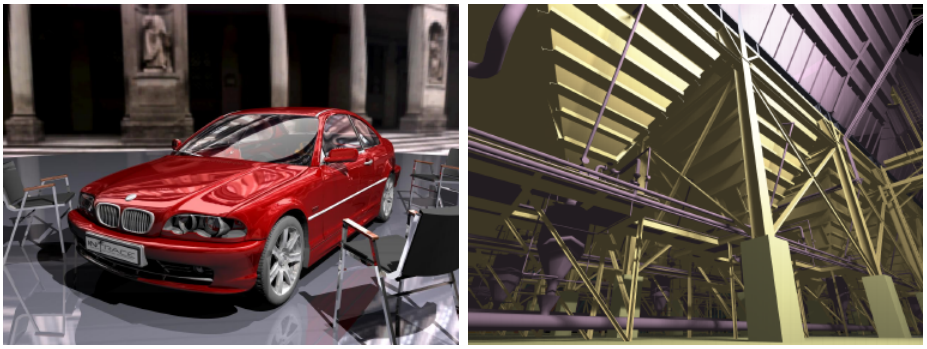


*Figure 2.6: Virtual reality application examples. The left image shows a virtual design study of a car in a photographic environment (design review). The right image shows a detail of a technical model of a power plant.*

Virtual Reality rendering is of course a basic discipline of interactive rendering. Figure 2.6 shows typical applications which benefit from the abilities of a ray tracer to render large, complex models and provide high realism at the same time.

VR systems are usually based on top of a specialized VR scenegraph library. This library controls rendering, allows user interaction, manages movement constraints, provides control over output devices like PowerWall$^{TM}$, Cave$^{TM}$ or Head-Mounted Displays and processes the input from user interaction devices like trackers. Such a library can be build on top of a ray tracer much more simpler than on GPU based systems because of the high modularity of the ray tracing algorithm and because there is no need for multi-pass rendering and thus no influence on the scenegraph system by the

rendering algorithm. At the time of writing, there is no OpenRT based VR system yet.

### 2.4.2 Augmented Reality and Mixed Reality

The high photo-realism available with a ray tracer in combination with appropriate shading algorithms makes it an interesting alternative to raster graphics. The modularity of the ray tracing algorithm allows to include data from 'real world' into the shading process for mixed reality applications.

Since mixed reality rendering based on interactive (software) ray tracing is the topic of this thesis, there will be a number of examples and a closer discussion in the following chapters.

### 2.4.3 Virtual Television Studios (Actor Insertion)

*Figure 2.7: Actor insertion applications. The left image shows two persons inserted into a virtual scene using video billboards (see Chapter 5). The right image shows a 3D visual hull reconstruction of three persons using the method introduced in Chapter 8.*

Virtual television studios allow actors to walk around in a virtual, live rendered scene (*actor insertion*). The high image quality and the possibility to easily add shadows and reflections of the real actors to the virtual scene part, make ray tracing a good candidate for rendering. The hard requirements of live real-time systems for television are still too high for current interactive ray tracer, though. Figure 2.7 shows two methods based on streaming video textures, using different technologies to represent the actors.

The topic of live actor insertion will be discussed in more detail in Chapters 5 and 8 of this thesis.

### 2.4.4 Interactive Global Illumination

Most modern global illumination algorithms are based on ray tracing. With the availability of an interactive ray tracer, one would expect that fast global illumination systems come up automatically. Unfortunately this is not true. Most algorithms need to be adapted to a distributed rendering system because of the lack of global data structures. A carefully redesign can provide impressive results, though (see [Wald04a]).

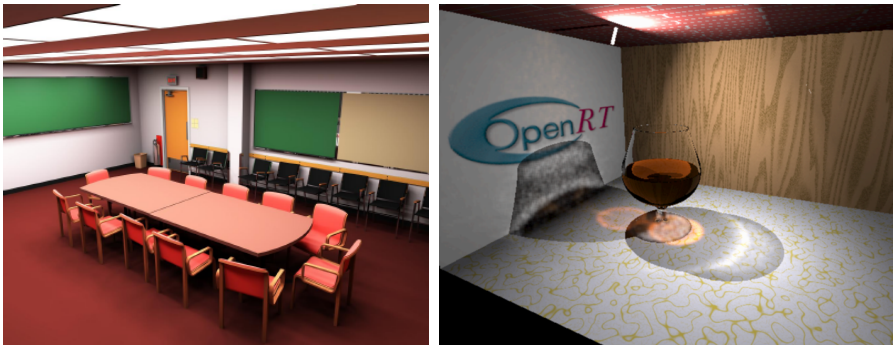



*Figure 2.8: Interactive global illumination examples. The left image shows a conference room lit by linear lightsources on the ceiling. The right image shows a simulation of the caustics projected by a drinking glass. A distributed, real-time photon mapping algorithm was used [Guenther04].*

The 'IGI2 system' build on top of OpenRT can provide global illumination solutions at interactive frame rates [Wald02b, Wald03a, Benthin03] (see Figure 2.8 left) using path tracing an the *instant radiosity* method [Keller97]. An implementation of the photon mapping algorithm [Jensen96] makes it possible to render complex caustics at real-time [Wald04c, Guenther04].

### 2.4.5 Massive Models

The ability of a ray tracer to handle very huge models due to the logarithmic complexity of the intersection operation makes interactive ray tracing the only candidate when it comes to displaying and walking thru models that are too large for processing on current raster graphics hardware [Wald01b]. An efficient, multi-level acceleration structure for the ray/scene intersection can be combined with a virtual memory manager to render even models that do not fit into the main memory completely. The ray tracer only needs to access memory for the primitives visible in the actual view. A pre-process for spatially sorting of the geometry data is needed.


*Figure 2.9: Examples of massive model applications. The left image shows the interactive rendering of a sunflower patch with trees. The whole scene features over one billion ($10^9$) triangles. The right image shows a Virtual Reality walk-thru of a highly detailed Boeing-777 construction model (about. 350 million triangles) (Model courtesy of Boeing Company).*

Figure 2.9 shows examples of interactive rendering scenery with plants and trees and a highly detailed technical model of an airplane [Wald04b, Dietrich04a].

### 2.4.6 Volume Rendering

Of course the ray tracing algorithm is not limited to rigid surface shading. Volumetric shading methods like *ray marching* can be used for atmospheric effects like fog or clouds. *Iso-surface* rendering can be efficiently implemented [Marmitt04] for direct displaying of implicit surfaces in real volumetric data, acquired for example with medical machinery (CT, MR). Figure 2.10 shows some applications.

The modularity of a ray tracer allows to combine this data with conventional (polygonal) data easily in the same scene. Shading on the volume data can be done in the same way as for polygonal data. Thus the whole palette of ray tracing effects like shadows and reflections can be used.

### 2.4.7 Games

The by far most important field of interactive (consumer) computer graphics is the game industry. Interactive ray tracing can provide more realistic looking game scenario. The ability of handling more complex model than raster graphics allows a much higher complexity of scenery and actors. The

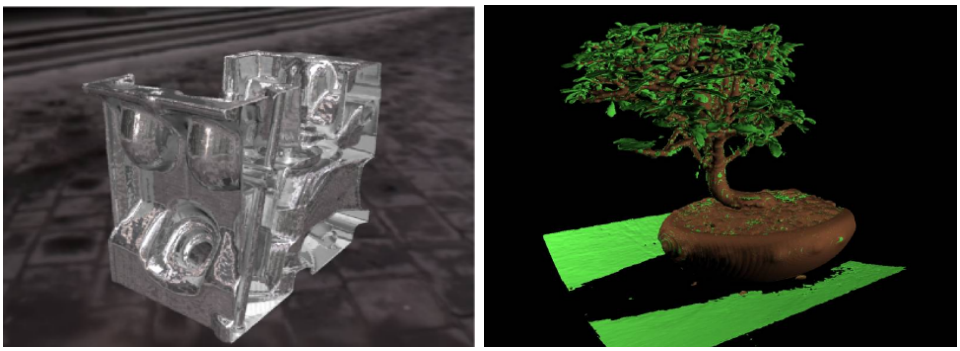


*Figure 2.10: Volume rendering examples with OpenRT. The left image shows a volumetric model of an engine part rendered with iso-surfaces. Ray tracing style shading allows reflections from the environment. The right image shows the rendering of a bonsai tree based on tomography data. Multiple iso-surfaces are shown in different colors according to the density values they represent.*

simple ray intersection test also allows easy implementation of user interaction ('shooting'). Unpleasant programming of complex multi-pass algorithms, like used with today GPUs, completely disappear with ray tracing [Schmittler04a]. Figure 2.11 shows some screenshots of the game 'Oasen' written exclusively for use on top of the OpenRT framework.

*Figure 2.11: Two screenshots of the game 'Oasen' written exclusively for rendering with OpenRT by Tim Dahmen et al. at Saarland University to show the benefits of ray tracing for game engines.*

# Chapter 3

# An Introduction to Mixed Reality Rendering

*Mixed reality* (MR) is the technology of creating applications that supply our perception with both: real and synthetic informations. The visual aspects of mixed reality are typically accomplished by the means of computer graphics. Since the main aspect of MR applications is the *interaction* of the virtual and the real world, interactive (or even better: real-time) methods are needed.

This chapter gives a brief background and the related definitions on mixed reality, necessary for understanding the problems and applications in this thesis. The focus is kept on the rendering aspects. A number of summaries and state-of-the-art reports are available on general AR/MR concepts and philosophy (e.g. [Azuma95, Azuma01, Bimber03b]). The chapter closes with an overview of related techniques for (interactive) mixed reality rendering.

## 3.1 Mixed Reality

Milgram [Milgram94, Ohta99] gives a good definition of what *Mixed Reality* (MR) is. His Reality-Virtuality (RV) continuum (Figure 3.1) shows the possible spectrum of mixed reality applications and defines the basic nomenclature. It spans the whole spectrum from the real world to the (pure) virtual world.

The terms *augmented reality* (AR) and *augmented virtuality* (AV) are used for the areas near the ends of the continuum, for methods where (only few) parts of one world are included in the other world.

Mixed reality should not be seen only in the (rather small) borders of computer graphics rendering. The philosophical aspects go much farther.

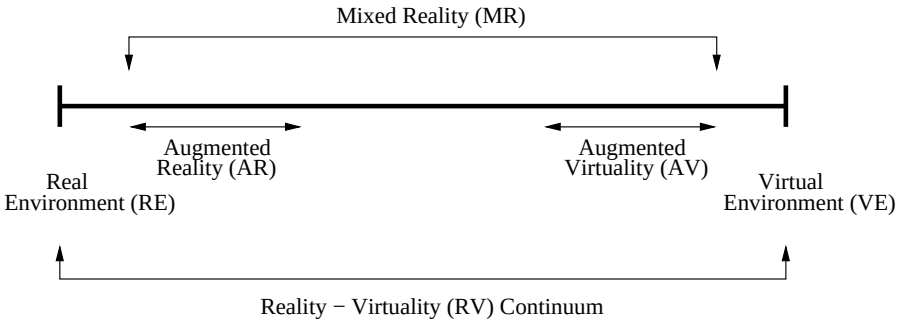



*Figure 3.1: Milgram's Reality-Virtuality continuum. Mixed reality applications span from augmenting real environments with synthetic parts (augmented reality), up to placing real objects or persons into synthetic environments (augmented virtuality).*

Even putting a monitor or video projector in a real world scenario can be seen as some kind of mixed (or augmented) reality.

In this thesis, I will concentrate on the aspects of computer graphics rendering for mixed reality. The aim of this thesis is to prove the applicability of interactive ray tracing to mixed reality rendering by giving a number of example applications from both ends of the Milgram continuum.

### 3.1.1 Augmented Reality

In augmented reality applications, a (real) environment is *augmented* by computer graphics. This can be done with the aid of monitors, displays, or video projectors. Often the *view* of a person is augmented. Here, the virtual part needs to follow the view and must fit into the real scene (*tracked*) in terms of a correct perspective. The virtual part can be just some flat and simple line drawings like arrows or text characters, up to photorealistically objects integrated into the real scene in a seamless manner.

The practical aspects of augmenting a subject's view can be solved by a number of techniques (see [Bimber03b]). The two most common methods are *optical-see-thru*, where the virtual parts and the real view are blended by a semi-transparent mirror, and *video-see-thru*, where the virtual parts are inserted electronically into a video view of the real scene.

A number of additional technologies are related to augmented reality. The are endless applications and methods for tracking, from large scale GPS[1] tracking of persons in wide areas to video-based tracking of gestures for inter-

[1]GPS: Global Positioning System.

action. A survey of applications and technologies can be found for example in [Azuma95, Azuma01].

To simplify the implementation of AR applications, a number of frameworks (*middleware*) specialized for AR purpose are available. For instance ARToolKit from Washington University [ARToolkit] provides easy camera matching and compositing of an OpenGL camera to the (video-based) view camera by the means of simple markers. ARToolKit replaces the common OpenGL GLUT library.

Examples for video-see-thru based AR technology in this thesis can be found in the Chapters 6 and 7. Details on the related AR techniques are give there.

### 3.1.2 Augmented Virtuality

Though the term *augmented virtuality* is rather seldomly used, there's at least one important application in this field: putting real persons into a virtual world and allow them to interact with each other and the virtual objects (*virtual actor insertion*). Possible application fields are (tele) cooperative work (e.g. the *blue-c* system [Gross03]) and *virtual television studios*. The latter allow actors to play in virtual sets and are the real-time counterpart of compositing work in the movies [Kelly00, Brinkmann99].

The aim of integrating a person in a virtual world can be accomplished with a number of methods depending on the desired degree of interaction. Two methods, *video billboards* and *3D reconstruction*, are explored in more detail in Chapters 5 and 8. The related research work, technical background and applications examples can be found there.

## 3.2 Related Rendering Techniques

Rendering in augmented reality applications spans from inserting simple 2D flat shaded objects like text, lines, or arrows, up to full 3D photorealism. The used rendering technique depends on the intended effect on the user and, of course, on the computational resources. Portable computers are, even today, still too slow for complex, realistic rendering. Stationary machines, especially parallel machines like shared memory architectures or clusters of commodity PCs can provide an immense higher computation power and thus high-quality rendering for (pure) virtual and also mixed reality applications.

Traditional rendering methods, based on *local* or *global* illumination models (e.g. [Glassner95, Watt92]), can also be used for MR/AR applications.

Common global illuminations methods are based on the *radiosity* approach [Goral84] or ray tracing based (e.g. (Quasi) Monte Carlo methods [Keller98]).

In the following sections, I will provide an overview over the most important computer graphics techniques related to mixed reality rendering. I will keep the focus on methods that can be used for interactive and real-time applications. A detailed discussion of illuminations methods for MR applications can be found in [Jacobs04].

### 3.2.1 Shadow Generation in MR

A number of publications deal with the (isolated) problem of generating (soft) shadows from virtual object cast onto a real scene. Shadows increase the realism of the composite result, but are only one aspect of consistent illumination in MR/AR scenarios [Slater95]. Convincing shadows require to recover the lighting scheme in the real background scene. This is often done by simple estimates or the lighting is known in advance in a lab setup (e.g. [Marschner97, Sato99b]). Once the illumination is known, real-time shadow methods based on hardware graphics (see e.g. [Moeller99]) can be applied. Often the incident light is represented by a (small) number of point lights. The position of the lights is optimized to fit the shadows to the real shadows present in the background scene. Examples of these *local illumination* methods are [State94, Haller03, Bimber03a].

### 3.2.2 Common Illumination

To augment a scene with virtual objects and their effects (shadows, reflections), often a (simplified) model of the real scene is needed. A number of methods and commercial tools are available for semi-automatic generation of such a model from a number of photographs (see [Jacobs04] for more details). A complete model consists not only of geometry but also material properties and lighting information.

Gibson et al. [Gibson00] describe a common illumination method using shadow maps. A variation followed in [Gibson03b, Gibson03a]. Debevec [Debevec98a] describes a method based on *differential rendering*. The real scene is subdivided into two parts: the local scene with the effect of the virtual objects and an unaffected, distant scene. The local scene needs to be represented by a (rough) model in the rendering process: geometry and light. The latter is acquired by a mirror ball lightprobe. Debevec does not describe the rendering process in detail and uses the Radiance [Ward94] system. [Debevec02a] gives a recipe for *Image-based Lighting* (IBL) with

Radiance. Sato et al. [Sato99a] describe a comparable method, but explain the rendering process in more detail.

Other methods are based on the radiosity [Goral84] approach. Fournier at al. [Fournier93] use a representation of the real scene by boxes textured with images of the objects. Drettakis et al. [Drettakis97a] uses the *hierarchical radiosity* [Hanrahan91] approach with clustering [Smits94, Sillion95]. A *line-space* hierarchy [Drettakis97b] is used for speed-up [Schoeffel99, Pomi99].

For a full discussion of the more than twenty methods and systems introduced in the last ten years, please refer to [Jacobs04].

### 3.2.3 Image-Based Lighting

Image-based lighting (IBL) uses an image instead of a number of specified lightsources for lighting virtual objects. The image is typically correlated with a mapping of spatial directions, e.g. a spherical *lightprobe* image. Spherical lightprobes can be created by photographing a mirror ball. For reproducing the dynamic range of a real scene, a *high dynamic range* (HDR) photograph is taken, typically by fusion of a number of conventional photographs with different exposures (e.g. [Mann94, Debevec97]). A lightprobe image represents the incident light from all direction at one point in a scene. A lightprobe image can be used for rendering by sampling (see next section) or by subdividing the image of the environment into patches and applying the radiosity method [Goral84]. Rendering with image-based lighting methods yields pictures of 'naturally' lit synthetic objects fitting seamlessly into a background photograph of the real scene. This technique is widely used in design and virtual product shots. Image-based lighting is often used in combination with the *common illumination* methods from the last section.

Single lightprobes only represent the incident light at one point in a scene. For complex local effect, like reproduction of a shadow pattern, multiple positions need to be taken into account. Acquisition of *incident lightfields* over planes can be performed with an array of lightprobes of by moving a single lightprobe around to take measures [Unger03].

.

### 3.2.4 Sampling of Incident Lightmaps

Very closely related to IBL are methods to sample maps of incident light (*lightprobes*). The samples are used for rendering with ray tracing based methods. The lightmaps represent a sphere of incident lighting directions. Its radiance values are usually given as an high dynamic range image.

A correct sampling method subdivides the lightmap into different regions represented by a sample position and an average irradiance value. The area of all regions sums to the area of the hemisphere. The size of each region is chosen to obtain the local frequency and importance of an area.

Beside regular sampling methods, *importance sampling* yield irregular sampling distributions related to the contribution of an area to the illumination of a surface point. Optimal sample distributions can be achieved using optimization methods. The LightGen plugin [LightGen] for [HDRShop] generates a (user supplied) number of point lights for a lightprobe. Kollig et al. [Kollig03] use a relaxation procedure based on quadrature rules to optimize sample placement. Agarwal [Agarwal03] uses a hierarchical stratifying method. In [Masselus02], a Voronoi based approach is used. [Szecsi04] gives some background on correlated importance sampling from a Monte Carlo view.

A good sampling distribution can yield smoother soft shadows with less samples and thus better performance in real-time applications. For generating soft shadows usually more samples are required than for lighting a virtual object.

All of these methods are computationally expensive and thus slow. They are not suited for reproduction of fast changing lighting conditions in real-time applications.

### 3.2.5 Relighting Methods

Relighting methods deal with changing the lighting in images (rendered images or photographs) of a scene. This is done by removing the light at the time of an image of the scene was generated and adding a new lighting situation (*image-based relighting*). This method can be used for objects or persons.

One class of methods exploit the *linearity property* of light: a number of photographs of the same scene but under different lighting conditions can be added to simulate the lighting effect that would be achieved with a combined lighting. Debevec et al. [Debevec00] presented a method for relighting of subjects. A (high) number of photographs are taken of a person, each under a single, different lighting direction. The resulting set of images shows the subject lit from many possible directions of incident light. Complex lighting, like from a lightprobe with incident light from an environment, can be achieved by adding all images with the proper weights. The weight for each direction is derived from the corresponding direction in the lightprobe image. If the photographs are taken under white light, the colors from the environment can be reproduced. The same method can be used for arbitrary

(small) objects (e.g. [Masselus02, Masselus03]).

A different class of methods deals with relighting of environments (like a room) and uses similar methods to Section 3.2.2 (e.g. [Loscos99]).

### 3.2.6 Inverse Rendering Methods

Inverse rendering methods try to reverse the classical rendering formula ($model+lighting+camera \rightarrow image$) to obtain a model (geometry and material) or the lighting setup (*inverse lighting* methods) from photographs. Typically, the geometric model (with the matching camera) has to be specified in order to obtain the material properties and/or the lighting. Inverse rendering methods are computationally expensive, not real-time and often unstable. Examples for BRDF recovery are [Yu99, Ramamoorthi01b, Lensch01] and for lighting setup recovery [Sato99b, Marschner97].

Photometric methods for geometry model generation can also be seen as inverse rendering problems (e.g. *shape-from-shading*). A full survey and discussion of inverse rendering methods can be found in [Patow03]. Most methods are closely related to other techniques mentioned in this section (in particular *relighting* methods).

### 3.2.7 Precomputed Radiance Transfer Methods

A method to compress the data in incident lightmaps are *spherical harmonics* (SH, [Ramamoorthi01a]). They provide a basis on a sphere analogue to a Fourier basis. The maximum reproducible frequence depends on the number of used coefficients (*implicit pre-filtering*). Obtaining the (diffuse) incident lighting integrated over the hemisphere according to a given surface normal can be done by just a simple multiplication of the normal vector with the SH coefficients matrix. This makes lighting based on SH ideally for hardware rendering. The method is typically used for lighting single objects (*product shots*).

Precomputed radiance transfer (PRT) methods add an occlusion term to the integration to render self-shadowing and self-interreflection effects. A (ray based) pre-process is used to generate a *radiance transfer function* (RTF) for a rigid object. This function is parameterized over the surface of the object and used in the SH lighting process [Sloan02].

### 3.2.8 Environment Matting

The term *environment matting* (from *environment map* and *alpha matting*, [Zonker99]) is used for methods to acquire the light-transport properties

of 'optically active' elements, typically transparent, refractive and reflective objects (e.g. a magnifying glass). The derived model is often represented as a *lightfield* [Levoy96, Gortler96] (*Image-based* environment matting, [Wexler02]). To aquire the optical properties of an object, *active lighting* methods are used. Often the object is photographed under light patterns generated on a computer monitor. The resulting model for the objects includes all optical effects and can be used for real-time rendering.

# Chapter 4

# Streaming Video Textures

Mixed reality rendering deals with combining parts of the real world with synthetic ones. Since the synthetic parts are typically generated by a computer, it is often easier to include the real world in the rendering process than to do the fusion of both worlds in the real one[1].

For including the real world parts in the rendering process, exact models are needed, including geometry, material description, and lighting. A number of methods of obtaining these parameters from real world objects are available today, but almost none of them can provide a dynamic model at real-time.

One alternative to a model is the use of images (*image-based rendering*). In terms of mixed reality rendering, this means that the real world parts are only represented as photographs or live video. These images are included in the rendering process like traditional *textures*. For interactive rendering with real world parts, a mechanism of including live video is needed. This results in the concept of *video textures*. Video textures can be seen as a basic building block for mixed reality applications.

Please note that the term *video textures* was also introduced by Schödel et al. for pseudo-randomly looped, animated textures [Schoedl00]. In the remainder, I use the term to refer to textures from live video streams.

In this chapter, I will give an overview of streaming video in a distributed rendering system and present a video texture subsystem for the OpenRT framework. A simple example application is given at the end of the chapter and will illustrate the use of video textures in the shading process.

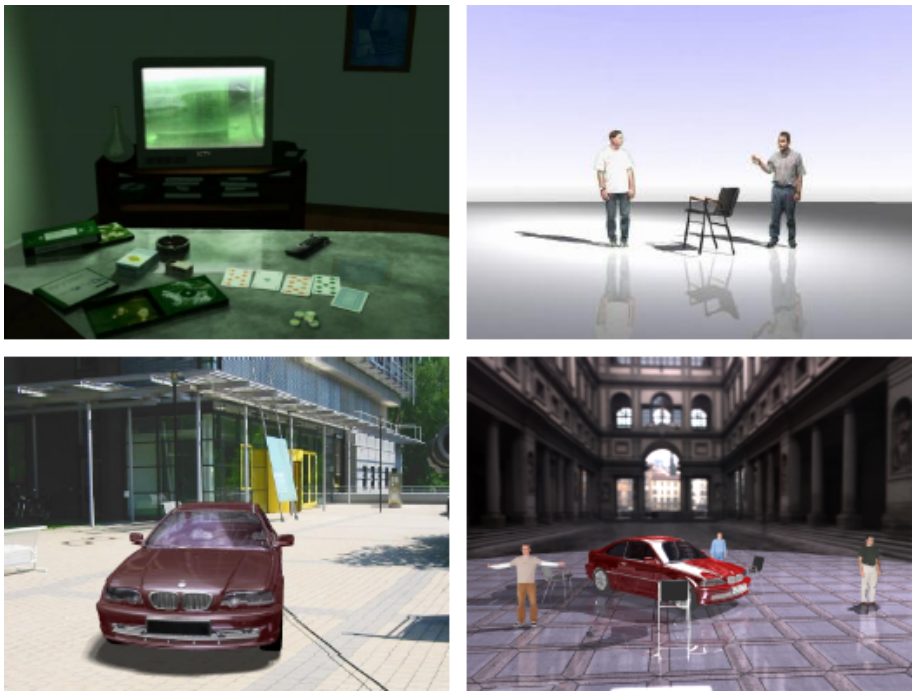


*Figure 4.1: Some application examples for streaming video textures.* **Top left:** *A video texture on a TV set (Section 4.4).* **Top right:** *Video billboards for actor insertion (Section 5.4).* **Bottom left:** *Image-based lighting of a virtual car in a dynamic environment (Section 7.6).* **Bottom right:** *3D visual hull reconstruction of several persons (Chapter 8).*

## 4.1 Video Textures

When it comes to introducing imagery (real photographs or synthetic images) to the rendering process, *texture maps* are the appropriate mean. Texture maps allow to access parts of single pixels (*texels*) of an image. *Texture filtering* methods specify how a texel is derived from the image map, e.g. by linear interpolation of its nearest neighbor texels.

Video textures are pretty much the same as conventional (static) texture maps. The only difference is that the image contents changes with the time. They can be implemented as a (pre-recorded) table of texture maps (e.g. [Schoedl00]).

For an interactive rendering application, the inclusion of *live video* is often desired. Here the rendering system needs to access an external video input

[1]The latter is done in projector-based AR (e.g. [Raskar01]).

(e.g. a frame-grabber board). Figure 4.1 shows a number of applications of video textures from this thesis. In a distributed rendering system (like OpenRT) rendering is performed on a number of clients simultaneously. A method of distributing the video content to the clients is needed (*streaming video textures*). In the following sections, I give an overview of how video data can be distributed to the rendering clients.

## 4.2 Video Data Distribution

To use a video stream in the rendering process, it is necessary to have access to the individual video frames from the shaders. On a shared memory machine a common memory area for all shading threads is easy to implement since the data is read-only for the shaders.

In a distributed rendering system connected by a network, a more sophisticated method is needed. The video data has to be distributed to the rendering clients and a *synchronization mechanism* has to be used. Since the different rendering threads on the clients can render parts of different rendering frames, the video texture system has to take care of using the appropriate video texture frame for the rendering process. In addition, the frame rates of the video textures and the rendering frame rate can differ. The expected video latency in a networked solution is higher than for a shared memory machine. For interactive applications the latency has to be minimal for obvious reasons.

There are a number of methods for distributing video texture stream data in a distributed ray tracing system. In the following, I will give a brief discussion of the most important and practical ones of these methods.

### 4.2.1 OpenRT Payload

OpenRT uses a *tile description* data structure for the communication of the rendering server with the individual rendering clients. This tile data structure can be augmented with arbitrary user data. The OpenRT network layer takes care of sending the data to the clients (see also Section 2.3.3).

This mechanism can be used to send video texture data from the server to the clients (Figure 4.2). The OpenRT client communication uses the TCP protocol [Stevens98] which is reliable, i.e. no video data can get lost on the network. Due to TCP/IP retransmissions there can be jitter, however. This jitter shows up as a 'hickup' effect in the rendering frame rate (*stalling*).

Since it is unknown in before which parts of a texture map will be accessed by a clients, the whole video texture frame has to be sent with each tile job.

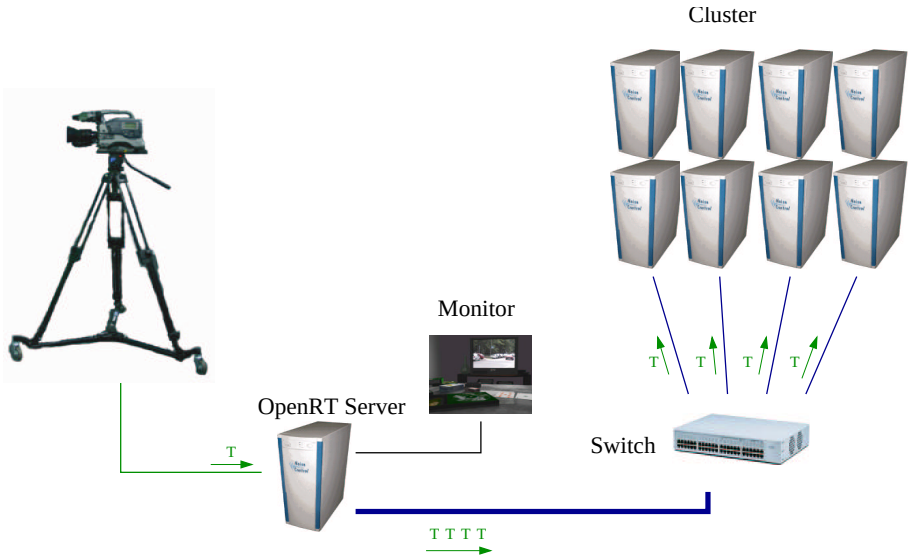



*Figure 4.2: Video texture data distribution via the OpenRT payload mechanism. The OpenRT server also acts as video texture server. A texture frame T (green) is copied with each tile job.*

This causes a bad scaling in the number of clients. The efficiency is low since the clients may only use a small part of the texture or even won't access it at all depending on the assigned rendering tile. There is no need for an explicit synchronization mechanism (see below) with this method.

### 4.2.2 A Demand Driven Approach

Another approach is a *demand-driven* one: when a shader needs to access a frame of a video texture, it asks the video texture server for the data. This method causes two network packets, a request from the client with the desired texture coordinates and a response from the video texture server. A texture filtering can be performed on the server to keep the network load minimal.

Figure 4.3 shows the demand drive approach. The main drawback of this method is the high latency introduced by the request-and-response system. The video texture servers have to process a high number of requests, which can be a problem of network subsystem performance and computing power. A synchronization mechanism is needed since the clients can render parts of different output images and have to specify the desired video texture frame number in the request.

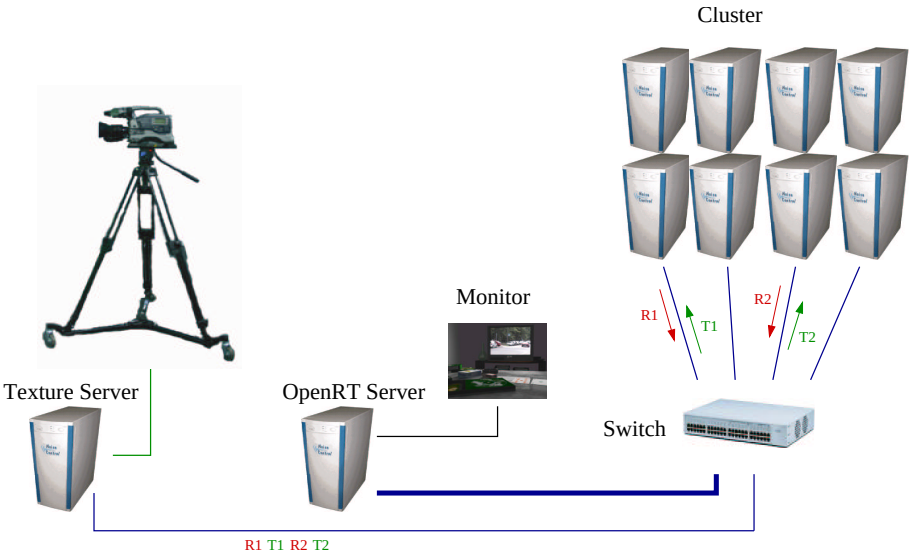



*Figure 4.3: A demand driven video texture approach. When a client PC needs to access a part of the current video frame, it asks the video texture server (request, $R_i$, red). The video texture server responds with a texture (tile) packet (texture, $T_i$, green).*

### 4.2.3 Direct Video Connection

To keep the network free of video data, a separate connection for the video texture data can be used. Figure 4.4 shows a system with an analog video source fed to all clients via a video distribution amplifier. Each client has its own frame-grabber board.

Of course, the hardware effort is immense when considering multiple video texture streams. Also a complex synchronization mechanism is needed. The only way to guarantee correct synchronization is to include timecode in the analog video signal. The SMPTE/EBU[2] VITC (Vertically Integrated Time Code), a time information included into the vertical blanking area of the analog video signal, is the appropriate method. A timecode generator is needed for each video texture source. On the client side the timecode information can be obtained from the frame-grabber in software (*Vertical Blanking Device*). The rendering server has to keep track of the video timecode to provide the synchronization information to the clients.

Even though the hardware effort is high, this is often the only solution

[2]SMPTE: Society of Motion Picture and Television Engineering. EBU: European Broadcast Union.

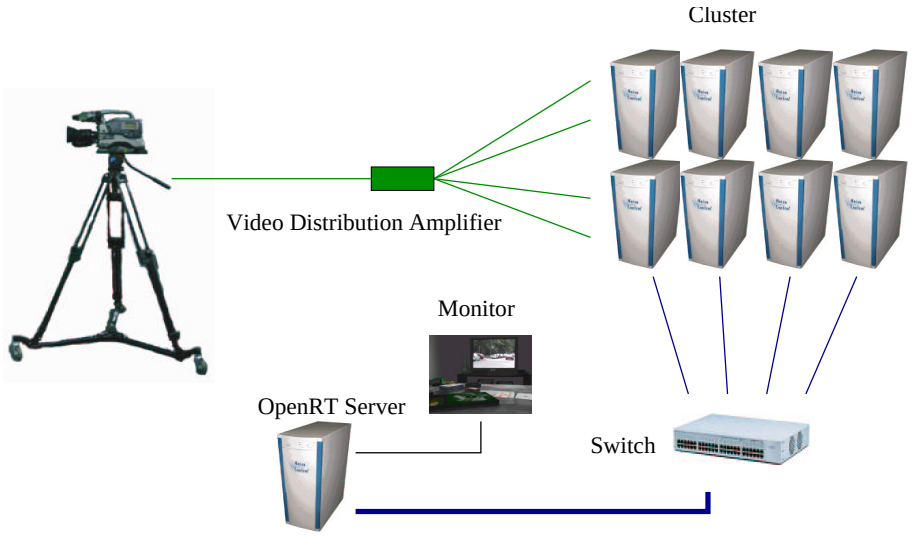



*Figure 4.4: Video texture data distribution via a separate connection. The video source is connected to all client PCs by an analog video distribution amplifier. There's no dedicated video server. The network is not used for video texture data.*

of hard real-time applications. There are alternatives to analog video distribution like uncompressed digital video interfaces (SDI [Poynton03]) or the IEEE1394 bus, which can stream arbitrary data to multiple hosts in an isochronous broadcast mode [IEEE1394, Anderson98]. The IEEE1394 bus can be seen as a special purpose video network.

### 4.2.4 Multicast Networking

Since video streaming in IP[3] is a common problem, a dedicated networking mode was introduced (of course not only for video data): *multicast networking*. A multicast packet sent by a host is automatically transfered to all the hosts that have subscribed for receiving the packet. To keep the network load as low as possible, the network hardware (routers and switches) copies the packet when needed, i.e. when the route forks for different hosts or subnets [Stevens98]. Figure 4.5 shows a multicast video texture system.

IP Multicast uses *groups* rather than IP host addresses. Group addresses are special IP addresses in a reserved address range. By choosing appropriate address groups, the user can restrict the group availability to the local net or up to the whole world. Group addresses are mapped to multicast addresses

[3]IP: Internet Protocol [Stevens98].

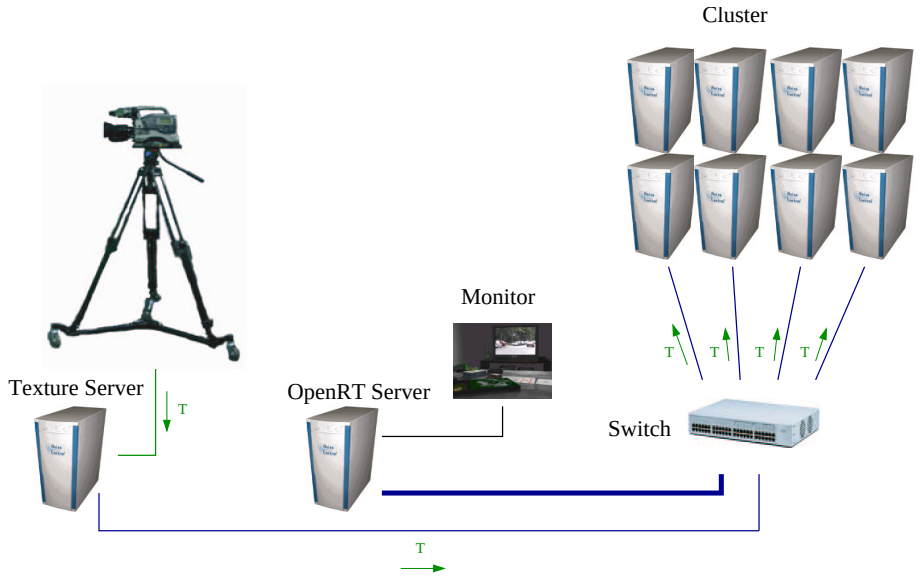



*Figure 4.5: Video texture data distribution via multicast. A video texture server streams the texture frames T (green) to the switch. The switch sends a copy of each packet to the participating client PCs.*

of the underlying media (e.g. Ethernet). Note that Ethernet uses fewer bits for multicast addresses than IPv4, which means a number of IP multicast groups are mapped to the same Ethernet multicast group [Stevens98]. This fact has to be considered when choosing group addresses for minimal network load.

Sending multicast data packets (*datagrams*) is done the same way as for conventional UDP[4] data. The only difference is the assigned IP address, which is one out of the reserved address range for multicast [Stevens98]. Clients can *subscribe* a certain multicast group using special network calls (*socket options*). A number of different multicast address ranges are used to control the propagation of multicast packets over WAN routers. For the video texture application it is desired to keep the packets in the local network, invisible to subscription from the outside world. This also facilitates choosing an appropriate group address.

Since IP multicast is based on UDP, packet transmission is not reliable. A high level protocol should be implemented on top of the plain UDP mechanism to detect packet loss. Note that there are protocols for reliable multicasting (e.g. the Reliable Multicast Protocol RMP [Whetten95]). They use

[4]UDP: User Datagram Protocol.

retransmission of lost packets and are not designed for low latency applications in local networks.

The multicast networking approach for distributing video texture data is used for the OpenRT video texture subsystem described in the following section.

## 4.3 The OpenRT Video Texture Subsystem

The following sections show how the OpenRT framework was extended with a streaming video texture subsystem. The subsystem is implemented as an OpenRT rendering object (see Section 2.3.3). No changes in the OpenRT library were necessary. A *video texture API* provides the shader programmer access the the texture data. The number of available video textures is only limited by the available network bandwidth.

### 4.3.1 The System Architecture

Figure 4.6 gives an overview over the OpenRT video texture subsystem. Multiple video texture servers can send their texture data via multicast to the clients. Each video texture has its unique texture ID. This ID allows the shader programmer to reference a specific texture stream. The ID is assigned manually to the texture server.

A texture server is basically a host running a video texture server applications. Multiple textures can be streamed from the same host. The texture server can obtain the video input from a local (pre-recorded) file or via a Video4Linux [Video4Linux] device. Video4Linux provides input from a number of devices like a frame-grabber board for an analog video camera, USB cameras or a TV tuner.

The video texture server is designed as a C++ class. It can be included in other applications like the Network Integrated Multi Media System Framework NMM [NMM, Lohse02]. Stream data is not limited to texture maps. Arbitrary data structures can be distributed in the same manner as conventional video textures (see Section 7.6 for an example).

### 4.3.2 Synchronization

To ensure that all client tiles of one frame use the same video texture frame independent of which client they are rendered on, a synchronization mechanism is needed. Synchronization is achieved by using *timestamps*.

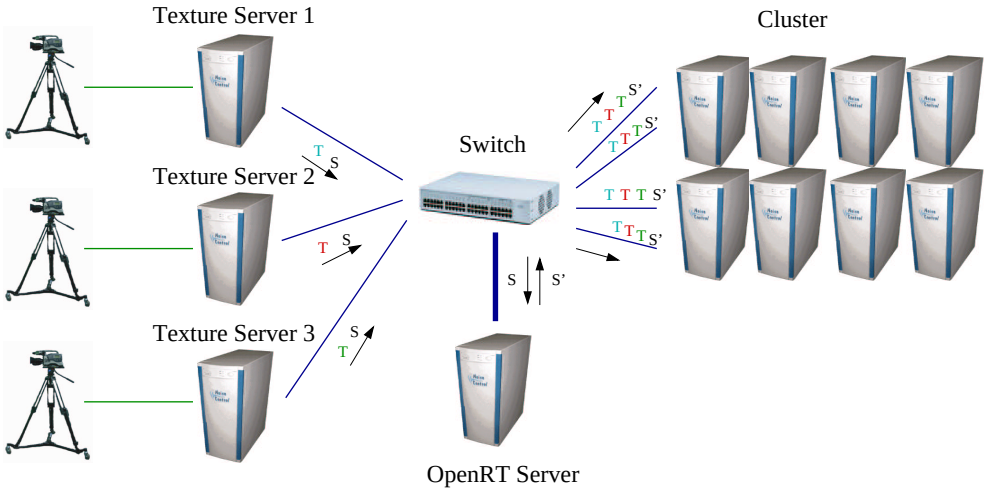



*Figure 4.6: The OpenRT video texture system. Several video texture servers send their data (T) via multicast to all client PCs. Additional synchronization information (S) is send to the OpenRT server on a separate multicast group. The server relays this information to the clients together with the rendering job packets to guarantee that each tile is rendered with the matching video texture frame.*

The synchronization mechanism works as follows: Each video texture server assigns consecutive numbers to the frames of a video texture. The frame data is sent out using a multicast group. The rendering clients and the OpenRT server (i.e. the video texture rendering object) subscribe this multicast group. The clients stores the received frames in an queue. A separate queue is used for each texture ID. Each frame in a queue can be identified by the timestamp.

The server also receives the texture frames but discards the actual texture data. It maintains a table with the last received timestamp for each texture ID. When a new rendering frame is started, the server can look at the table and expect that for each texture the texture frame *preceding* the one identified by the timestamp in the table should be available in the clients texture frame queue.

The server thus sends a table with timestamps for each texture ID with each tile job using the OpenRT payload mechanism. When a texture is accessed in the rendering process of a tile, the appropriate texture frame is chosen by the timestamp from the table. The texture frame queues on the clients have a fixed size (e.g. 10 texture frames).

It is not recommended to stress the network bandwidth of the server

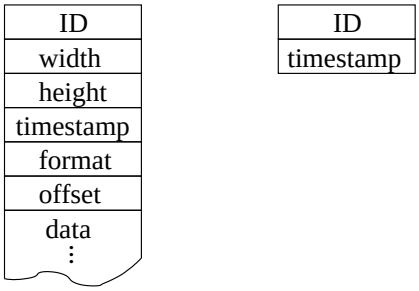



*Figure 4.7: The data structures contained in a video texture network packet.* **Left:** *The packet type used sending the etxture data to the clients. It contains the texture ID, texture image size, the timestamp, the texture data encoding format, the offset of the data in the packet relative to the texture frame and the image data.* **Right:** *The packet type used for synchronization between the texture servers and the OpenRT server (see Section 4.3.2).*

connection with the texture data since here is already a network bottleneck caused by the returned rendering tile data. The OpenRT server needs only the timestamps and not the actual texture data. Instead of subscribing the same packets as the clients, we use a smaller packet type here. It contains only the texture ID and the timestamp (see 4.7). These packets are transmitted on another multicast group that can be subscribed independently from the texture data group. In Figure 4.6 these packets are marked $S$ while the full texture packets are marked $T$.

### 4.3.3 Packetizing

Since the maximum size of a network datagram packet is limited (typ. UDP size is 64 Kbyte), it is necessary to break the data of a texture frame down into smaller packets. Each packet should contain the full texture specification (size and format) since an arbitrary number of packets can get lost (see Section 4.3.5).

Figure 4.7 shows the data structures contained in OpenRT video texture multicast packets. The packet header contains the texture ID, the texture map size, the timestamp of the texture frame the packet belongs to, a texture data encoding format description (see Section 4.3.4), and the offset of the texture data in the packet in relation to the texture frame.

There are several ways to split a texture frame into smaller parts. For instance the tile approach used in OpenRT could be used. We chose to break down the texture by the individual scanlines of the texture image.

Each network packet contains one scanline (or a part of a scanline for large texture sizes). Compared to the tile approach, this method is simpler and needs less memory copy operations.

Other splitting schemes could be considered. For stream based compression like MPEG this packetizing scheme would provide difficulties in synchronization since the MPEG stream cannot simply split into frames and packet loss can influence a number of frames due to the MPEG interpolation scheme. MJPEG[5] is better suited to this approach since it has a frame based structure.

### 4.3.4 Texture Data Formats

Texture images can have very different data representations in terms of color space, dynamic range, compression, etc. To provide a flexible approach for accessing texture data, a modular concept for adding new formats was chosen. Texture formats are identified by a unique, global texture format ID number. Each texture buffer (and each texture network packet) contains a data field with the format ID.

New formats can be implemented by inheriting from a base format C++ class and supplying methods for texture memory buffer access. Typical access methods are texture interpolation schemes like bilinear or box filtering. Access to an additional alpha channel needs separate functions since OpenRT does not support an alpha value in its color primitive. The texture manager on the clients holds a list of formats available and instantiates the appropriate texture format handling class when the first texture packet arrives.

The implemented texture data formats include e.g. RGB(A) or RGB565 (16bit) [FourCC] to save network bandwidth. For more complex applications like the image-based lighting example in Chapter 7, high-dynamic range formats like the 32bit/pixel RGBE format [Ward96] are available.

A *raw format* allows the user direct access to the texture buffers without interpolation. This mechanism is useful for streaming data structures like sample tables (Section 7.6) or for implementing texture compression (Section 8.4.2).

### 4.3.5 Network Packet Loss

Multicast uses a datagram protocol (UDP/IP) with no handshake, hence packet loss can occur. Since all hosts in a OpenRT system are typically

[5]Motion JPEG.

located in one subnetwork and are usually connected to one central switch, the question arises *where* packets can get lost.

Lets have a look at the multicast data flow in the OpenRT videotexture system. The video texture server sends a texture packet via a socket call to the network subsystem. The packet is wrapped in an Ethernet packet and put into the output queue of the appropriate network board. The board sends the packet to a switch, which buffers all incoming packets and sends copies to the connected clients. The packets are received in the client network board buffer and submitted to the network subsystem. If a client socket listens for these specific packets, the client application (the OpenRT video texture manager) receives the packet with a `read` socket call.

Ethernet packets can always be discarded if a buffer on a network board or in a switch overflows. Since we use UDP, the loss of an Ethernet packet containing UDP data causes also the loss of the UDP packet. A UDP packet can be split into several Ethernet packets (*fragmentation* [Stevens98]) since Ethernet packets have a maximum size (MTU[6]). Fragmentation is handled transparently by the network subsystem. If one fragment gets lost, all other fragments belonging to the same UDP packet will be discarded. Since UDP has no handshake, the packet will be lost without notice.

Our experiments with the multicast mechanism for video texture under Linux showed that especially one missing feature of the kernel (or the network board drivers) causes packet loss. There seems no signaling when a network board buffer overflows and the kernel is out of buffer space for a certain socket. The POSIX specification demands to return the `write` call an appropriate error code or to block the sending process, which seems both not to be implemented under Linux for multicast[7]. The same effect occurs with the `select` call which always indicates that the socket is ready for writing.

Sending packets in a higher speed than the network can handle thus causes packet loss. A typical indication for this kind of packet loss is a *burst scheme* of consecutively missing packets. Please refer to [Servetto02] for a more detailed discussion on multicast problems in local area networks.

How can the problem of packet loss be solved? Since the clients can keep a list of all received texture packets of a texture frame, it is easy to determine the missing parts. A client could request missing texture data from the texture server, but this would block the rendering process until the texture server sends the response (see also Section 4.2.2).

Another solution is the reuse of texture data from previous frames. Single missing image map scanlines could also be interpolated from the neighboring

[6]MTU: Maximum Transferable Unit, typ. 1500 bytes for Fast Ethernet.

[7]We used Linux kernel version 2.4.19.

scanlines of the actual frame. This solution could be improved by augmenting the texture data with a certain redundancy, e.g. by using a forward error correction (FEC) mechanism.

### 4.3.6 The OpenRT Video Texture API

To allow the user to access video textures when writing shaders, an API was specified. Calling of a static function of the *video texture manager* object (singleton) is used to obtain a reference for a specific video texture using the texture ID. This reference is used to access texture data. The video texture manager hides the synchronization and networking issues to the user. More details of the video texture API can be found in Appendix C.

## 4.4 A Video Texture Example Application

An example application demonstrates the usage of video textures in the rendering process. Figure 4.8 shows a simple living room scene with a TV set. A video texture is used to display a video image on the TV screen. The texture data is encoded as 16bit RGB (RGB565). Video input is provided by a Video4Linux frame-grabber board. It is connected to a camera outside the lab showing the cars passing down the street.

*Figure 4.8: An example application for OpenRT video texture. A video texture is used on the screen of a TV set in a virtual room. The right image shows the room when the video texture also acts as a lightsource for the room. Note the soft shadows cast by the table's legs.*

### 4.4.1 Lighting from Video Textures

To enhance the realism of the rendered scene, the TV set can be used as a lightsource (Figure 4.8 right). The front screen of a TV set is an area lightsource. We simulate the light emission by a fixed number of point lightsources placed on the screen rectangle. A modified OpenRT point lightsource shader is used to derive the individual lightsource intensities and colors from the video texture.

Each call to the pointlight shader results in averaging a certain part of the video image and returning the appropriate light color. The material shaders for the objects in the scene use the point lights for shading.

The only exception is the TV screen shader which shows only the video texture and ignores the point lights. A cosine term gives the screen a directional light quality. The effect can be seen on the wall (Figure 4.8 right). In the example application the point lights are positioned in a regular grid. Note the soft shadows cast by the TV light.

Since this simulates direct light only, we add a dynamic ambient term. This term is calculated in the shader for the scene objects. A singleton mechanism guarantees that the term is only computed once per frame and client for all instances of the shader. The ambient light is estimated by averaging over the whole video texture (Figure 4.9).

*Figure 4.9: These images show how the light in the room changes with the video contents. In the left image a red car passes in front of the camera. In the right image a green translucent bottle is held in front of the camera.*

Future versions of the IGI2 system [Benthin03] will include direct sampling of textured area lightsources. The IGI2 system must be made aware of that the texture content is changing and a resampling must be performed.

This will allow a full global illumination solution containing also indirect light.

### 4.4.2 Results

Figures 4.8 and 4.9 show examples of the TV setup [Pomi03]. The lighting in the room changes interactively with the video content (Figure 4.9). The video texture is sent with 25fps@320x240 pixel. For the lighting, 9 samples were used. Table 4.1 gives the resulting frame rates with and without lighting.

| Scene | Figure | #CPUs | Resolution | fps |
|---|---|---|---|---|
| TV unlit | 4.8 left | 16 | 320x240 | 20.1 |
| TV unlit | 4.8 left | 6 | 640x480 | 5.3 |
| TV unlit | 4.8 left | 16 | 640x480 | 14.1 |
| TV unlit | 4.8 left | 24 | 640x480 | 18 |
| TV w/lighting | 4.8 right + 4.9 | 6 | 640x480 | 1.8 |
| TV w/lighting | 4.8 right + 4.9 | 16 | 640x480 | 4.4 |
| TV w/lighting | 4.8 right + 4.9 | 24 | 640x480 | 6.7 |

*Table 4.1: Frame rates for the TV example. All measurements were done on Athlon MP 1800+ CPUs. The OpenRT version is limited to about 20fps@640x480 due to network bandwidth.*

When video streaming is stopped, the frame rates increases slightly (about 1–2 fps). This is because the increasing packet loss due to the larger network load causing TCP retransmits in the OpenRT client connections.

We experienced the bust packet loss in the network as occasional dropouts in larger texture areas. The texture lines are sent in ascending order and are not shuffled randomly. The dropouts occur very irregularly in time.

## 4.5 Conclusion and Future Work

The implementation of streaming video textures with a low video latency in a distributed rendering system is hard. The discussed multicast problems under Linux are hard to fix and network system developers unfortunately tend to point out the un-reliability of UDP as an excuse. The related texture dropouts can be disturbing. Multicast is the only practical approach when scalability in the number of clients is needed, however. An *forward error correction* (FEC) scheme can be included for compensating dropouts.

Packet loss problems restricts the multicast approach to non-critical applications. The shared memory version of OpenRT provides dropout free video textures and is thus suited for more critical application fields like virtual television studios. Todays limited availability of (inexpensive) multi-CPU shared memory machines restrict the available rendering power for shading effects, though.

Future improvements of the OpenRT video texture subsystem could comprise an adaptive frame rate feedback from the OpenRT server to the texture servers to adapt the video frame rate to the rendering frame rate. This would result in a minimal necessary network load and thus minimizing packet loss.

Further research could be done on reducing the bandwidth by using a compression scheme for the texture data. Section 8.4.2 gives an example of a fast and simple loss-less compression method for silhouette image data. A multi resolution image set (like a *mipmap* [Watt92]) could be adapted to provide at least a lower resolution image in case of packet loss. Appropriate methods could be found in progressive picture compression schemes.

Other application examples for video textures can be found in the Chapters 5 (Video Billboards), 7 (Image-Based Lighting), and 8 (Visual Hull Reconstruction).

# Chapter 5
# Video Billboards

One frequent task in mixed reality is the insertion of real persons in virtual scenes (*actor insertion*). It can be seen in most of todays action and SciFi movies. Actors are 'cut out' from the background of the filmed action and 'pasted' into a new environment. Care has to be taken to match the lighting conditions to provide a convincing result. In the movies this is done offline, of course. Errors produced by the software, like wrong segmentation, can be corrected manually.

Live compositing of persons in virtual scenes provides a bigger challenge. To isolate the actor from the (studio) background, a real-time segmentation method is needed. Often a blue or green backdrop in combination with chroma keying is used. The next step is an appropriate compositing method. This can be done together with the rendering of the virtual scene or in a post-process. Rendering has to be performed in real-time.

All of todays actor insertion method are based on raster graphics hardware (GPU). Ray tracing can provide a good alternative since environmental effects like reflections and shadows are more easy to generate in comparison to raster graphics methods.

This chapter describes the main application of live actor insertion, virtual television studios, and the related technology. I present an implementation of video billboard for OpenRT using the video textures from Chapter 4. I close with a discussion of the billboard method for actor insertion.

## 5.1 Virtual Television Studios

The idea of a *virtual studio* is to place a subject into a (virtual) studio set in real-time. Building, storing, and maintaining of a real set becomes obsolete. The absence of a post-process makes virtual studios ideal for live broadcast

purpose, although the technology is most time used for recording of daily single presenter shows where a short production time is essential.

The concept of virtual studios originated already in the late 70s. One example is the Magicam [Starlog81] technology. Here a video camera in a blue studio was mechanically tracked and connected to a snorkel camera[1] filming a model. The model camera copied the movements of the studio camera and created a perfect matching background plate. The presenter was keyed in the background electronically. The system was used for documentaries and allowed a presenter to walk in ancient building that were lost with the time like the grand library of Alexandria.

Today, the computer has replaced real models. The rest of the system remained still the same, however. But due to modern computer graphics and broadcast technology like chroma keyers, the quality is much higher today.

The secret of a convincing combined output image is matching camera perspective and providing visual cues like shadows. Ideally both, real scene and virtual model, are lit the same way. Virtual lighting can be done to match the studio light, at least for a static standard TV studio lighting setup.

Camera tracking is used to ensure the right perspective. A number of companies supplies tracking equipment specialized for virtual studios. A complete tracking system often uses a mix of technologies. Camera parameters (zoom, focus, aperture) are often tracked mechanically. Camera pose can be determined by ultrasonic or active optical tracking with additional tracking cameras. [Orad] uses a special blue backdrop with a binary coded pattern and a special image processing hardware to determine all camera parameters from the cameras' video output.

The rendering system used for virtual studios must be highly reliable, at least when used for live purpose. Often, large high end machines (SGI Onyx) are used since they can guarantee a constant frame rate and provide the proper video connections for broadcast environments.

Interlaced, field based [Poynton03] video output is needed. The rendering is performed using OpenGL. Video textures (supported by special high bandwidth hardware in high end systems) allow to insert video content (like news inserts). Depth-of-field simulation aids to match foreground and background view [Wojdala98].

A survey over current virtual television studio technology can be found in [DP97, DP00a, DP00b].

[1] A camera with a periscope featuring a very small mirror at the end.

### 5.1.1 Video Compositing for Virtual Studios

An overview over the hardware setup of a traditional virtual television studio is shown in Figure 5.1. A number of studio cameras are used in a studio equipped with a blue or green backdrop. Hardware chroma keyers provide the compositing of the cameras signal with the rendered background. Camera tracking feeds the rendering servers with the necessary information to match the virtual cameras. Each camera has its own rendering system to provide a composited image on the preview monitors. The director in the control room has the same view of the studio on his monitors like with a real set.

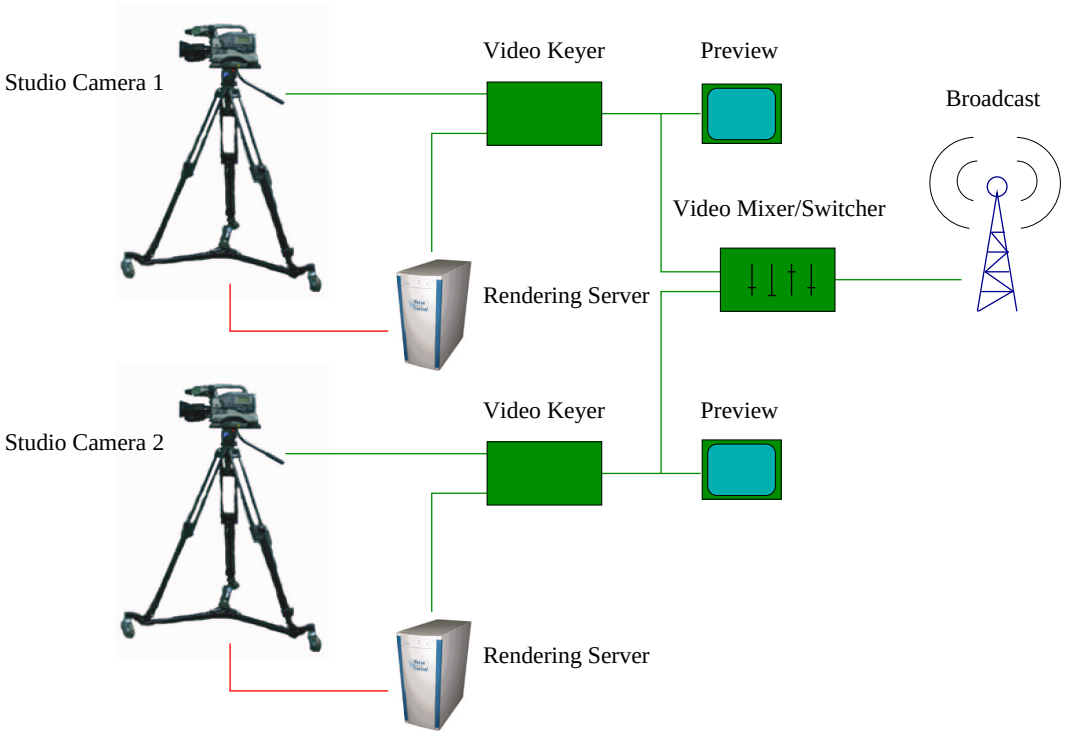



*Figure 5.1: A traditional virtual studio hardware setup with two cameras. Each camera has has its own rendering server to provide a preview of the composited scene. Video (chroma) keyers are used for compositing. The cameras are tracked to provide matching information for the virtual cameras (red). The video switcher decides which camera goes on air.*

Exact real-time generation of the virtual set is not possible. Camera tracking and rendering create a certain video latency of several frames. The video keyer features a delay line for delaying the camera signal for better matching with the rendered video.

This concept can be arbitrary scaled, e.g. down to the simple setup seen at the TV news every day. Here often static camera (without tracking) and a simple rendered background wall with a video insert for the news trailers rendered in 2D are used. Video processing technology remains the same, though.

Keying is performed with special keyer hardware. A keyer is a device that mixes two video signals according to the content of a third signal (*key signal*). The latter is usually supplied in monochrome. The luminance of the key signal controls the mixing (*luma keyer*). Chroma keying can be done by adding a circuit that converts the color difference to a key signal [Keith96, Poynton03]. The key signal is comparable to the *alpha channel* in computer graphic image processing.

An alternative to the traditional compositing method is to do the compositing on the rendering servers, for example as part of the rendering process.

### 5.1.2 Consistent Lighting

Consistent lighting is essential when a convincing result is expected. This means not only matching of virtual and studio light but also providing visual cues like shadows cast by the actor or the reflection of the actor on a shiny floor.

The usual way to obtain a shadow of the actor on a floor is to key out the real shadow in the studio together with the actor. High end keying systems are designed to provide keying with respect to shadows [Ultimatte].

This approach does not work when the shadow is cast onto a virtual object different from the floor plane, e.g. a table. Here we expect the shadow to adapt to the contours of the table which is impossible with the keyed shadow. A blue stand-in object for the table can help (or using a real table keyed out together with the shadow).

Another approach uses a *virtual* shadow of the actor. This shadow is implicitly cast on virtual objects in the right way. Since the actor has no representation in the virtual scene, casting a shadow of him is difficult.

A solution is using an auxiliary camera (*shadow camera*, [Wojdala00]). This camera is mounted near the shadow casting lightsource in the studio and provides a perspective view of the actors silhouette as seen from the lightsource. The video image of this silhouette can be used as shadow map in the rendering process. A projective texture map, projected from the position representing the shadow camera in the virtual scene, generates a shadow on the floor and other objects. Since the real actor is not present in the virtual

scene, there is no depth test necessary as with conventional shadow maps [Moeller99].

Reflections of the actor, e.g. on a shiny floor, can also achieved by keying. The studio floor is just made reflective for this purpose. A sheet of plexiglass is often used. More complex reflections need to be done in rendering. Again there's a problem since the actor is not represented in the virtual scene. Sometimes, simple floor reflections are simulated by inserting a mirrored video image. Due to the missing perspective information, the insert position needs to be tracked with the actor [DP00a].

Matching of the lighting conditions in the studio and in the virtual scene are done manually. It is difficult to create a dynamic scene with changing lighting conditions (e.g. a fire) and have the real scene automatically matched. Debevec et al. provide some ideas how this can be done (Lightstage 3, [Debevec02b]). Even when lighting is matched, something more is needed to provide a convincing result: a color calibration of real video and rendering (e.g. [Wenger03]).

Consistent interaction of the actor with the virtual scene is also important. Often real props and furniture are used in the studio and keyed together with the actor. When virtual furniture is desired, often blue painted stand-in objects are placed in the studio. For instance the presenter can stand behind a shiny glass table reflecting the virtual scene while in the studio there is only a blue board on a blue stand. Here rendering of the table is the only solution, a real table would reflect the blue studio instead of the virtual scene and also cause chroma keying problems.

Often convincing interaction involves occlusion of virtual objects by the actor. Since the compositing is done completely in 2D, there is no depth information. Think of an actor walking behind a row of columns. There are two solutions: building blue stand-in columns (which can be simple boards) or forcing the keyer to place the virtual columns always in the foreground (e.g. by using a *garbage matte*, Section 5.2.1). The latter can only be accomplished when the renderer can control the keying process by the means of an additional signal (alpha channel). Usually the keyer has a dedicated input for a special monochrome video signal that is combined logically with the key signal from the chroma circuit. Rendering machines aimed on virtual studios (SGI Onyx) can provide such signals.

Special cameras designed for virtual studios try to overcome the problem of missing depth information in the compositing process (e.g. Z-Cam by [3DV]). They can provide a (rough) depth measurement per pixel (*depth map*). This map can be used in the rendering process to control the keying, e.g. to 'switch' objects to foreground or background as an actor walks around of them.

Note that though the problem of consistent lighting is discussed for actors here, the same principles apply for arbitrary objects in MR compositing work.

## 5.2 Foreground Segmentation

To insert people (or objects) shot in a studio into a virtual scene (or real footage) there's the need to *cut out* the people from the background of the shot. A segmentation method for segmenting an image (or video frame) into foreground and background pixels is needed.

This *foreground segmentation problem* can be stated as follows: A foreground segmentation algorithm has to state for each pixel in an image whether the pixel belongs to a foreground object or to the background. The output of the algorithm is a binary picture called a *matte*. There is no algorithm that can solve this problem for arbitrary pictures and thus this is a fundamental problem of computer-based *image understanding* research. But often there is a solution if some constraints are introduced, like a fixed background or a disjunct color scheme for foreground and background. Sometimes this solution can be accomplished in real-time.

There are several methods for computing a matte under practical conditions. In the following I will explain some of them in more detail.

### 5.2.1 Garbage Matte

A *garbage matte* is a (often coarse) manually created mask for segmentation [Kelly00, Brinkmann99]. Garbage mattes are used in combination with the automatic methods described below. The user manually indicates image areas as strict background. These are typically areas for which automatic recognition fails, e.g. where no color backdrop can be applied or equipment like lighting fixtures need to be placed. A garbage matte can also be used to lower the computing power needed for automatic segmentation by restricting the necessary calculation to those areas outside the garbage matte.

### 5.2.2 Chroma Keying

Petro Vlahos was the first who patented a method for foreground segmentation based on colors [Vlahos78]. Segmentation is performed by putting a subject in front of a background with a unique color not present in the subjects color palette. In this early days of *chroma keying*, the process was difficult and done in the lab using a color separation process. A black and white film with the matte was the result.

With the availability of digital processing, more elaborated chroma keying methods were invented. Today, the Ultimatte Company [Ultimatte], holding still some of the Vlahos patents, is the most famous provider of chroma keying technology available as ready-to-use hardware or software plugins. The highly elaborated Ultimatte method uses more than thirty parameters to adapt the process to the actual needs. It is even possible to segment glassware or smoke. [Smith96] describes the theory of the chroma keying process.

Chroma keying can be done using arbitrary colors, but the spectrum of the key color should not overlap the spectra of the foreground. Due to the color of caucasian skin, green or blue are used for work with actors. Red is sometimes used for model shots. Since the human eye is less sensitive to blue, video compression algorithms tend to use less resolution for the blue channel and a green backdrop is often a better choice.

One problem of the chroma keying method is *color spilling* from the background. This effect occurs when colored light from the backdrop hits the foreground subject and causes parts of it to reflect the background color. Color spill leads to wrong segmentation results.

Another problem is the high level of light necessary for the background. To provide a good segmentation, the color needs to be as pure and uniform as possible. Shadows cast by the actor onto the background can cause problems. Drama lighting with a dark foreground (*low key* [Viera93]) is difficult to achieve.

*Figure 5.2: The Lighting device for the BBC Truematte technology. A ring of colored LEDs on the camera lens in combination with a highly retro-reflective backdrop fabric provides a nearly perfect chroma keying method.*

Addressing those problems, BBC R&D [BBC RD] came up with an interesting solution. The BBC *Truematte* technology uses a highly retro-reflective fabric (comparable to the ScotchLite$^{TM}$ material used in traffic signs) for the background. Under normal white light, it just appears grey. A special lighting device (Figure 5.2) is used on the camera. A number of green or blue

high efficiency LEDs[2] illuminate the backdrop and cause the camera to see a evenly distributed color. Due to the qualities of the fabric, only a low light power is necessary. No spill is cast. Light from other directions than the camera axis has no influence. This technology is available under several trademarks (e.g. HoloSet, Chromaflex).

Chroma keying algorithms are based on the *color distance* of foreground and background color. Color models like HLS[3], HSV[4] of YUV are better suited than the RGB model.

[Bergh99] describes a simple method used for real-time segmentation in VR applications. The chroma keying method used for the billboard example in Section 5.4 is based on their method. Please see Section 5.4.4 for a more detailed description.

A number of other methods, for example based on Baysian networks [Chuang01, Chuang02] have been published. Most of them use an optimization process and are not suitable to real-time requirements. In [Peters03], a combination of chroma keying and background subtraction is used. A camera noise filtering process ensures proper operation under difficult lighting conditions.

### 5.2.3 Invisible Keying

Another approach to provide a proper matte is to use an *invisible* keying. This can be accomplished e.g. with infrared (IR), ultraviolet (UV) or polarized light [Ben-Ezra00].

In [Debevec02c], a practical approach is described to perform segmentation for the *Lightstage 3* setup. An infrared keying system is used (Figure 5.3). Two cameras – one equipped with an infrared stopping filter and on with a filter that only passes infrared light – are connected via a beam-splitter (see Figure 7.7) to a common lens. The background needs to have a high infrared reflectivity and is lit by IR LEDs. The output from the color camera is used as foreground signal, while the second cameras output (which can be a monochrome camera) provides a (binary) key signal.

Note that the invisible key technique overcomes the problem of color spilling on the foreground since no highly saturated colors are needed. A post-processing for spill removal is hence not necessary. The method is also useful when the foreground is near the background and foreground lighting spills onto the background. Instead the reflective backdrop a translucent one can be used in combination with back-lighting.

[2]LED: Light Emitting Diode.
[3]HLS: Hue, Lightness, Saturation.
[4]HSV: Hue, Saturation, Value.

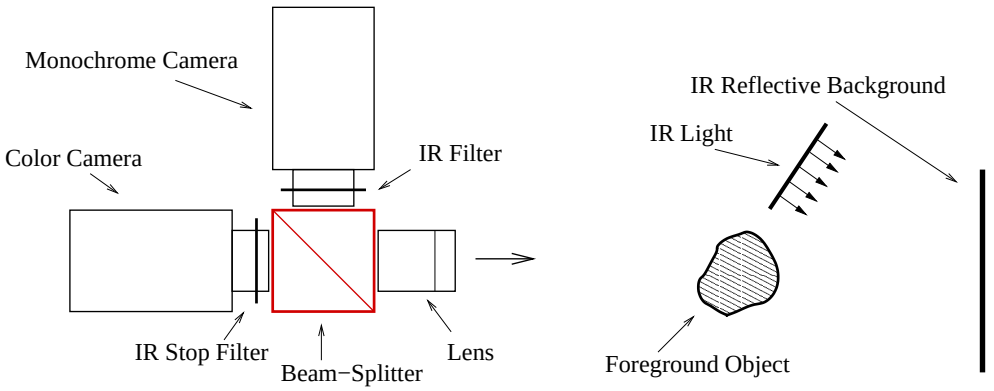



*Figure 5.3: Principle of invisible keying with infrared light. The object is photographed in front of a background lit with infrared light. Two cameras are used in combination with a beam-splitter. One (color) camera has a infrared stopping filter while the second one (monochrome) has a filter that stops all light except infrared. The latter camera provides a key signal.*

### 5.2.4 Background Subtraction

*Background subtraction* methods need no special (colored) backdrops or other dedicated hardware. The only necessary prerequisite is that the background colors clearly distinguish from the foreground objects. This makes those methods attractive for complex (multi-) camera setups where a backdrop is difficult to set up.

A simple background subtraction method works as follows: The camera takes an image (a video frame) $I_0$ of the background without any foreground objects. For further frames $I_i$, including the objects, a foreground mask can be computed from

$$D_i = |I_i - I_0| \tag{5.1}$$

where '−' is an appropriate color difference operator (hence the name *background subtraction*). High pixel values in the (monochrome) color difference image $D_i$ indicate (possible) foreground pixels. A comparison with a threshold $T$ provides a matte image:

$$matte_{pixel} = \begin{cases} foreground & if \quad D_{pixel} > T \\ background & if \quad D_{pixel} \leq T \end{cases} \tag{5.2}$$

The color difference can be computed in RGB space but other models like HLS or HSV seem to be more suitable for this purpose.

One implicit assumption such a simple method makes, is that changes in the foreground do not affect the background. In reality this is often difficult to accomplish, e.g. the shadow of a person on a floor can cause the shadowed floor part be accidentally segmented as foreground. In [Matusik00] a background subtraction method is used for 3D reconstruction of an actor (see also Chapter 8) and causes an artefact in the silhouette due to a hard shadow. Careful lighting can overcome this problem.

Another problem arises when using noisy video cameras (and most inexpensive cameras are noisy), the noise in the frames can cause color differences in the background and hence wrong segmentation. A (software) noise filtering can help [Peters03].

The color difference can be calculated in real-time. Intels [OpenCV] library provides functions for a simple background subtraction method, but most *real world* applications need more robust methods.

*Optical flow* methods (e.g. [Chuang02]) provide a more complex attempt for the background subtraction problem. With robust real-time computation of optical flow becoming available (e.g. [Bruhn03]), these algorithms can be easily adapted for foreground segmentation. One assumption is needed indeed: the objects or persons in the scene need to move at least once. If a person passes in front of a fixed foreground object the object will be lost in the matte when it reappears behind the person. A combination of optical flow methods and color differencing heuristics can avoid those problems.

## 5.3 Video Billboards

One of the drawbacks of direct video compositing is the one-to-one relation between the two images: the frame line of the insert image must match the frame line of the background. If you think of a long shot of an actor, this can cause problems when the studio is too small. With digital video effects units (DVE) available, this problem diminishes. DVE units allow to scale and place the segmented video image of the actor arbitrarily in the video frame. But there remains still a problem if, for example, the virtual camera moves around in the scene and the real camera can not copy the movement due to studio space restriction. This compositing problem cannot be accomplished in pure 2D post-processing.

An alternative to 2D compositing is to represent the actor in the 3D scene by a model and to perform compositing implicitly in the rendering process. Since a complete actor model in 3D acquired in real-time is hard to obtain (see also Chapter 8), *billboards* with the video insert provide an alternative.

A billboard is a (flat) rectangle, textured with the video insert image

of the actor. The billboard is positioned with its surface normal pointing always towards the camera. Billboards can be found in computer games and VR applications when full geometry would need too much rendering power. When the billboards texture has an alpha channel, parts of the billboard can be transparent.

A video billboard can be arbitrary scaled and placed in the virtual scene. It can even move around in the scene, e.g. when the actor stands on a virtual vehicle or if he pretends to walk along a huge scene where the real studio space is too small and the actor is walking on the spot.

Billboards can be implemented simply in raster graphics. The OpenGL *alpha test* enables conditional rendering based on the alpha channel information. OpenGL *blending* allows mixing of foreground and background using the alpha value. Both features can be used for billboards alternatively. Shadows cast from a billboard can be created by using projective texturing together with the billboard texture. Reflections can be achieved using multi-pass rendering [Moeller99].

Software ray tracing on the other hand provides full programmable shading and thus allows to do demand-driven segmentation in the shader. Blending with the background is done by tracing secondary (transparency) rays for the transparent part of the billboard. Shadows and reflections come up automatically and do not need special shaders on the scenes objects. There's no need for the scene shaders to access the billboard texture. Ray tracing even allows the use of refractive objects in front of the billboard to distort the view which is hard to do with raster graphics.

Compared to traditional compositing, the billboard methods can provide better solutions when depth and occlusion is important. Though the billboard object yields a (rather) constant depth value (distance to the camera), it is possible to place a billboard in a scene to allow the actor to move around between virtual objects.

### 5.3.1 The Concept of In-Shader Compositing

Billboards can be seen as an application of an *in-shader compositing* method. In-shader compositing means that the compositing process is performed together with the shading process in one single step (Figure 5.4).

For billboards — or for using textures in general — this seems obvious, but compared to the traditional compositing method (Section 5.1.1) in-shader methods have some benefits: Since rendering and compositing merge into one step, the compositing process can make use of all the information that is available in the shading process. This is especially important for 3D related information like depth which is needed for correct rendering of occlusion.

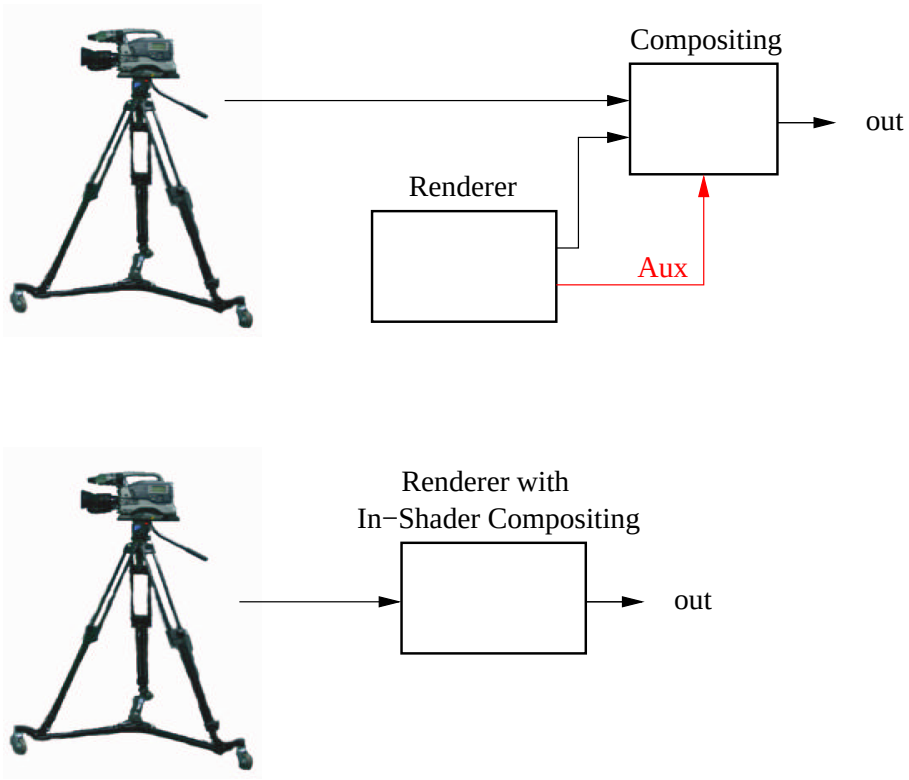



*Figure 5.4: Traditional compositing (top) vs. in-shader compositing (bottom) in a virtual studio application. The in-shader concept integrates rendering and compositing into one step. There is no need to carry auxiliary information (red) from the renderer to the compositing step.*

With traditional compositing it is necessary to carry this information from the rendering step to the compositing step (e.g. depth or alpha channel). A separate compositing step also makes a system more complex.

For in-shader compositing in combination with ray tracing there's another benefit: the demand-driven concept of a ray tracer leads automatically to a demand-driven compositing step, i.e. the necessary calculations for the compositing step are only performed when needed. For the billboard example the means demand-driven background segmentation for instance.

Other applications of the concept of in-shader compositing are described in Section 6.2 (AR view compositing) and Chapter 8 (3D visual hull compositing).

## 5.4 An OpenRT Video Billboard Example

To illustrate the possibilities of video billboards, I provide a small OpenRT application example. One or more subjects are inserted into a virtual background using rectangles with video textures.

Chroma keying and compositing is done inside the billboard shader. Ray tracing effects like shadows and reflections come up nearly automatic. It is even possible to render a person thru a refractive glass object [Pomi03].

### 5.4.1 Hardware Setup

Figure 5.5 shows the necessary hardware setup for the video billboard example. The subject is photographed by a video camera. An evenly lit greenscreen is used as backdrop. The camera signal is fed to a PC acting as video texture server (see Chapter 4).

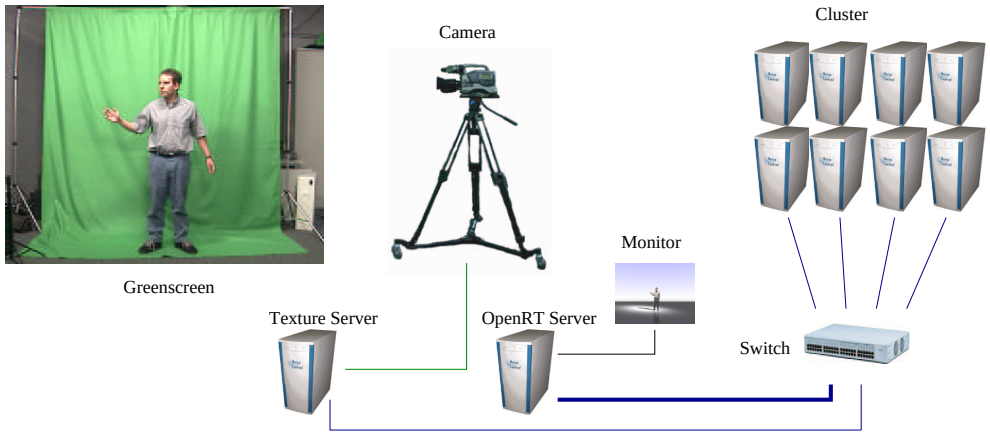



*Figure 5.5: The hardware setup for the video billboard example. A person is photographed by the video camera in front of a greenscreen. The camera output signal is fed to a video texture server equipped with a frame-grabber board. The OpenRT server PC is connected via a Gigabit network to a central switch and outputs the final rendering on a monitor. The video texture server and the cluster PCs are connected via 100Mbit/s FastEthernet.*

Care has to be taken for the camera setup. All automatic features like white balance and auto exposure have to be switched off. White balance must be performed with the camera aimed fully at a white target (a piece of paper for example) to avoid confusion of the white balance circuit by the green background. Wrong white balance yields a video image with a low

saturated greenscreen and wrong subject colors. Auto exposure in a video camera is typically done by averaging the video signal output over the whole chip area. The auto exposure circuit controls the lens aperture with a motor (on professional cameras like the JVC GY-DV500 used in the example). If auto exposure would be switched on, the aperture would constantly change when the subject walks in front of the greenscreen. A proper aperture setting is needed to get a saturated green background color and to expose the subject right. Deep black areas in the subjects clothing can cause problems with a simple chroma keying algorithm. The subject should not wear any green for obvious reasons.

### 5.4.2 OpenRT Setup

The necessary OpenRT setup just comprises the video texture feature from Chapter 4. An appropriate rendering object is used to provide video texture support. The scene contains two billboard rectangles (Figure 5.6 right). No compression was used for the video textures in this example (for an suitable compression method please see Section 8.4.2).

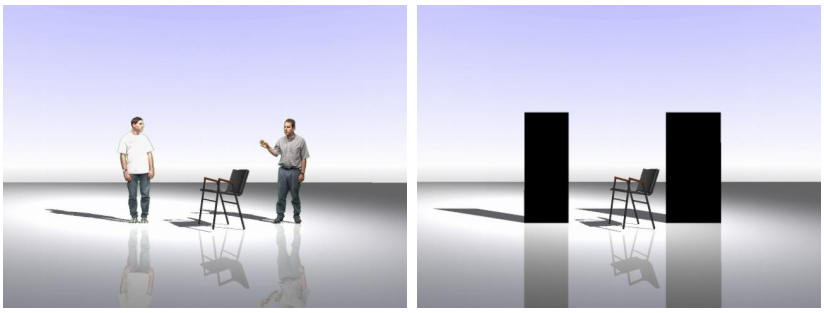


*Figure 5.6: A video billboard application. The left image shows the final rendering output with two persons inserted separately into a virtual background scene. The right image shows the billboard rectangles in the scene description.*

A special billboard shader is assigned to the rectangles, each instance receiving a different texture ID. The source for one billboard was the live camera while the other shows a pre-recorded video. Chroma keying is performed in the billboard shader.

### 5.4.3 A Billboard Shader

Figure 5.7 shows the principle of the billboard shader assigned to the rectangles. If a ray hits the rectangle, the video texture is accessed at the position

determined by the interpolated texture coordinates assigned to the rectangles vertices. The chroma keying test (see next Section) is performed to determine whether the hit point belongs to the silhouette of the subject or not.

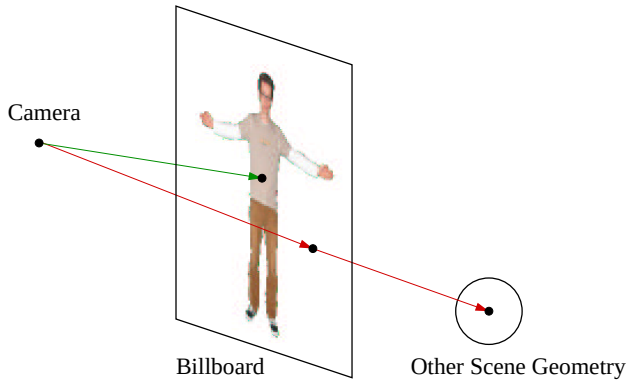



*Figure 5.7: Principle of a billboard shader. Where a ray hits the silhouette (green), the color value of the texture is returned. When the silhouette is missed, a transparency ray (red) is spawned and intersected with the scene. This also works for shadow rays.*

Where a point inside the silhouette is hit, the shader returns the color from the video texture (foreground). If the silhouette is missed, a transparency ray is generated and traced from the hit point into the scene in the same direction as the incident ray. If this secondary ray hits anything, the color is returned (background). OpenRT features a separate shading call for shadow rays. Here, no color calculation is necessary. The billboard shader just performs the chroma keying and returns a binary flag for transparency of opacity.

The complete C++ source code of the billboard shader can be found as an OpenRT shader example in Appendix D of this thesis.

### 5.4.4 Chroma Keying

Chroma keying is performed on demand inside the billboard shader. In the following, I describe the *principal component* chroma keying method used in this example. The method is based on finding the principle component in the input color vector and ensuring a certain distance to the other components. For the greenscreen used in the example setup, the green component in the video texture image needs to be examined.

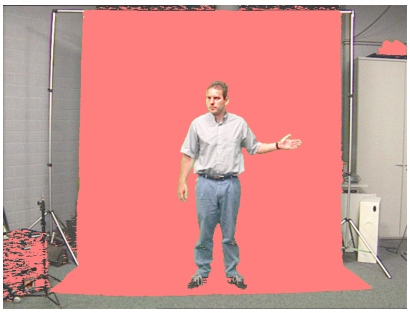
a)  b) 

*Figure 5.8: Fast chroma keying using the principal component method.* **a)** *The input image shows the author in front of a greenscreen.* **b)** *The output of the chroma keying method. The background parts are colored in red. Some parts from the hair, the shoes and on the knees are missing. The right arm shows color spilling from the backdrop. The used chroma scaling factor was* 0.8.

The according (binary) segmentation function for the color value of a pixel is described as

$$Seg(pixel) = \begin{cases} background & if \quad pixel.green > pixel.red \quad and \\ & \quad pixel.green > pixel.blue \\ foreground & otherwise \end{cases} \tag{5.3}$$

Due the to minimal necessary distance of components to trigger the function, it acts rather unstable. A *scaling factor* for the green component provides the means of adjusting the green level to the scene needs. With the scaling factor $s$, Equation 5.3 yields

$$Seg_s(pixel) = \begin{cases} background & if \quad s \cdot pixel.green > pixel.red \quad and \\ & \quad s \cdot pixel.green > pixel.blue \\ foreground & otherwise \end{cases} \tag{5.4}$$

Problems still remain for dark area in the input signal. The video circuit clips dark greens (e.g. in the shadows cast on the green background) to black. The color information is lost and Equation 5.4 defines black as foreground.

Figure 5.8 shows the resulting images when using Equation 5.4 for chroma keying. The video input is given in Figure 5.8a. Figure 5.8b is the output image. The part segmented by Equation 5.4 as background is colored in red. Problems are caused by green color spilling (diffuse and glare reflections) on the subject.

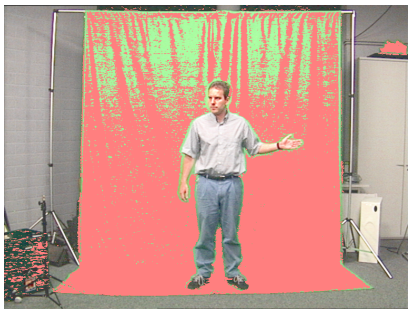
a)  b) 

*Figure 5.9: Choosing of an appropriate scaling factor is important.* **a)** *A too low scaling factor can result in too few of the background being segmented. Problems by shadows on the wrinkles of the green fabric are clearly visible (scaling factor* 0.65*).* **b)** *Using a too high scaling factor causes loosing of the 'problem zones' like dark parts of spill zones. Here a factor of* 1.0 *was used.*

Choosing of an appropriate color scaling factor is very important. Figure 5.9 shows the result with factors higher and lower than the ideal factor for the scene (about 0.8). Wrong segmentation of shadow parts in the background and spill zones cause artefacts. The color scale factor can be set interactively in the billboard shader by means of a slider in the OpenRT application. Finding the right value for a setup is hence a matter of adjusting the slider.

### 5.4.5 Results

Figure 5.10 shows some more examples for the billboard application. The subject can even be rendered through a refractive glass sphere, which is difficult to do with a GPU based approach.

The rendering performance of a billboard is mainly based on the number of necessary transparency rays. This number is typically high since the billboard rectangle is usually wide to allow the actor to move around. Is has also to be taken into account that secondary rays like reflected rays and shadow rays hit the billboard. For each ray the chroma keying is performed. The chroma keying method used in this example is rather simple and fast, though. A more sophisticated keying method provides better results on closeups.

Table 5.1 gives some frame rates for video texture sizes of about 320x240 pixel and different numbers of CPUs. The aspect ratio of the texture was adapted to the billboard geometry by cutting out the subject using texture coordinates. Switching the video streaming off (resulting in a still image on the billboard), yield slightly higher rates (about 1–2 fps faster).

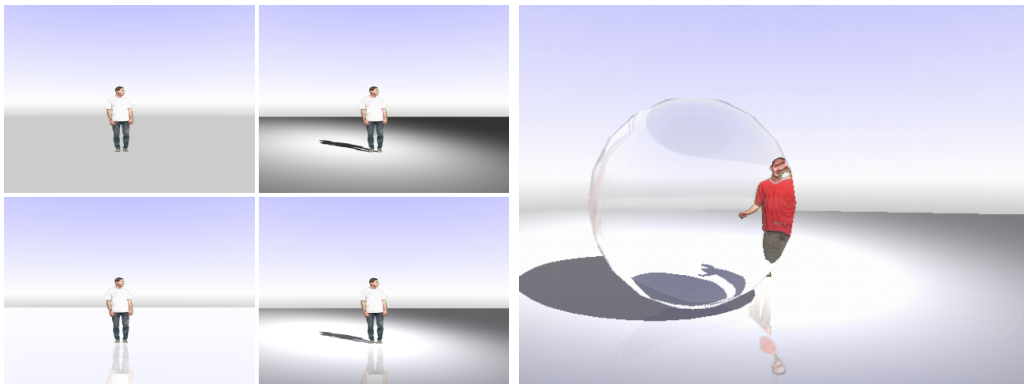


*Figure 5.10: Some examples of video billboards with OpenRT. The left image shows how rendering effects like shadows and reflections influence the result by providing the necessary visual cues to 'anchor' the subject in the scene. The right image shows a person walking behind a refractive glass sphere. This example is difficult to achieve with graphics hardware.*

| Scene | Figure | #CPUs | Resolution | fps |
|---|---|---|---|---|
| 1 person w/o | 5.10 left | 6 | 640x480 | 8.3 |
| 1 person w/o | 5.10 left | 16 | 640x480 | 17 |
| 1 person w/o | 5.10 left | 24 | 640x480 | 20.1 |
| 2 person w/lighting+reflection | 5.6 | 6 | 640x480 | 4.5 |
| 2 person w/lighting+reflection | 5.6 | 16 | 640x480 | 14.3 |
| Glass sphere | 5.10 right | 6 | 640x480 | 5.5 |
| Glass sphere | 5.10 right | 16 | 640x480 | 16.3 |

*Table 5.1: Some frame rates for the video billboard examples. All measurements were done on Athlon MP 1800+ CPUs. The OpenRT version is limited to about 20fps@640x480 due to network bandwidth.*

## 5.5 Drawbacks of Billboards

Billboards are a simple and effective drawing primitive and allow easy insertion of a subject into a virtual scene. Due to their flat, two-dimensional character, they have a number of drawbacks, however.

*Perspective distortion* is a problem with billboards (Figure 5.11). The billboard definition dictates to rotate the rectangle always perpendicular to the camera axis. In a ray tracer, secondary rays can have arbitrary directions different from the primary camera direction. This may cause wrong perspective in the rendering of reflections.

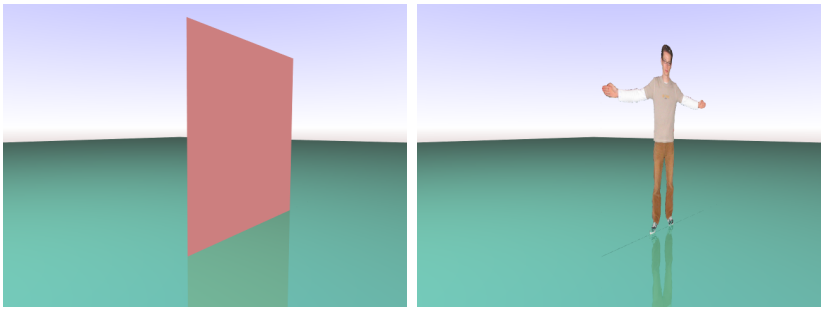


*Figure 5.11: Perspective distortion of a billboard when viewed at a shallower angle than the right angle. The billboard could be aligned perpendicular to the camera axis, but not for primary and secondary rays at the same time.*

Another problem with billboards is the constant *depth value* they provide. Depth-based compositing of a scene, to allow an actor to walk around an object, is not possible.

Since billboards show a 2D (perspective) image of the real 3D scene in the studio, perspective is a problem if the subject walks forth and back in relation to the camera. The depth dimension is mapped to an up and down movement in the perspective projection performed in the camera lens. This results in a *floating* effect (Figure 5.12): the subject begins to levitate in the virtual world if he walks away from the camera in the real world.

The floating effect can be compensated by adjusting the billboard texture area in a way that the subjects silhouette always begins at the bottom of the billboard. This can be accomplished by adding a certain offset to the vertical texture coordinate inside the shader. The offset is chosen according to the bounding box of the silhouette. Note that the relation of the texture image to the billboard geometry is constant and thus not influenced by perspective.

Since a billboard shows only one view of an actor, reflection rays hitting the billboard can result in *wrong reflections*. Figure 5.13 shows an example

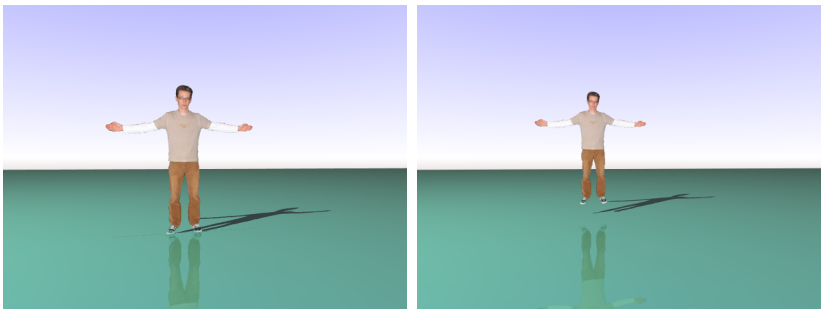


*Figure 5.12: The floating effect that occurs with billboards when the actor walks away from the real world camera. The effect is caused by the perspective mapping inside the camera.*

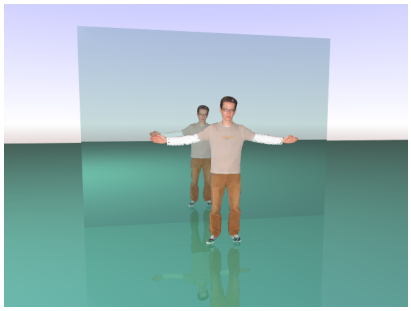


*Figure 5.13: A mirror behind a billboard yields a wrong reflection since there's only one view of the subject available.*

with a mirror behind the actor. This is an obvious effect due to the simple nature of a billboard. Multi-view billboards can compensate this effect. A multi-view billboard features a view-dependent texture [Debevec98c], i.e. the texture image is switched according to the incident ray angle. Multiple cameras are needed for acquisition. Because of the other drawbacks still remaining, a 3D reconstruction approach (Chapter 8) provides a better result than the use of multi-view billboards.

The same problem as with reflection rays occurs with shadow rays. Since the billboard is aligned to the camera position, a lightsource 'sees' the billboard from a shallow angle and thus yield a *smaller projected area* of the silhouette shadow. Figure 5.14 shows an example of a vanishing shadow as the incident light direction changes.

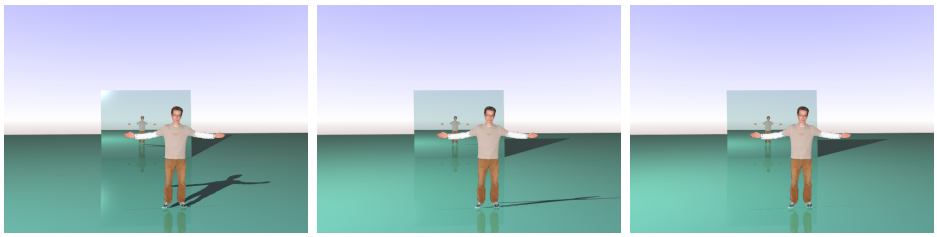


*Figure 5.14: Using billboards to project a shadow from shallow angles results in a too small shadow area. For the light direction parallel to the billboard rectangle, the shadow disappears.*

## 5.6 Conclusion and Future Work

Video billboards provide a simple and effective means of inserting subjects into a virtual scene. While coming from the raster graphics world to overcome problems with rendering performance (image-based rendering), the billboard concept also fits well into a ray tracing framework. In contrast to complex multi-pass rendering effects and projective texturing necessary in raster graphics for rendering environmental effects like shadows and reflections from billboard, a ray tracer provides a much more elegant way to accomplish these effects. The necessary shaders for the billboard object and the rest of the scene are completely independent.

The billboard concept is more flexible than the traditional compositing approach (Section 5.1.1), but also has a number of drawbacks. These drawbacks could be diminished by extending the billboard concept. But since most drawbacks are caused by the 2D nature of a billboard, a proper 3D approach is the better choice (see also [Grau01]).

In Chapter 8, I will describe a full 3D solution for live actor insertion into a virtual scene by extending the billboard concept to an *in-shader 3D reconstruction method*, overcoming with all billboard drawbacks.

# Chapter 6

# Augmented Reality View Compositing

In this chapter, I describe an extension to the OpenRT framework for *video-based* augmented reality rendering. The concept of *in-shader* compositing, introduced in Chapter 5, is used for implementing *differential rendering* methods to enhance the real video background with effects from the synthetic objects like shadows and reflections.

## 6.1 Video-Based Augmented Reality

For augmented reality applications, a method to combine the real parts of a scene with the synthetic, computer generated parts is needed. This is referred to as *compositing* (see also Chapter 5). One (simple) method to accomplish this is called *video-based augmented reality*. The synthetic parts of the scene are rendered over a video signal from a camera (the *view camera*).

Figure 6.1 shows a traditional system using a specialized *keyer*. This system is very similar to the one used in virtual TV studios (Section 5.1.1). A keyer is a video device that allows the blending of two (synchronized) video signals according to the luminance of a third signal (*key signal* [Poynton03]).

In an AR system, the key signal is delivered by the renderer, e.g. in terms of an alpha signal [Poynton03], where in the virtual studio the key signal is usually derived from the camera input (e.g. by chroma keying). This type of AR compositing system is also referred to as *video-see-thru*.

Today this approach is a bit outdated since most computers feature a video input for a camera. Compositing can then be done in software or using appropriate OpenGL framebuffer operations [Woo97]. The final AR output is displayed on the computer monitor. Simple AR applications just render

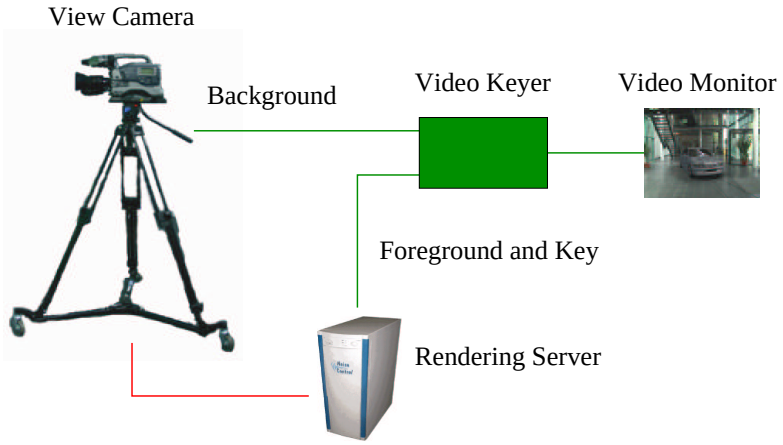



*Figure 6.1: A traditional video-based augmented reality system. Both, the view camera and the rendering server are connected to a video keyer. Based on the key information in the video signal of the rendering output (alpha, luma or chroma), the keyer blends the camera video with the renderer output. A full interactive system needs matching of the virtual camera to the real camera by some tracking connection (red).*

the synthetic scene over the video frames by initializing the framebuffer with a frame from the camera rather than clearing it.

Alternatives to the *video-see-thru* method are for example the *optical-see-thru* method, using semi-reflective mirrors for compositing (e.g. in front of glasses showing the real scene and a 'mirrored-in' TFT display), and the augmentation of the real environment with the aid of video projectors (*spatially augmented reality*, e.g. [Raskar01]).

Please see [Bimber03b] for a comprehensive discussion of the endless number of possibilities of AR compositing. In this thesis, I refer to video-based compositing methods only.

### 6.1.1 Camera Tracking

For a convincing result, matching of the virtual camera to the real video camera acquiring the video view is necessary (*tracking*). This has to be performed in real-time.

To match the perspective and camera pose the *intrinsic* and *extrinsic* camera parameters have to be known. The intrinsic parameters describe the camera mapping from real world points to the image plane considering lens distortion, pixel resolution, etc. They are typically determined by taking

pictures of a calibration target like a checker board. Often the intrinsic parameters remain constant during a session and can be acquired offline in a pre-processing step.

The extrinsic parameters describe the camera position and pose. For tracking, there are several methods: optical tracking, electro-magnetic tracking or mechanical tracking are the most important ones.

*Optical tracking* of camera pose can be done in two ways. One method is to use the camera output and a number of markers with known 3D positions in the scene. The 2D position of these markers is tracked in the images and the camera position and pose is determined by solving a non-linear system. The other method is to use a number of secondary cameras tracking a set of markers on the AR view camera.

*Electro-magnetic tracking* is mostly used in VR environments for user interaction. Due to the influence of all the other equipment needed for AR, the electro-magnetic field is highly distorted and calibration is cumbersome. This method is seldom used for AR tracking.

A good alternative for environments where optical or electro-magnetic tracking cannot be used is *mechanical tracking*. The video output device (often a flat TFT display) is mounted together with the view camera onto a hinged arm. The hinges positions are tracked by electronic position sensors like rotary encoders or precision potentiometers. The user can move the display around and 'inspect' the real scenery with the virtual parts in it by seeing 'thru' the monitor. This method is often referred by the metaphor of a 'window'.

Note that the same issues apply to camera tracking for virtual television studios (Section 5.1). Another discussion of camera calibration in the context of 3D reconstruction can be found in Section 8.3.1.

### 6.1.2 AR Compositing

In [Porter84], Porter and Duff describe compositing from an algebraic point of view. Two images can be combined by a number of operators. These can be just as easy as replacing the color information in one image by the other. More complex operators use auxiliary information for combining the image data. This information can be thought as a third image controlling the combining process. In practical terms this image is called a *mask* (also *matte* in term of photography/cinematography or *alpha channel* in computer graphics).

For a AR view compositing system, this information is usually supplied from the renderer as an alpha or a depth signal. Compositing is done using

a special hardware device (*keyer* [Poynton03]) or directly in the framebuffer [Woo97].

Since the video view camera provides no 3D information from the real scene in terms of distance of objects to the camera (*depth*), it is hard to render mutual occlusion effects of real and virtual objects.

This problem can be overcome by using rough models (*stand-in*, *double* or *phantom* geometry) of the real scene in the rendering process. Occlusion can be simulated by forcing the alpha signal to use the video background as foreground (*depth based compositing*). Manual (or semi-manual) work is necessary to generate those models of the real scene (see e.g. [Gibson03a]).

## 6.2 The Concept of In-Shader Compositing for Augmented Reality

*In-shader compositing* methods break with the concept of an explicit matte for blending foreground and background. The calculations necessary for generating a matte are implicitly performed in the shading process. The background image is used as an additional input to the shader. There's no need for a compositing post-process any more (see also Chapter 5).

Traditional compositing methods only allow to replace (or to blend) the color of the video background with the color from the rendering. Also, the color of the background does not affect the rendering process in any way.

With in-shader compositing, the additional background information can not only be used for generating the implicit matte (e.g. in-shader chroma keying), but also for performing *differential rendering* methods (see further below).

How does in-shader compositing works? Like for traditional compositing, we assume that the view video camera and the virtual camera of the renderer match in terms of perspective. Corresponding pixels in the background image and the rendered image are thus related to the same scene parts.

For each output image pixel the corresponding background image pixel color is accessible in the shading process. The shading algorithm can use the background color for segmentation (as described in Chapter 5), or for blending with a synthetic object. The simplest operation would be to just overwrite the background color with the synthetic objects' color. For a semi-transparent synthetic object, the background color can be used in the same way as the shading result of a transparency ray in ray tracing. The result of the shading calculation is the final composite output.

For output pixels that do not show any scene geometry, usually no shad-

ing calculation is performed (at last in the demand-driven concept of a ray tracer). For this case just the background color is used instead of the result of a shading process. Thus image parts that contain no scene geometry automatically show the background.

The concept of in-shader compositing has a number of benefits over traditional methods:

- The overall system design gets simpler since there's no separate compositing step. Also there's no need to carry auxiliary information (alpha or depth) from the renderer to the compositing process. Precision of auxiliary data is no issue (e.g. for depth compositing).

- In-shader compositing is an implicit 3D compositing method. All 3D related information like surface normals or the absolute position of a shaded surface point in the scene are available in the shading process and can thus be included into the compositing.

- Combined shading and compositing methods, like *differential rendering*, are easy to implement. There is no need for intermediate mattes and difference images (see e.g. [Debevec98a] for a 'traditional' approach to differential rendering).

- In-shader compositing fits quite good into the demand-driven concept of ray tracing. Since compositing is only performed where virtual objects are shaded, there's no unnecessary compositing calculation compared to the framebuffer compositing approach [Porter84].

There are also some drawbacks of in-shader compositing. To use in-shader compositing on graphics hardware (GPU), the shading process needs to be programmable. The background image can be accessed by using a texture map in combination with automatically generated projective texture coordinates that match the view frustum [Woo97].

In a distributed rendering system like OpenRT the background view needs to be streamed to the rendering clients. This can introduce additional latency which is usually unwanted in interactive AR applications.

## 6.3 Differential Rendering

The process of *differential rendering* is a method to combine the rendered parts with the real parts of an AR scene with the aim to show the effects (shadows, reflections or caustics) of the virtual part on the real part. Simply

speaking, this is achieved by 'appropriately' modifying the color information on the background image.

Let's assume we have a synthetic model of a real (background) scene that allows us to render an image of it. Camera calibration ensures that this rendered image matches a real image of the background. We call the rendered image $R_{without}$, which means it is rendered *without* the synthetic objects we want to insert. $R_{with}$ should denote its counterpart *with* the synthetic objects. The real background image is called $B$.

When we compare $B$ and our synthetic version $R_{without}$ of it, we will notice a *difference* $D$ due to the fact that the BRDF of the model does not correctly match the real scene (assuming the illumination in both scenes is equal, see also Chapter 7). The difference can be written as

$$D = R_{without} - B \tag{6.1}$$

We get the same error when we compare the full synthetic scene $R_{with}$ and the final composited output image $F$, thus

$$F = R_{with} - D \tag{6.2}$$

We can write

$$F = B + (R_{with} - R_{without}) \tag{6.3}$$

which is the basis equation of *differential rendering* [Debevec98a]. Whenever $R_{with}$ and $R_{without}$ are the same in the scene (i.e. where the virtual objects have no influence on the local scene), the output becomes simply $F = B$, what means we just see the background image. Where $R_{with}$ is darker than $R_{without}$, light is subtracted to form shadows and where it's lighter, light is added for reflections or caustics.

For practical purpose, it is recommended that the rendering takes a good approximation of the BRDF of the local scene into account. Otherwise $D$ can get too large and Equation 6.2 results in a negative $F$. [Debevec98a] gives a method for estimating the BRDF (see also e.g. [Fournier93, Sato99a]).

An alternative approach is rather based on the *relative* error than on the absolute value $D$ [Debevec98a, Sato99a]:

$$F = B \cdot \frac{R_{with}}{R_{without}} \tag{6.4}$$

This *quotient method* is better suited for using with in-shader compositing since the background color needs just to be multiplied by

$$\delta = \frac{R_{with}}{R_{without}} \tag{6.5}$$

.

The example application at the end of Chapter 7 uses this method.

For an illustration of a differential rendering method, please see also Figures 6.3 and 6.4 in Section 6.4.3.

### 6.3.1 Stand-In Geometry

Since the real scene is only represented as an 2D image in the in-shader compositing process, additional information is necessary to represent the 3D relation between virtual and real objects. This relationship is important to correctly render mutual occlusion of real and synthetic objects and shadows or reflections of synthetic objects onto real ones.

The necessary information is supplied as a *model* of the real scene. The AR view camera setup must be chosen to match a rendered view of this model to the real scene. The model contains *stand-in* geometry and material descriptions for the *real* objects. Together with the model of the *virtual scene* (the inserted objects) this is the scene input to the AR renderer. Special (e.g. differential rendering) shaders are used with the stand-in geometry.

This stand-in geometry is only necessary for the parts of the real scene affected by the virtual scene. We call this part of the real scene 'near' the virtual scene the *local* scene. The unaffected part of the real scene is accordingly called the *distant* scene [Debevec98a].

We have thus three scene parts for differential rendering: the *virtual* (or *synthetic*) scene, containing the objects to insert, the *distant* scene, which just contains the surrounding background and which needs not to be known in the rendering process[1], and the *local* scene showing the effects of the inserted objects and about which we have to know at least the geometry [Debevec98a].

Stand-in objects can be a very rough model of the real world. For instance for rendering the shadow of a virtual object onto a real floor, a small rectangle representing the affected area of the floor can be sufficient.

Stand-in objects are usually invisible in the composite AR image. Only rendering effects like reflections or shadows give an idea of their presence. 'Invisible' stand-in geometry can be used to simulate mutual occlusion between real and virtual objects. The stand-in object representing the real object is thus associated with a simple shader that just returns the background color.

[1] This is not true for reflective virtual objects showing the whole real scene. This cannot be accomplished with simple in-shader compositing. See Section 7.6 for a possible solution.

## 6.4 AR View Compositing in OpenRT

In the following, I give an overview over the design of the AR view compositing system for OpenRT and how the background video is streamed to the rendering clients. A simple example application shows how AR view compositing can be used for implementing differential rendering in an AR application.

### 6.4.1 AR View Video Streaming

Since we want to render in a distributed OpenRT system, we need to stream the video data of the AR view camera over the network to the clients. Section 4.2 already provided a discussion of video streaming methods in the context of video textures.

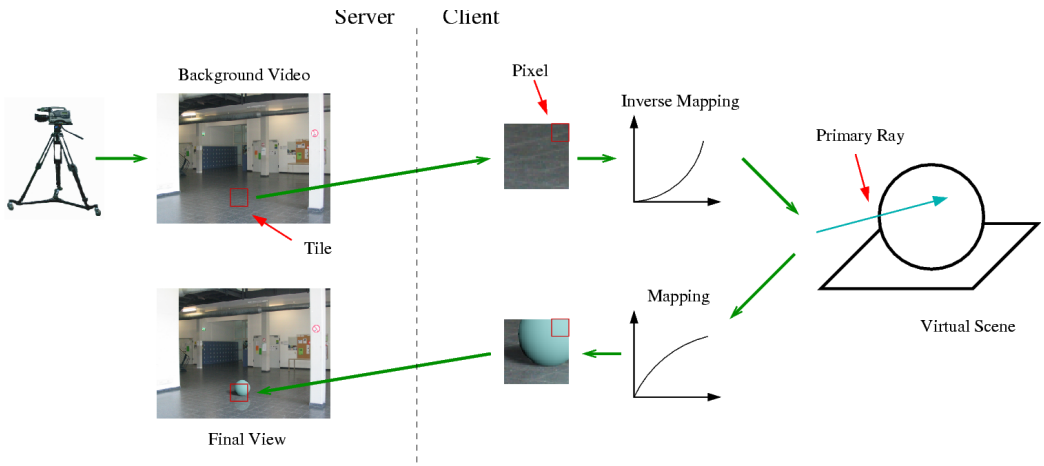



*Figure 6.2: Data flow in the OpenRT AR compositing subsystem. The camera captures the background video. The image part for one tile is scheduled on a client. The clients iterates over the pixels in the tile and applies the inverse tonemapping (Equation 6.7). A primary ray is initialized with the background color and intersected with the virtual scene. The result is tonemapped (Equation 6.6) and the completed tile sent back to the server.*

The multicast methods (Section 4.3) could be also applied for streaming the AR view information. This would be a waste of network bandwidth, though. Each client renders only small tiles (abt. 16x16 or 32x32 pixel) of the final rendering frame. Thus, a clients doesn't need to access the whole background image. Furthermore, the dropout problems related to the multicast streaming would be disturbing.

We opted for a variation of the method from Section 4.2.1 instead: the view data is streamed using the OpenRT payload mechanism (see Section 2.3.3). The payload mechanism uses TCP/IP, dropouts do not occur.

For AR compositing, the AR input image has the same size as the OpenRT rendering framebuffer. The framebuffer is subdivided into tiles for rendering. For each tile, the appropriate image data is taken from the input image. Together with the job description of the tile, the data is sent to a client determined by the OpenRT scheduling mechanism (Section 2.3).

The clients generate one (or more) primary ray(s) per pixel of a tile. If an object is hit, the `shader` function of the associated shader is called with an OpenRT *ray state* data structure as argument. This data structure contains a *color* field, which is used to store the color resulting from the shading calculations. The shader can read and set this color field. In an OpenRT version without AR view compositing, this field will be uninitialized when entering the `shade` function. The shader fills in the result, which will be stored by the rendering loop (contained in the rendering object, Section 2.3.3) in the tile image. Once a tile is completed, the tile image will be sent back to the OpenRT server.

To provide in-shader AR compositing, we just initialize the color field with the color taken from the background tile image. The shader can access this information and use the color in the shading process, e.g. for a differential rendering method. Note that it was not necessary to extend the OpenRT API since the API calls for setting and writing the color field were already available. Figure 6.2 illustrates the data flow.

The `shade` function is only called if an object is hit by a (primary) ray. When nothing is hit, the color field ray state data structure is directly stored in the output tile. This ensures, that the background view can be seen in those areas of the output image where no scene geometry was hit.

The main drawback of the in-shader AR compositing methods is that *all* view data is streamed to the clients. Since typical AR applications show virtual objects and their effects in only small portions of the output image, a lot of view data is streamed without any processing performed on the clients. Since the OpenRT server does not know whether a part of the render image contains any geometry, there is no (simple) alternative.

The AR view mechanism is implemented as an OpenRT rendering object (Section 2.3.3). Video input is supported by [Video4Linux].

Note that in a full-duplex network the additional load from the background video data equals the load caused by the tile images returned by the clients.

*Figure 6.3: An example of differential rendering. The left image shows a frame from the background video. In the right image, the virtual scene parts were just rendered over the video. A stand-in object for the floor plane was used to capture the shadow and the reflection of the sphere. Without differential rendering, the floor stand-in will not match to the real floor texture. No camera calibration was used.*

### 6.4.2 Tonemapping

One problem occurs when comparing a rendered image of the local scene and a frame from the view camera: the rendering is based on measured physical values like the incident irradiance while the (low dynamic range) video image is non-linearly compressed in dynamics. For rendering often a *dynamic compression function* (often referred to as *tonemapping*) is used to map the output values of the rendering algorithm (which can be in the range $[0, \infty]$) to $[0, 1]$ to match the needs of the low dynamic range output video display (Figure 6.2).

For the examples in this thesis, I use a simple compression function to map the intensity $I$ of the color channels R,G,B with

$$I' = 1 - e^{-I \cdot scaling} \tag{6.6}$$

where *scaling* is a user supplied value ('brightness slider'). We can use the inverse mapping function

$$I = -\frac{\log(1 - I')}{scaling} \tag{6.7}$$

for processing the video input used for a primary ray. This ensures that the two mappings cancel each other out when nothing is hit by the ray. Note that the scaling value then only affects the inserted objects and their

*Figure 6.4: The final output frame of the differential rendering method can be seen in the left image. The color of the stand-in for the floor was taken from the video background pixels this time. The shadow was only roughly estimated and not generated by measuring the incident light like for the IBL example in Chapter 7. The right image shows the same scene without the visual cues provided by the differential rendering method.*

effects while the video background remains at the same brightness. The video background brightness can be controlled via the Video4Linux controls.

### 6.4.3 A Differential Rendering Example

In this section I provide a simple example application in OpenRT to illustrate the mechanism of differential rendering in combination with the in-shader AR view compositing concept.

Figure 6.3 shows the background video (left image) and the virtual scene containing a sphere and a stand-in rectangle for the floor plane (right image). The incident light in the entrance hall is coming from the right of the frame and is only roughly estimated. The shader on the floor rectangle generates a shadow and a reflection of the sphere.

In Figure 6.4 the final output, achieved when the shader on the floor is adapted for differential rendering, is shown. Differential rendering was performed using the quotient method described in Section 6.3. The right image in Figure 6.4 shows the same scenario without effects of the virtual object. The lack of visual cues for the sphere makes it impossible to determine its position. The sphere seems to float in space. The AR view camera was not calibrated or tracked for this example.

Another example of an differential rendering applications can be found in Section 7.6.

### 6.4.4 Results

The OpenRT AR view mechanism achieves a frame rate of about 20 fps for a resolution of 640x480. The current[2] OpenRT implementation is limited to this frame rate due to network bandwidth and load balancing when rendering over the network. No video compression is used. For lower resolutions than 640x480 the full input video rate of 25 fps is obtained.

The video latency of the distributed version is about 4–5 frames. For the local rendering version a latency of about 1–2 frame is achieved. Note that the latency is at least one frame due to the synchronous buffering of the video input device.

Simple scene geometry like in Figures 6.3 and 6.4 do not have a noticeable impact on the frame rate. All measurements were done on Athlon MP 1800+ CPUs.

## 6.5 Conclusion and Future Work

The concept of in-shader compositing provides an interesting alternative to the traditional, matte based compositing approach. A number of benefits, especially in a demand-driven ray tracing framework, make it a rewarding choice for AR applications.

The OpenRT AR view compositing system is the basis for implementing AR rendering applications in the OpenRT framework. Streaming the background image to the clients allows to use the background colors in the shading process and fulfills thus the necessary prerequisites for differential rendering methods. Compositing in done inside the shader. This results in a straightforward way in the implementation of AR applications.

The main drawback is that all video data needs to be streamed to the clients, unregarded whether there's any scene geometry visible in a tile. The video latency is slightly higher than with traditional AR compositing methods when rendering over the network. Rendering on the local host only can provide a minimal latency but also slows down the achievable frame rate. For simple virtual objects and a rough quality of soft shadows this approach may still deliver interactive frame rates, though. Unfortunately human perception is very sensitive to this kind of latency and a delay of 4–5 frames at a video rate of 25 fps can already be distracting.

There is no video compression used for transmission of the background color data to the clients and for the tile data back to the server. Compression methods based on the same schemes as used for MPEG and JPEG

[2] At the time of writing this thesis.

[Poynton03] (macroblocks) could be adopted. They should fit the needs of compressing the rather small tiles (typically 16x16 or 32x32). Compression increases the latency due to the processing time for encoding and decoding, though.

Using the OpenRT payload mechanism based on TCP/IP ensures that there is no packet loss on the network and thus no video dropouts. Dropouts would be very disturbing to the AR user.

With the presented AR compositing method, real refraction of the background view by synthetic objects is not possible. If refractive effects are needed, a video texture with a panoramic representation of the background must be used.

For simple AR application a video resolution of 640x480 is often sufficient. In addition the number of pixels in the rendering that are affected by the virtual scene is often low. This meets the demand-driven concept of ray tracing and can allow a low latency rendering on a single machine without networking. Note that ray tracing also simply adapts to the needs of *interlaced* rendering [Poynton03] that is useful when using standard video output equipment for AR output.

# Chapter 7

# A Real-Time Lightprobe

Photorealistic augmented reality rendering is the art of 'fitting' synthetic, rendered objects into a real (background) scene in a convincing way. The human sensivity for very subtile visual errors in the composite scene makes this task very hard. All effects that are visible on the real objects like shadows and reflections need to be simulated for the virtual objects. *Consistent Lighting* is one of the key factors for convincing results. Other factors include e.g. matching camera perspective. For real-time application the challenge is even higher than for offline rendering.

*Figure 7.1: An Augmented Reality application example with OpenRT. The left photograph shows real cars in a hall. In the right image a virtual car is rendered live into the background video of a hall. A real-time lightprobe (on the floor at the cars position) captures the incident light for rendering. A differential rendering method creates an appropriate soft shadow on the real floor.*

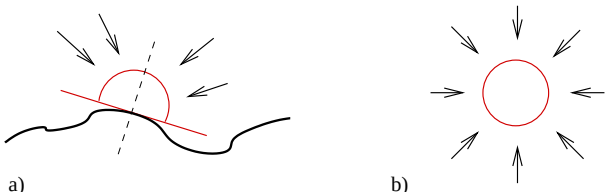


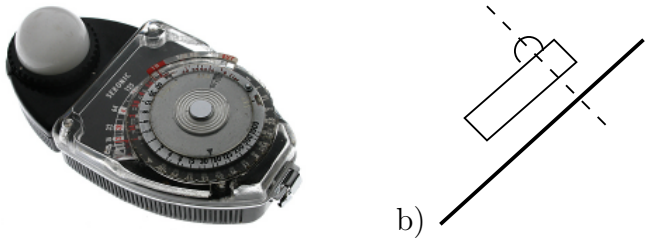

*Figure 7.2: Incident light.* **a)** *For the lighting (shading) of a surface point, the incident light over a hemisphere defined by the surface normal needs to be known.* **b)** *If the object shape is not known at the time the light is captured, the incident light from all directions must be measured.*

To get an idea of the spectrum of arising problems related to a photo-realistic AR rendering task, just imagine the following example application: Figure 7.1 shows in the left image a (real) car in a hall with large windows. Since there is no direct sunlight, the light thru the windows has a certain soft character that can be seen in the soft shadows on the floor. The image on the right side shows an interactively rendered version of a synthetic car in a hall. Before I explain the implementation details for this example, I will give an introduction to *interactive image-based lighting* methods and the related hardware technology. I believe, it is necessary to understand the limits of current technology to get a feeling what is possible today.

The key to interactive image-based lighting is *live measurement* of the incident light. The necessary light directions to shade a surface point of a virtual object are defined by the surface normal of the point. For interactive applications, where the final shape of an object is not known in before, a full spherical measurement is needed in order to determine the important sub-hemisphere later (Figure 7.2). Omnidirectional measurement of incident light can be accomplished with a *lightprobe* device (e.g. [Debevec98b]). For real-time purpose the lightprobe device must be capable of contiguous acquisition of incident light. A video camera is a simple example of such a device but not for incident light from all directions. It can provide a good starting point to built one, however!

In this chapter, I describe the basic principles of measuring incident light using digital cameras (Section 7.1), high dynamic range (HDR) camera technology (Section 7.1.2), fusion of a series of images of a scene at different exposures to a HDR image (Section 7.2), the basics of panoramic environment acquisition (Section 7.3), building of a real-time lightprobe (Section 7.5), the principles of image-based lighting (Section 3.2.3) and the implementation details for the example AR application shown in Figure 7.1 (Section 7.6). Finally, I provide a discussion and an outlook to future research work.

## 7.1 Measuring Incident Light



a) b)

*Figure 7.3: A Photographers Incident Lightmeter.* **a)** *The white diffuser hemisphere is used to integrate the incident light. A single sensor element under the hemisphere converts the light level to an electric voltage. The scale gives a readout in photometric units. A dial allows calibration for film speed.* **b)** *To take a measure of the light illuminating a plane, the lightmeter is held parallel to the plane.*

Figure 7.3 shows a photographers (and cinematographers) incident *lightmeter*. It is used to determine the necessary film (or video) exposure for a scene. A hemispherical diffuser provides a cosine weighted integration of the incident light. To take a reading, the lightmeter is held parallel to a surface. The lightmeter displays a value related to the reflected light, based on the measured incident light and a fixed reflectivity (nominal amount of 18 percent). A photographers lightmeter is only suited for static lighting conditions.

For the real-time rendering tasks, we have four additional demands to a lightmeter:

1. It has to provide a continuous stream of measuring values ('*live*').
2. The incident light from each direction has to be represented as a separate value (*directional resolution*).
3. We want the full sphere of directions (*omnidirectional*).
4. We want to measure 'real-world', physical values (*high dynamic range*).

A video camera provides at least a solution for points 1 and 2, so it's suggesting to start the design of a *real-time lightprobe* with a video camera. In the following sections, I provide the necessary background for understanding video camera sensors and how to generate high dynamic range (HDR) video to accomplish point 4 of the above list.

### 7.1.1 Digital Image Sensors

Modern still and video cameras use semiconductor chips as sensors. The sensors feature a grid of light sensitive cells, each acting like a tiny light-meter. Additional circuits on the chip provide readout, digital conversion and integration time control [Luther98, Ward00].

Color capability is achieved by combining three sensors with a beam splitter and color filters (red, green and blue) or by putting a color filter pattern directly in front the cells (e.g. Bayer pattern [Bayer76]).

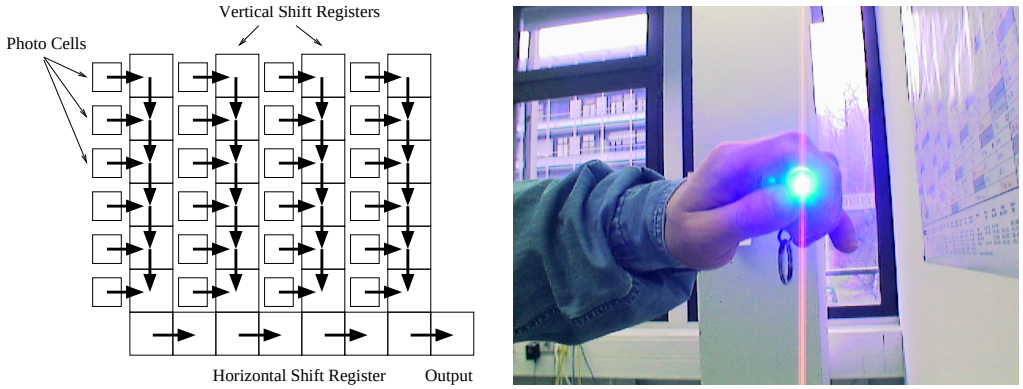



*Figure 7.4: A basic CCD sensor. Light sensitive cells feed vertical shift registers with charges proportional to the integrated incident light. A horizontal shift register provides the final (analog) output. The right image shows the* vertical smear *artefacts that occur with CCD sensors when very bright light causes leaking on the chip. The image was taken with a Sony DFW-V500 camera. A bright blue LED flashlight is held in front of the lens.*

Most digital cameras today use CCD[1] sensor technology. Light sensitive semiconductor cells collect electric charges. Each column of cells of features a vertical (analog) shift register feeding a single horizontal shift register outside the lower base of the light sensitive area (Figure 7.4). This layout was chosen to provide compatibility to the line based transmission of television video standards. The shift registers are protected from light to avoid changes in the charges after readout from the light sensitive cells [Ward00].

Excessive incident light can cause leaking of light directly into the vertical shift register. This results in image artefacts called *vertical smear* (see Figure 7.4 right). The timing of the transfer process from the cells into the vertical shift register can be varied to act as an electronic shutter. Note that vertical

[1]CCD: Charge Coupled Device

smear artefacts cannot be avoided by lower shutter times when an electronic shutter is used since penetrating of light directly into the shift registers occurs always and is not affected by the integration time.

Modern CCD chip designs try to overcome the smear problem by adding additional registers and a modified fast readout timing (FIT[2] technology). New HAD[3] and *hyper*HAD technology provide increased dynamic range and sensivity [Luther98, Ward00]. This makes CCD chips more expensive and is still used in professional (television) cameras only.

In recent time more and more cameras using CMOS[4] technology based sensors are available. The CMOS technology allows to build cameras with an increased dynamic range and does not suffer from artefacts like vertical smear [Luther98].

### 7.1.2 High Dynamic Range Cameras

Conventional camera sensors are only capable of capturing a moderate dynamic range of a few f-stops[5]. 'Real World' lighting conditions typically comprise a much higher contrast (dynamic ratio) than (LDR) sensors or photographic film emulsion is able to record correctly. In film and television work it is usual to adapt the existing lighting to the needs of the camera by adding or reducing light with a wide number of tools [Box99].

For measuring real-world lighting conditions, we need a camera sensor with an appropriate dynamic range to capture the high numbers of magnitude occurring in nature: a *high dynamic range (HDR)* camera. The output data of a high dynamic range video camera can be used for live rendering or recorded to disk in terms of a video file. Common high dynamic range image formats (like RGBE [Ward96] or OpenEXR [OpenEXR]) are not designed for video storage at high frame rates. A MPEG like compression is desirable, [Mantiuk04] describes the issues of modifying the MPEG data structures for high dynamic range capabilities.

High dynamic range cameras can be built on a number of different basic principles. In the following, I will give an overview and explain these principles in more detail and discuss their drawbacks and their applicability for a real-time lightprobe.

[2]FIT: Frame Interline Transfer

[3]HAD: Hole Accumulated Diode

[4]CMOS: Complementary Metal Oxide Semiconductor

[5]The f-stop is a photography related unit. A difference of one F-stop marks a doubling or halving in the amount of light. In electrical engineering the dynamic range is measured in dB (Dezibel). One F-stop correspondents to 6dB.

### 7.1.3 True High Dynamic Range Sensors

In the recent years, a number of truly HDR capable sensor chips have appeared on the market. Some of them are capable of capturing live video and (a small number of) off-the-shelf HDR cameras are available (e.g. [HDRC]). Video output on those cameras can be implemented by proprietary (digital) interfaces or by a dynamics compression system (logarithmic scaling) for standard video system output. These sensors are typically designed for computer vision under heavy lighting conditions, like quality checking at automatic assembly lines in factories. Here it is sufficient that often only *monochrome* images are acquired.

Due to the high price, the proprietary interfaces and since nearly all available cameras can only provide monochrome video, a true HDR camera is only a minor option for our (inexpensive) HDR lightprobe.

### 7.1.4 Spatially Varying Pixel Exposures

A high dynamic range sensor can be built out of a normal camera sensor and a spatially varying neutral density filter [Nayar00]. The filter is applied to the chip in the same manner as in color imaging (RGB pattern, e.g. Bayer pattern [Bayer76]), e.g. as a 2x2 group of pixels is filtered for four different exposures (see Figure 7.5). This reduces the effective resolution of the resulting HDR camera. A post-process reconstructs the final HDR image. This type of camera is rather inexpensive in production and capable of outputting HDR video. Since the ND (Neutral Density) filters already take the space in front of the sensor, these cameras typically deliver monochrome images.

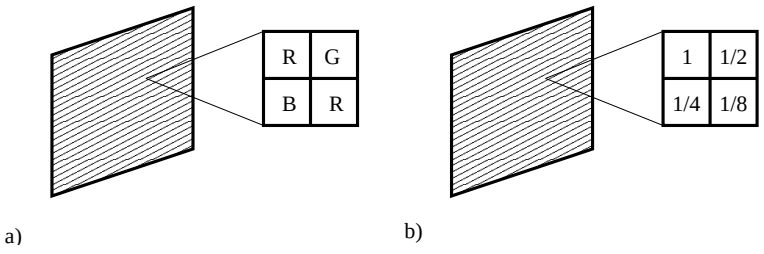



*Figure 7.5: A Spatially Varying Pixel Exposure Sensor. Instead of a color filter pattern on the sensor* **(a)**, *a neutral density pattern is used* **(b)**.

This type of camera is only a variation of the true HDR chip based cameras from the last section. Also fitting a (low cost) video sensor with an appropriate filter grid is hard. Hence this method provides no option for a lightprobe application.

### 7.1.5 Spatially Varying Image Exposures

Another approach to acquire a high dynamic range image of a scene is to project several congruent images of the scene with different exposures on the same (LDR) sensor side by side (Figure 7.6). The several images on the sensor are then composed by software to one HDR image.

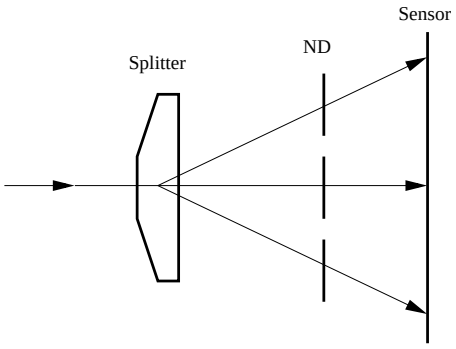



*Figure 7.6: A high dynamic range camera system based on the spatially varying image exposures method. A splitter prism is used to project the same input thru different ND filters on the same sensor side by side.*

The projection can be accomplished by a splitting device used behind or in front of a single lens. A set of different ND filters in front of the sensor areas ensure different exposures. The splitting of the incident light energy causes lower energy levels on the individual sensors compared to a single sensor solution. The necessary sensor gain amplification introduces noise.

Even though this method has low practical impact for industrial HDR sensor construction, the ICT real-time lightprobe [Waese01] is an example of a HDR camera based on this concept (see Section 7.5).

### 7.1.6 Multiple Sensors

A *beam-splitter* in combination with normal (LDR) cameras can be used to build a HDR camera[6]. Figure 7.7 illustrates the principle.

One drawback is the fixed and low number of simultaneously exposures due to the connections on the splitter. Stacking of splitters is not recommended because of optical problems.

Due to mechanical tolerances, the cameras need to be aligned for pixel registration in the individual images. This can be done by shifting the images

[6]In fact the same method is used in 3-chip color cameras for RGB colors.

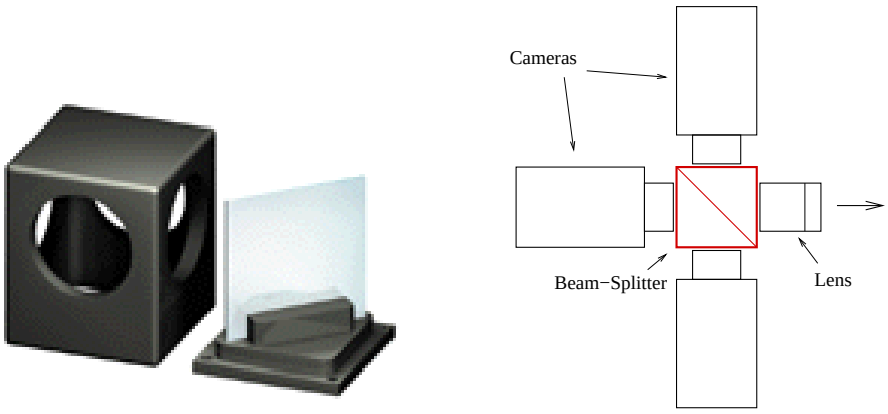



*Figure 7.7: A beam-splitter with a semi-reflective mirror allows three cameras to be connected to the same lens. Each camera is set to a different exposure.*

in software. Another problem is the increased *flange focal depth*[7] due to the space taken by the splitter, i.e. the lens is farther from the sensors as normal. When using standard (of-the-shelf) lens in such a configuration, it is not possible any more to focus at infinity. Monolithic beam-splitter based high dynamic range cameras need special constructed build-in lens therefore. The beam splitter also diminishes the signal-to-noise ration due to energy distribution to the several cameras and transmission loss in the semi-reflective mirror.

The cameras need to be synchronized to expose at the same point in time to deliver proper registered images. This method is capable of delivering HDR video. To combine the images of the cameras, a post-process is necessary that can be performed in real-time (see Section 7.2).

### 7.1.7 Sequential Multiple Exposures

One very popular method is to create high dynamic range images by acquiring a series of (LDR) images with different exposures (*exposure bracketing* like the photographer says) and combine them digitally. Exposure can be varied by *shutter time*, *aperture* (f-stop), *ND (Neutral Density) filters* or *gain* (on electronic cameras).

Varying the *shutter time* is the common method. Note that the shutter time can only be selected from a fixed set of times in a progressive (about halving) row (e.g. 1/50, 1/100, 1/250, 1/500, 1/1000, and 1/2000 on cheap video cameras).

[7] german: *Auflagemass*

Changing the *aperture* should be avoided since it influences the optical performance of the lens and possibly introduces a change in depth-of-field. Nevertheless, this is the method used on most professional (and also consumer) video cameras to automatically control exposure since the aperture can be set continuously and not in steps like shutter times. Aperture setting are describes in f-stops (e.g. F1.4, F2, F2.8, F4, F5.6, F8, F11, F16). The stop numbers resemble the light flux in relation to the aperture area (hence the $\sqrt{2}$ based numbers).

*ND filters* put in front of the lens are used when the available shutter times are to long for proper exposure. Some cheap video cameras only have a fixed shutter time and perform exposure via gain. Since ND filters cannot be changed automatically they are of limited use.

Most (video) cameras provide additional boosting of sensivity by *electronic gain* (amplification of the signal) increase. Higher gain causes noise and should be avoided. Cheap cameras use the gain setting for continuous auto exposure (*Auto Gain*) and have a fixed aperture.

To combine the images to a final HDR image a numerical process is necessary that can be performed in real-time (see Section 7.2). Note that all automatic exposure features on a camera have to be switched off for acquiring an image sequence. Also automatic white balancing should be performed first for a moderate exposure time and then locked to avoid color hue changes for the extreme exposure times.

This method is limited to static scenes since the exposures are taken at disjunct points in time and a moving camera or objects can not be registered properly for the HDR combination process. Some recent work tries to overcome with this drawback by using motion estimation [Liu03] (for slowly moving objects causing motion blur) and image warping [Kang03], but cannot provide real-time results. Despite the drawbacks, we chose the *sequential multiple exposure* method for our implementation of a real-time lightprobe since it is simple to implement and the cost is low compared to other methods.

## 7.2 Principles of Multiple Exposure High Dynamic Range Imaging

A wide number of publications and patents describe the basic principles of combining a series of LDR images into an output image with a higher dynamic range (see [Nayar00] for an overview). Practical method are described in detail in [Mann94, Robertson99, Debevec97, Madden93, Mitsunaga99, Battiato03] for example.

Mann [Mann94] refers to a collection of images that only differ in exposure as a *Wyckoff Set* (in honor of Charles Wyckoff [Wyckoff61]). Wyckoff was the first to exploit multi-exposure imaging in 1961 with the aim to enhance the dynamic range of photographic film. Figure 7.8 shows a sequence of images taken at different exposures.

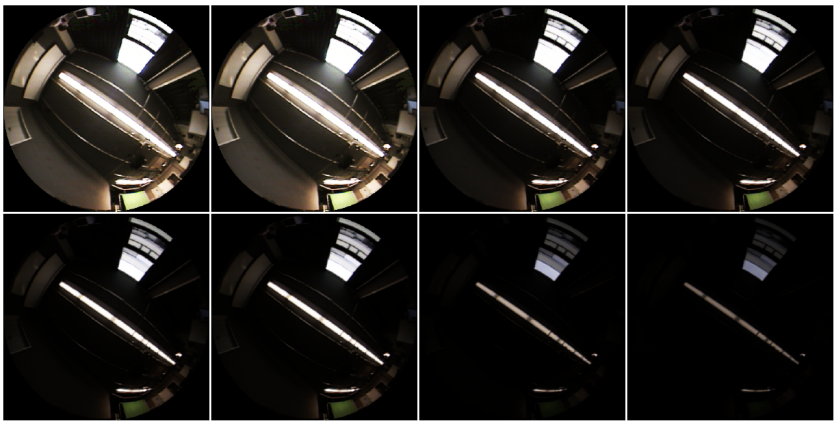


*Figure 7.8: A sequence of LDR images from a fish-eye lens lightprobe (see Section 7.5) with different exposures sorted from longest shutter time (top left) to shortest.*

In the following sections, I describe the process of combining a number of LDR images to a HDR output image. It is based on [Debevec97] and used for the real-time lightprobe device introduced in Section 7.5.

For the fusion of the images of a bracketing sequence, two steps are necessary: the camera response curve of the used camera has to be acquired and the weights for blending the images have to be determined. Figure 7.9a shows a typical camera response curve (also for photographic film emulsion). The curve is drawn as a function of *density* (**D**, from the chemically reduced silver particles in a developed film negative) over a logarithmic scale of exposure (**log E**). Since only a part of the immense dynamic range in real-world is transferred, the function 'clips' the density at the ends of the curve. The linear part in the middle section can be used for measuring. A 'certainty' function (Figure 7.9b) results in a high value for the linear parts. The certainty function is basically the first derivative of the response curve.

Figure 7.10 illustrates the principle of combining multiple images into one HDR image. The different exposures are chosen to fit together in their linear parts to form a new response curve. This curve can be seen as the virtual response curve of the (simulated) HDR camera. A 'certainty' functions

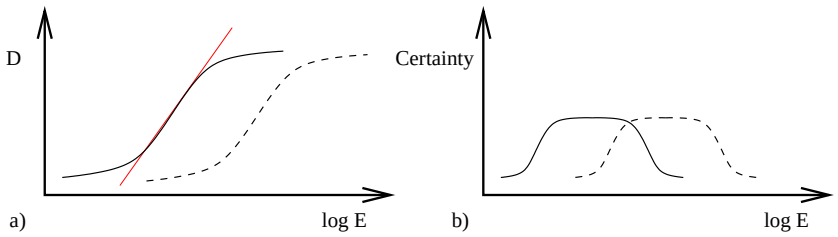



*Figure 7.9: A typical* **D log E** *(camera) system response curve.* **a)** *Two response curves for different exposures. The nearly linear section is marked by the red line.* **b)** *The corresponding 'certainty' functions.*

are used for blending the curves. The exposure difference in the bracketing sequence defines the 'shift' of a curve in the graph.

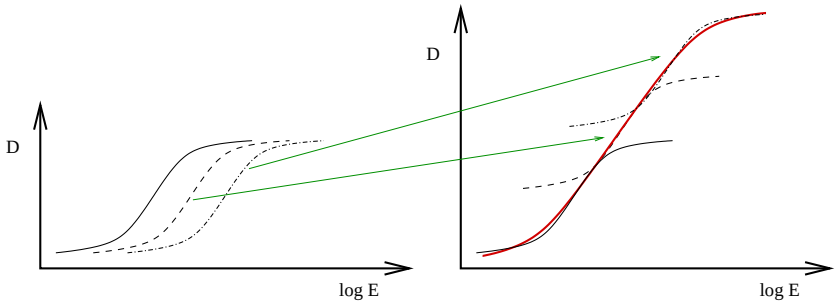



*Figure 7.10: The basic principle for compositing multiple (LDR) images with different exposure to one high dynamic range (HDR) image. The three response curves on the right are shifted vertically and combined to act like a single response curve (red) of a camera with increased dynamic range.*

### 7.2.1 Camera System Response Function

To determine a response curve, the whole system must be taken into account. This system consists not only of the camera but also the lens, aperture, the sensor, video gain, analog/digital conversion characteristics, the color matrix used to encode the colors and the interface to the host. This response curve needs only to be determined once for such a system.

In the following, I describe the necessary algorithms to determine the response curve and to reconstruct a high dynamic range image as used for the real-time lightprobe introduced in Section 7.5. The principles follow the method outlined in [Debevec97].

The exposure $X$ to which an image sensor had been exposed, is defined as the product of the irradiance $E$ arriving at the sensor cells and the exposure time $\Delta t$. For each pixel on the sensor the camera outputs a digital number $Z$, which is related to the exposure $X$ by a nonlinear response function $f$ (Color cameras output several numbers per pixel according to the color coding scheme, e.g. RGB444 or YUV422 [FourCC]). Once $f$ is recovered, the exposure $X$ at each pixel can be determined as $X = f^{-1}(Z)$, assuming that $f$ is monotonically increasing and thus $f^{-1}$ is well defined. The reciprocity assumption states that only the product $E\Delta t$ is important and e.g. doubling $E$ and halving $\Delta t$ produces the same result. If the exposure time $\Delta t$ is known, the irradiance can be computed by $E = X/\Delta t$.

An algorithm to determine the system response curve takes an exposure series of $P$ input images (Figure 7.8) with known exposure times $\Delta t_j$. A pixel value in the image $j$ at position $i$ is denoted by $Z_{ij}$. Reciprocity gives

$$Z_{ij} = f(E_i \Delta t_j) \tag{7.1}$$

The function $f$ is assumed to be invertible:

$$f^{-1}(Z_{ij}) = E_i \Delta t_j \tag{7.2}$$

Applying a natural logarithm on both sides yields:

$$\ln f^{-1}(Z_{ij}) = \ln E_i + \ln \Delta t_j \tag{7.3}$$

Substituting $g = \ln f^{-1}$ gives:

$$g(Z_{ij}) = \ln E_i + \ln \Delta t_{ij} \tag{7.4}$$

The two unknowns in this equation are the irradiances $E_i$ and the function $g$, which is smooth and monotonic.

In order to recover $g$, only a finite number of possible values of $g(Z)$ needs to be recovered, because the domain of camera output values $Z$ is finite (usually $Z \in [0...255]$). We denote the total number of pixel position in an input image by $N$ and the lowest and highest possible values of $Z$ by $Z_{min}$ and $Z_{max}$. We only need to recover the $(Z_{max} - Z_{min} + 1)$ values of $g(Z)$ and the $N$ values of $\ln E_i$ that minimize the following objective function to satisfy Equation 7.4 in a least-square sense:

$$\mathcal{O} = \sum_{i=1}^{N} \sum_{j=1}^{P} [g(Z_{ij}) - \ln E_i - \ln \Delta t_j]^2 + \lambda \sum_{z=Z_{min}+1}^{Z_{max}+1} g''(z)^2 \tag{7.5}$$

The first term in Equation 7.5 ensures that the solution satisfies the set of equations from 7.4 in a least-square error sense and the second term ensures

smoothness of $g$. The factor $\lambda$ serves as a weighting factor between the data fitting term and the smoothness term. It is dependent on the expected noise in the input images.

A solution can only be determined up to a scaling factor. The fitting of the data can be improved by exploiting the knowledge about the basic shape of a camera response curve with steep slopes near $Z_{min}$ and $Z_{max}$ causing poorer fitting. A simple triangle function

$$w(z) = \begin{cases} z - Z_{min} & for \quad z \leq \frac{1}{2}(Z_{min} + Z_{max}) \\ Z_{max} - z & for \quad z > \frac{1}{2}(Z_{min} + Z_{max}) \end{cases} \tag{7.6}$$

as weighting function will put higher weights on the smoothness and fitting terms in the middle of the curve. This yields

$$\mathcal{O} = \sum_{i=1}^{N} \sum_{j=1}^{P} w(Z_{ij})[g(Z_{ij}) - \ln E_i - \ln \Delta t_j]^2 + \lambda \sum_{z=Z_{min}+1}^{Z_{max}+1} [w(z)g''(z)]^2 \tag{7.7}$$

In this algorithm $N$ pixels and $P$ input images are taken, while it is sufficient to solve only for $N$ values of $\ln E_i$ and $(Z_{max} - Z_{min})$ values of $g$. This means that not every set of corresponding pixels has to be taken into account – also for reasons of computational complexity. In order for the linear system to be sufficiently overdetermined, it should hold that $N(P-1) > (Z_{max} - Z_{min})$. A set of pixel positions can be generated by the use of evenly distributed random numbers. Since $\mathcal{O}$ is quadratic in $E_i$ and $g(Z)$, minimizing is a linear least squares problem that can be solved using singular value decomposition (SVD) [Press99]. Figure 7.11 shows a reconstructed response curve for a JAI CV-S3300 camera [Jai], the camera we used for the real-time lightprobe in Section 7.5. Reconstruction of the curve was done using 256 random pixel positions from 5 input images [Hoffmann03]. Computation takes several seconds depending on the input image data.

### 7.2.2 Image Reconstruction

With the system response curve $g$ known, pixel values can be composed to high dynamic range values by a simple algorithm. Equation 7.4 solved for $E_i$ yields:

$$\ln E_i = g(Z_{ij}) - \ln \Delta t_j \tag{7.8}$$

For best results, all exposures should be taken into account to compute the radiance values as a weighted average. The weighting function from

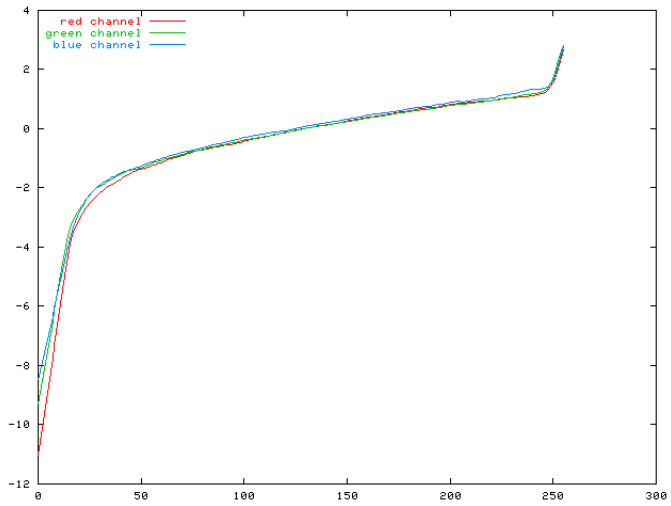



*Figure 7.11: The response curves of a JAI CV-S3300 camera for the red, green and blue channels (including the frame-grabber conversion) reconstructed from 256 random samples. The x-axis shows the image output values in the range* [0...255]. *On the y-axis the function values for g according to Equation 7.4 are shown. Note that the axes are mirrored compared to the standard* **D log E** *plot.*

Equation 7.6 can be reused to give exposures in the middle of the response curve a higher weight and ignore saturated pixels near $Z_{min}$ and $Z_{max}$:

$$\ln E_i = \frac{\sum_{j=1}^{P} w(Z_{ij})(g(Z_{ij}) - \ln \Delta t_j)}{\sum_{j=1}^{P} w(Z_{ij})} \tag{7.9}$$

The reconstruction algorithm now works as following:

```
for all pixels j do
    for all color channels c do
        total weight W = 0
        sum s = 0
        for all images i do
            weight w = weightingFunction(i, j, c)
            s = s + w × (responseCurve(i, j, c) − log(exposureTime(i))
            W = W + w
        s = s/W
        resulting HDR value v = exp(s)
```

The computational cost for composing one high dynamic range image from $P$ input images for one color channel is thus: $2P$ table lookups (one for the weighting function and one for the response curve), $P$ multiplications, $P$ logarithms, $3P$ additions, 1 division and 1 exponentiation.

This method allows a fast reconstruction at video rates on modern PCs [Hoffmann03]. Note that the linear system for the response curve recovery must only be solved once for a camera system. For further results see Section 7.5.

## 7.3 Panoramic Acquisition

To complete the list of demands from Section 7.1, there is still one point missing: panoramic acquisition of incident light from all directions. There are a number of methods trying to accomplish this. I will give a brief discussion of the most important ones in the next sections.

### 7.3.1 Mirror Balls

A common method to acquire incident light from all direction is to photograph a mirror ball. The reflection in the ball shows nearly the full environment, except a small angle behind the ball. Often a mirror ball shot is assumed to be ideal, i.e. the ball would be at an infinite distance from the camera and thus the camera can be assumed to be orthographic. This yields a simple mapping for a direction $D = (x, y, z)$ to image coordinates $s, t \in [-1, 1]$ for a cropped mirror ball image[8]:

$$s = \frac{x}{\sqrt{2(z+1)}},\ t = \frac{y}{\sqrt{2(z+1)}} \tag{7.10}$$

Since the mapping is highly distorted at the outer image parts of the ball, often two photographs from different angles are taken and combined (Figure 7.12). This has also the advantage to get rid of the unwanted reflections of camera and photographer. To preserve the better sampling achievable with the two shots, another mapping is needed. Otherwise the additional information would be lost again.

Most mirror ball images (*lightprobes*) use an *angular mapping* ([Debevec]):

$$s = \frac{x}{\frac{1}{\pi} acos(z) \frac{1}{\sqrt{x^2+y^2}}},\ t = \frac{y}{\frac{1}{\pi} acos(z) \frac{1}{\sqrt{x^2+y^2}}} \tag{7.11}$$

[8] The actual equations depend on the axis assignment. Please note that the assumption of the infinite distance is not true for a real mirror ball shot.

*Figure 7.12: Two shots of a mirror ball from different camera positions enable to remove the reflection of the photographer and provide a better sampling compared to a single shot. The lightprobe image on the right is composed from both shots and uses angular mapping. (Images Courtesy of Paul Debevec).*

This mapping provides a better sampling in the off-center regions. Other mapping include *paraboloids* [Heidrich98] or a *cubemap* [Haeberli93]. Heidrich [Heidrich99] provides a discussion of the sampling properties of different mappings. Mirror ball shots are often used as *environment maps* [Greene86] to simulate reflections and to include a surrounding panorama around a scene.

Special panoramic cameras with a full or half mirror ball or paraboloid are available off-the-shelf. They are usually expensive and do not feature HDR output. Most devices are thus homebrew like the ICT lightprobe [Waese01]. There's even a publication in which the reflection in a human eye ball is used [Nishino04]. A large number of mirror ball shots (often as HDR lightprobes) can be found on the Web for free (e.g. at Paul Debevecs website [Debevec]) or can be bought from companies (e.g. [Dosch]).

In combination with a (HDR) video camera this method can be used for live video acquisition of an incident light map (lightprobe).

### 7.3.2 Fish-Eye Lens

A *fish-eye lens* provides another means for panoramic acquisition, though available lenses usually feature a field of view less than a hemisphere. The resulting image resembles a mirror ball but depends on the actual angle and distortion of the used lens. A calibration procedure is needed [Swaminathan00]. The sampling density depends on the lens distortion but is usually comparable to a mirror ball. Figures 7.8 and 7.19 show images acquired with a fish-eye lens.

To overcome the smaller acquisition field, two ore more fish-eye rigs can be combined. A fish-eye lens allows simple mounting on a (HDR) video camera. We used this acquisition method for out real-time lightprobe (Section 7.5).

a)  b) 

*Figure 7.13:* **a)** *A GlobuScope™ moving slit camera. Under the metal cap is a small conventional camera rotating uniformly around the vertical axis performing a single exposure. The slit in the cap is in front of the center of the wide angle (abt. 180 degree) lens. The camera is held at the bottom handle over the head of the photographer and the trigger button is pressed. It produces a very wide picture on 35mm film.* **b)** *The Spheron SpheroCam™ is an example of a digital rotating panorama camera. The SpheroCam HDR version includes a control software capable of acquiring HDR panoramas in less than 3 minutes. (Right photo courtesy of Spheron GmbH, Germany).*

### 7.3.3 Moving Cameras

Modern digital pocket size cameras often come with a panorama *stitching* software and a special photographing mode for (cylindrical) panoramas. The photographer takes a number of pictures while he turns around 360 degree (or less if an incomplete panorama is wanted). The software uses the lens setting information (focal length) in the image files [Exif] and creates warping maps for rectifying and composing the images to one panoramic picture. The camera mode ensures that all pictures are taken with the same settings and shows the necessary overlapping in the (LCD) finder.

In the days before computers and digital warping, ingenious engineers worked on the mechanical predecessors of our todays digital panoramic cameras. Since a later stitching of multiple pictures was to difficult, they tried to build a camera that exposes a single piece of film continously and found the *slit camera* principle for panorama cameras. A camera performs a 180 degree rotation with a constant speed and the film moved in synchronization. Instead of a shutter, a small slit, positioned in the center of the lens image, is used. This results in a very wide image on the film featuring only small distortion in the horizontal direction.

An example of these ancient (and still used) camera construction is the famous GlobuScope camera (Figure 7.13a). A modern descendant is the Spheron [Spheron] SpheroCam, a rotating camera that can acquire a full panorama in less than 3 minutes. For comparison: the old GlobuScope takes only about one second and delivers a panorama at a resolution about ten times higher than the Spheron on photographic film.

A very interesting and related approach is sketched in [Mann97]. A handheld video camera is used to 'scan' the environment, a post-process combines the video frame to a spheric panorama. The method can even be used to acquire HDR when the timing of the auto-exposure feature of the video camera is taken into account. While the camera is moved around, the auto-exposure constantly adapts and thus consecutive frames in the video are taken at different exposures. The panorama reconstruction can be combined with a photometric calibration to combine the frames into a HDR output [Mann96]. Unfortunately, there's no (stable) implementation yet.

For obvious reasons, moving camera methods are not suited for acquisition of live panoramic video and thus not applicable for our car example. The method of Mann [Mann96] could provide an inexpensive option for acquiring HDR panoramas at huge resolutions, though.

### 7.3.4 Multi-Sensor Rigs

To get panorama (video) at a high resolution, several individual (video) cameras can be rigged according to the desired panorama parameterization (cylindrical, spherical or cubic). A geometric calibration procedure and a post-processing (*stitching*) in software are necessary [Swaminathan00, Nielsen01, Nielsen02].

Figure 7.14 shows a commercial solution, the Point Grey Ladybug camera [PointGrey]. It is capable of recording panoramic video of up to 15 fps on a special recording unit for later download to a host (only low dynamic range). This special hardware is rather expensive. The Ladybug was e.g. used for generating shadows in [Hughes04b, Hughes04a].

## 7.4 Restrictions of a Single Panoramic Lightprobe

When using lightprobes and environment maps for rendering, we implicitly assume that they show lights and objects at an infinite distance from the rendered virtual objects (*distant scene* vs. *local scene*, see also Section 6.2 and [Debevec98a]). For practical applications, this assumption is seldom

*Figure 7.14: The Point Grey Ladybug camera uses six 1024x768 color video sensors that can deliver up to 15 fps LDR (up to 30 fps for lower resolution). A separate storage unit with a hard disk array and a proprietary 1.2 Gbps optical link to the camera head is used for panoramic video acquisition. The data can be downloaded to a host PC via an IEEE1394 interface. (Photo courtesy of Point Grey Research Inc., Vancouver BC).*

true. When you think of a full environment for the car scene (Figure 7.1) taken at the cars position, the floor in the lightprobe is *not* a distant object. Using the probe to render a reflection of the floor on the car will result in a wrong, distorted perspective.

The problem of the environment mapping assumption is sketched in Figure 7.15. Nearby objects cause an offset in the lookup and a wrong assumed direction. *Parallax effects* are the result.

Another problem is that a lightprobe is usually taken only at single position in a setup. The resulting lightprobe image thus shows only the incident light from all directions at this single point. In the example of the car scene, this problem can cause a person walking nearby the car to be rendered by a very distorted, grotesque and far too large reflection. Also patterns of light on the floor, like those caused by the shafts of light thru the windows, cannot be reproduced in the virtual car lighting.

In some cases, these problems can be diminished by using a *reprojection* of the single lightprobe on stand-in geometry representing the local scene [Debevec98a, Gibson03a]. This causes the virtual object to 'see' the nearby, real objects in the right directions. Reprojection is often done in a pre-process and by texturing the stand-in geometry with the lightprobe image content. This additional geometry is then used for ray tracing (e.g. [Debevec98a]) or radiosity simulation (e.g. [Gibson03a]). Of course, in a dynamic live environment it is not possible to provide stand-in geometry on demand for e.g. a person walking nearby the car.

To overcome the restrictions of a single lightprobe for acquisition, the suggesting solution is to use multiple probes. This approach is used for acquiring full *incident lightfields*.

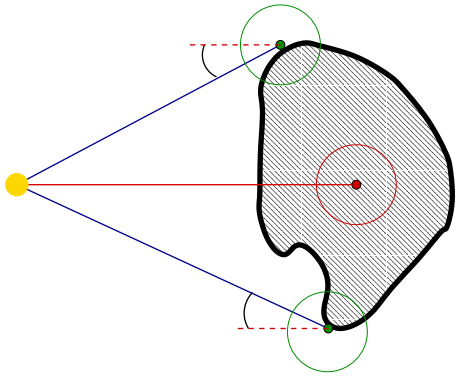


*Figure 7.15: The environment assumption fails for near lightsources. A lightprobe (or an environment map, red) is acquired at the center of an object and used for lighting the object surface (green). The actual direction towards the lightsource (yellow) from the surface points differs from the direction seen in the acquired map (angle between red dashed and blue lines). If the lightsource moves away to infinity this angle disappears and the environment assumption is a good approximation.*

### 7.4.1 Acquiring Incident Lightfields

An incident lightfield describes not only the incident light at one point but at a number of points of a surface. An even more complete information would be comprised in a whole *irradiance volume*, describing the irradiance at all points in a room from all directions [Greger98, Gibson03a]. Acquisition of such a full description is not possible today.

A simple method of acquiring light at different positions is to move around a mirror ball in the desired area and to take a photograph (of HDR series) for each position. The exact positions need to be considered for the rendering. This method is obviously not suited for real-time purpose.

Another method is to take a single (HDR) image and to use a bunch of mirror balls (Figure 7.16a). Since the balls occlude each other at a number of angles, usually only the upper hemisphere can be reconstructed properly [Unger03].

Figure 7.16b shows a method to overcome the occlusion problem: a single probe (fish-eye) is moved around by a motion control rig [Unger03].

Since there is today no method to acquire the irradiance volume of a room or even the incident lightfield over the floor plane, we concentrate in the following on the single probe approach, bearing in mind that it can fail in some situations.

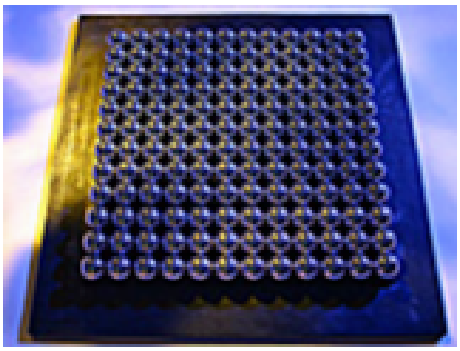
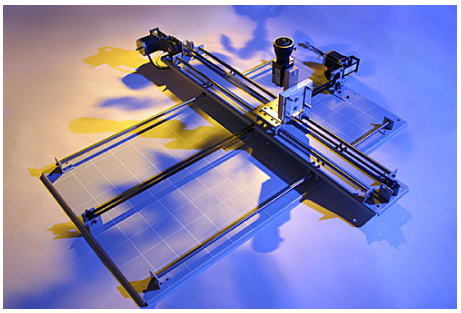
a)  b) 

*Figure 7.16: Two methods for the acquisition of incident lightfields.* **a)** *An array of mirror balls.* **b)** *A fish-eye lens camera on a 2D moving rig. Motors allow to place the camera at arbitrary positions. (Photos Courtesy of ICT, Paul Debevec)*

## 7.5 A Real-Time Lightprobe

In [Waese01], Waese and Debevec present an interesting design of a real-time lightprobe device: the ICT real-time lightprobe. This lightprobe uses the 'Spatially Varying Image Exposures' HDR camera principle explained in Section 7.1.5. A consumer video camcorder is mounted on a bar with a small mirror ball in front of the lens (Figure 7.17a). An effect prism (an off-the-shelf part available in photo shops, Figure 7.17b) on the lens in combination with several ND gel filters [Lee] fixed to the facets of the prism projects multiple images of the mirror ball at different exposures on the (LDR) sensor of the camera (Figure 7.17c). A post-processing step cuts out the individual images of the ball and combines them to a HDR lightprobe video. As an application example, [Waese01] provide a video clip with a person walking with the lightprobe around in their lab. In an offline rendering step, a spaceship model is rendered with an image-based lighting technique (see Section 3.2.3) in a video that was taken walking the same path without the effects prism (Video available at [Debevec]). Due to the video resolution of the camcorder and the prism the effective resolution of the final HDR lightprobe video is rather low.

For our car example, we need a *live* real-time lighprobe. The processing for the ICT probe could be done at real-time, but the fragile setup and the low HDR resolution seems to be inappropriate for us. We opted for a fish-eye lens based setup with a HDR camera instead. In the following sections, I will give a report of the implementation of our version of a real-time lightprobe.

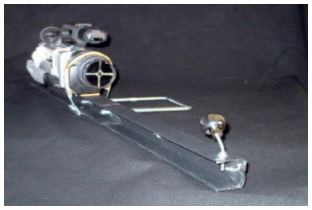
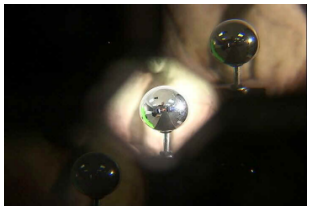
a)  b)  c) 

*Figure 7.17:* **a)** *The ICT real-time lightprobe consisting of a consumer video camcorder, a small mirror ball, an optical effect prism and ND filter gels.* **b)** *An effect prism filter to achieve multiple images.* **c)** *One frame of the recorded video. A post-process generates a HDR lightprobe video. (Photos a+c Courtesy of ICT, Jamie Waese)*

### 7.5.1 Building a Simple Video Lightprobe

The idea of a real-time lightprobe is to have a (small) device that can be put everywhere we want to know about the incident light. This device should ideally

- be inexpensive,
- be small,
- use off-the-shelf hardware,
- have a suitable dynamic range (HDR),
- have the highest possible resolution for reproducing specular reflections,
- run under Linux,
- and be capable of delivering several frames HDR per second.

When we started to work on a real-time lightprobe in 1999, real HDR cameras (Section 7.1.3) were not available. Since today only a few HDR cameras are on the market, all feature a proprietary interface (often without a Linux driver) and most of them are monochrome only.

Inspired by [Nelson99], we decided to go for a fish-eye based solution with a conventional video camera and exposure bracketing (Section 7.1.7). [Nelson99] used a Sony DXC-LS1 camera. A Nikon CoolPix FC-E8 fish-eye lens [Nikon] adapter (normally used to put in front of the CoolPix build-in lens) in combination with a BFI 2.8mm fixed focus lens [BFI] we found to

have a low and almost linear radial distortion[9]. The field of view is slightly less than 180 degree.

As a camera we decided to use the newly available Sony DFW-V500 camera [Sony] with an IEEE1394 interface [Anderson98] for digital video output and control. IEEE1394 support was brand-new and very experimental under Linux those days [Linux1394]. It took almost one year of experimenting [Repplinger01] to get the camera proper running. A firmware update at Sony Services was also necessary.

a)  b)  c) 

*Figure 7.18: The components of our real-time lightprobe.* **a)** *A JAI CV-S3300 camera with a BFI 2.8mm fixed focus lens. The cable is for serial control and can be extended with conventional XLR cables to the controlling host.* **b)** *A Nikon CoolPix Fisheye Adapter (Nikon FC-E8) on the lens mounting adapter.* **c)** *Our custom lathed lens adapter made of brass. The inner side is partially blackened to prevent lens flares.*

When we started our first tests with the assembled DFW-V500 based lightprobe, we had to learn about the bad dynamic response of the Sony CCD chip used in the DFW-V500. We were not able to find appropriate exposures to get images with direct lightsources in sight without vertical smear artefacts (see Figure 7.4). Nevertheless the Sony camera could change the shutter time very fast, at a video rate of 30 fps only one or two frames were unreliable.

Discouraged, we rested our efforts for a while and finally decided to try other cameras. Since some of our (expensive) Sony cameras refused any operation after two years (we bought a number of cameras for the visual hull experiments we intended, see Chapter 8), we wanted to go for a cheaper solution. We bought an inexpensive analog color CCD camera with serial (RS232) control: a JAI CV-S3300 [Jai]. Since the optics (the fish-eye adapter

[9]That means the function that maps the distance to the optical center of a pixel on the output image to an elevation angle is almost linear. Radial distortion can be examined simply by putting a tube of grid paper over the lens and observing the circles in the image.

and the BFI lens) we had from the Sony camera were designed for 1/3″ CCD sensors we hat to stay with this chip size.

The JAI camera (Figure 7.18a) provides analog CVBS and Y/C video output. The latter should be used in HDR applications due to cross luminance problems with high lighting frequencies [Poynton03]. Serial RS232 control at 9600 Baud allows to change shutter time and other parameters. The rather slow communication speed implies upcoming problems with the fast shutter time change rates we intended caused by a slow internal control processing speed. The JAI camera features shutter time from 1/50 to 1/10000 of a second.

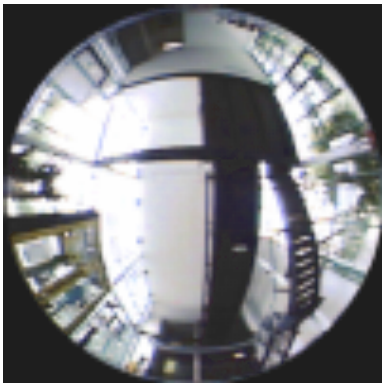
a)  b) 

*Figure 7.19: Our hemispherical real-time lightprobe.* **a)** *The assembled JAI camera based lightprobe. Three connections for power (12 VDC), serial control (RS232) and video output (Y/C) are necessary. The brass lens adapter allows to mount the Nikon FC-E8 on the BFI 2.8mm lens.* **b)** *An image taken in the entrance hall of the computer science building.*

Figure 7.18 shows the components of our lightprobe device. A custom lathed lens adapter was used to fit the fish-eye extension lens in front of the fixed focus wide angle lens. In Figure 7.19, the final assembled device can be seen.

The control software [Hoffmann03] takes care of changing the shutter frequencies and assembles the final HDR frames using the algorithm outlined in Section 7.2. In a photometric calibration postprocess the response curve of the lightprobe can be determined and saved to a file (see Figure 7.11). A frame-grabber board [Video4Linux] is used as an interface between the analog camera an the host PC.

### 7.5.2 Results

The optical quality of the lightprobe device is fair. The main problem is the bad design of the flange focal depth adjustment on the JAI camera which is done by screwing the CS-to-C mount adapter in and out[10] The adjustment is very important since the BFI lens is fixed focus and the focal plane moves slightly with the aperture setting. The aperture is the only parameter set at the lightprobe device manually. Slight shocks to the device can cause changes in the delicate flange focal depth (The depth is typically set in precision of a 1/100 mm).

Our real-time lightprobe captures only one hemisphere, or more exactly: slightly less than 180 degree. A full hemisphere could be achieved by combining two lightprobe devices. Experiments showed that the missing 2–3 degrees to the full hemisphere and the difficult mechanical alignment of two devices are cumbersome.

The JAI camera provides only a slow shutter time changing rate, i.e. when the command on the serial remote is send it takes a while for the camera to setup the new shutter time and several unreliable frames (at unknown exposure time or even with artefacts) are generated. Experiments showed that the reconstruction software has to skip at least 7–10 frames. In comparison the Sony DFW-V500 produced reliable output after only 1–2 frames. Note that the Sony DFW outputs a video rate of 30 fps compared to the 25 fps of the JAI camera. The slow time seems caused by the slow internal control processor. We also tested a Sony EVI-D100P camera with 38400 Baud control (VISCA) and experienced a comparable slow speed. The EVI camera showed a bad dynamic response by the way. Since it has a build-in zoom lens it is not suitable for our lightprobe.

The slow shutter time changing rate is the reason why we get only slightly more than one frame HDR output per second with 3 shutter times. The reconstruction algorithm is no limiting factor since the computational effort is rather low (Section 7.2.2). With the Sony DFW camera we could get about 8 fps HDR output with 3 shutter times but the bad dynamic response of the Sony sensor renders it unusable for our application.

In the recent time, newer IEEE1394 based cameras [PointGrey] featuring the necessary 1/3" sensor width and C-mount for our lens combination are available (even with CMOS sensors). For future experiments these cameras seem worth to take into account.

The JAI camera has a good dynamic response (compares to our Sony DFW and EVI cameras). This allows to use a minimal number of different

[10]Sony DFW cameras provide a much more precise control with a screw and a sliding wedge.

shutter times and thus a higher output frame rate of the lightprobe. Vertical smear is no problem since it occurs very seldom and could then be compensated with a more closed aperture.

The frame-grabber delivers a resolution of $640x480$ pixel with the image of the lightprobe centered in an area of about $400x400$ pixel. Without the black border around the circular image the effective resolution of the lightprobe is

$$400^2 \cdot \frac{\pi}{4} \approx 400^2 \cdot 0.7853 \approx 125600\, pixel. \tag{7.12}$$

This resolution is rather high compared to the ICT lightprobe [Waese01] but low in comparison to static lightprobe images taken with high resolution digital (photo) cameras. The resolution of the lightprobe is an issue when the image is used for specular reflections on a virtual object (see Section 7.6).

The analog video camera delivers an interlaced ([Poynton03]) video signal, which means that two *fields*, one containing only the odd and the other containing only the even pixel lines of the image, are captured rather than full frames (*progressive*). The correct rate of the video camera is thus defined as 50i (50 fields interlace) in opposition to the more desirable 25p (25 frames progressive). A 25p camera (or 30p, like the Sony DFW series) would be a better choice for a lightprobe application.

Our real-time lightprobe was used for the AR setup in Section 7.6 to capture the incident light at 'live' conditions and to add shadows and lighting to a virtual car composed into a live video background.

## 7.6 An OpenRT IBL Application Example

In this section, I explain the 'car in a hall' OpenRT example, introduced at the beginning of this chapter, in more detail. The basic idea was to create a real-time version of the methods described in [Debevec98a] and [Sato99a]. Live captured incident light from the upper hemisphere above the floor is used to light the car (including reflections) and to generate a soft shadow on the floor (*image based lighting*, IBL). The video texture mechanism (Chapter 4) and the AR view compositing (Chapter 6) are both used.

### 7.6.1 Hardware Setup

Figure 7.20 shows the necessary hardware setup. The real-time lightprobe device is put at the desired position for the virtual car in the hall (see also Figure 7.23 left). A JVC GY-DV500 video camera is used for acquiring a view of the hall for compositing.

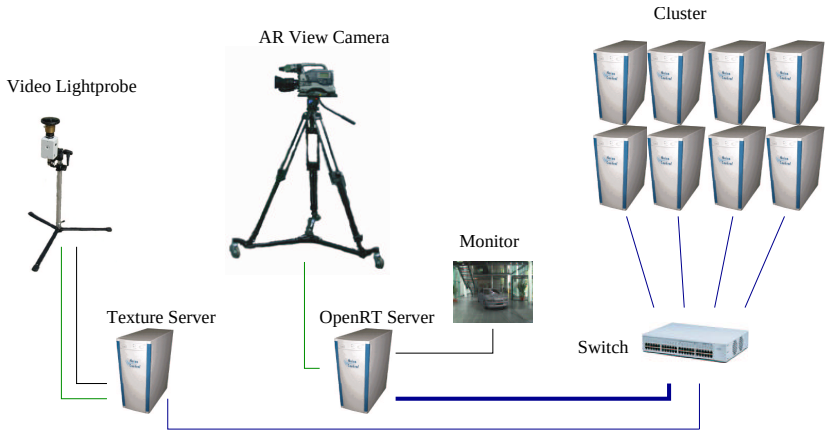



*Figure 7.20: The hardware setup for car IBL example. The real-time lightprobe is connected to a special video texture server to provide shutter control (RS232, black) and video input (green) for HDR reconstruction. The OpenRT server uses a video camera for AR view compositing of the output image.*

### 7.6.2 OpenRT Setup

The OpenRT (software) setup comprises the video texture mechanism and the AR view compositing. The HDR lightprobe image is sampled on the texture server (see next section). The texture server sends the list of samples, and also the full resolution HDR texture image for reflection mapping, to the clients. The *raw format option* (see Section 4.3.4) is used for the sample list. A lightprobe image can bee seen in Figure 7.19.

The OpenRT rendering object manages view compositing and video texture synchronization and is thus a combination of both default rendering objects. The background is tonemapped inversely to allow latter remapping for adjusting the contrast of the virtual car image (see Section 6.4).

### 7.6.3 Light Sample Generation

To light the car and to create a (soft) shadow on the real floor, it is necessary to sample the lightprobe. The number of samples directly affects the rendering speed because each sample causes a shadow ray to be intersected with the the car model. The number of generated samples should hence be easy controllable.

The sampling methods discussed in Section 3.2.4 are not suited for real-time applications because they are too slow due to the iterative optimization methods used.

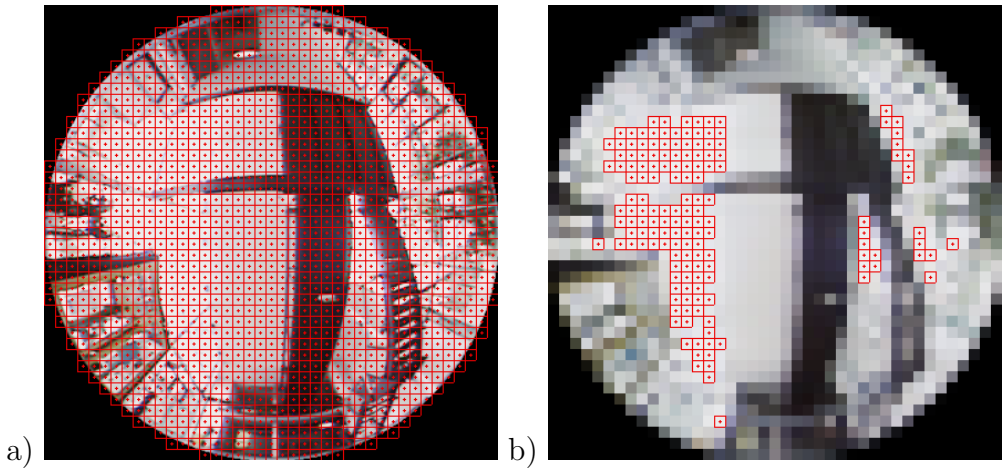


*Figure 7.21: Fast sampling of the real-time lightprobe images.* **a)** *A grid with sample positions for the lightprobe from Figure 7.19. The selected grid width was 10x10 pixels.* **b)** *The set of cells with the average value of the luminance exceeding a given threshold are marked red.*

We opted for a brute-force approach instead. A grid of sampling cells is placed over the lightprobe HDR image (Figure 7.21a). The pixels of each cell are averaged and those cells exceeding a given threshold value are put into a sample list sorted by ascending luminance $Y$ (Figure 7.21b). This yields a list with $N$ samples. The number $N$ depends on the particular lighting situation and the threshold value and is difficult to control when the lighting changes. To achieve a better control, we generate a second list with $M$ samples from the first list. The luminance $Y$ of the HDR values $R$, $G$ and $B$ for sorting is computed by

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \tag{7.13}$$

The number $M$ needs to be high for soft shadows. It should also be guaranteed that important but dimmer secondary lights also get sampled. Sometimes a random sub-list of $M$ samples provides better results than the sorted version.

Since samples at the edge of the disk have a lower impact on the integral lighting (*cosine-law*), the sampling method can be improved by taking the distance of the sample position to the center of the disk into account. This can be done by multiplying the luminance with the appropriate cosine value or by using a variable threshold value.

Using a rectangular filter kernel for averaging on a spheric lightprobe

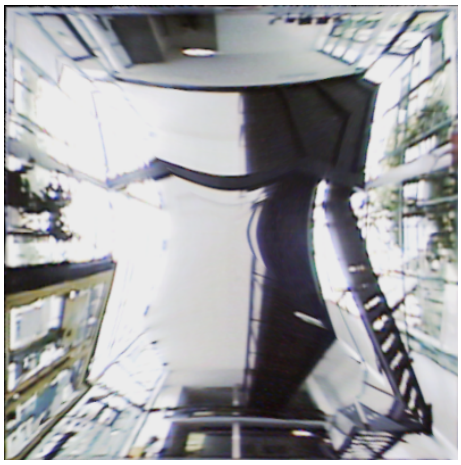
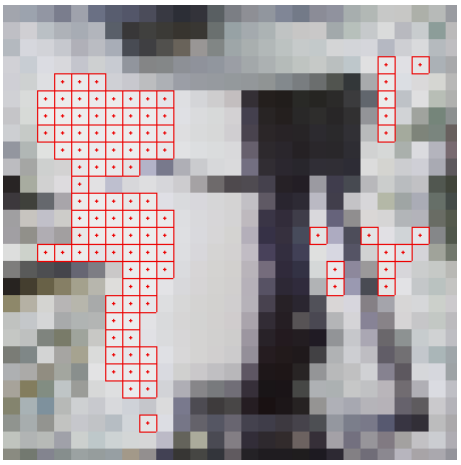
a)  b) 

*Figure 7.22: A warping of the lightprobe from Figure 7.19 using the low-distortion mapping from [Shirley97].* **a)** *The warped lightprobe image.* **b)** *The set of cells with the average value of the luminance exceeding a given threshold are marked red. The sample positions are in the center of the cells.*

image is not correct, of course. An appropriate filtering method should use elliptic, spatially-variant filter kernels. But those are computationally too expensive for real-time applications.

An alternative is to warp the disk image to a rectangular, low-distortion representation. In [Shirley97], Shirley et al. present an appropriate low-distortion warping from a disk to a square (see also [Masselus02]). There is no correct method to warp a lightprobe and to sample it with rectangular filter kernels, but [Shirley97] provides a good approximation. Figure 7.22a shows the warped lightprobe image. In Figure 7.22b the result of the grid sampling method is shown.

Note that the sample optimizing methods of Section 3.2.4 can yield nicer soft shadows with a lower number of samples (and hence a lower number of necessary shadow rays) due to a better spatial distribution of the samples.

### 7.6.4 Shadows and Reflections of Virtual Object in the Video Background

To provide a visual cue of the position of a virtual object inserted into a background, shadows and reflections of the object in the real scene are necessary. Soft shadows can be created using the light samples generated by the sampling of the lightprobe images.

Stand-in (*phantom*) objects with an appropriate shader assigned and a *differential rendering method* are used for representing the real objects of the local scene part (see also Chapter 6). To perform a differential rendering method, the incident light, measured by the real-time lightprobe, must be set in relation to the partial occlusion caused by the virtual objects. In the following, I will derive the necessary equations.

The irradiance $E$ at a point $\vec{x}$ is defined by the integral over the hemisphere of incident light $L$

$$E = \int_{\Omega^+} L(\vec{x}, \vec{\omega}) \cos\Theta \, d\Theta \tag{7.14}$$

With a hemispherical parameterization $\Theta \in [0, \frac{\pi}{2}]$ (elevation) and $\phi \in [-\pi, \pi]$ (azimut) this yields

$$E = \int_{-\pi}^{\pi} \int_{0}^{\frac{\pi}{2}} L(\Theta, \phi) \cos\Theta \sin\Theta \, d\Theta \, d\phi \tag{7.15}$$

For computing the double integral, Equation 7.15 needs to be approximated by discreet sampling over the hemisphere (Section 7.6.3). The number of used samples is $N$. The irradiance $E$ can the be approximated by

$$E = \frac{2\pi}{N} \sum_{i=0}^{N-1} L(\Theta_i, \phi_i) \cos\Theta_i \tag{7.16}$$

When we refer to the direction of incident light $(\Theta_i, \phi_i)$ by a vector $\vec{D}_i$ and take the visibility of the point $\vec{x}$ from that direction into account, this yields

$$E = \frac{2\pi}{N} \sum_{i=0}^{N-1} V(\vec{D}_i) \, L(\vec{D}_i) \, < \vec{n}, \vec{D}_i > \tag{7.17}$$

with $V(\vec{D}_i) \in 0, 1$ and $V(\vec{D}_i) = 0$ if the light direction $\vec{D}_i$ is occluded by a virtual object seen from the surface point $\vec{x}$. $\vec{n}$ is the surface normal at the point $\vec{x}$.

Applying a *differential rendering method* (see Section 6.3) yields the following equation for the intensities of the color channels

$$I'_c = I_c \frac{E_c}{E^0_c} \qquad c \in \{R, G, B\} \tag{7.18}$$

where $I_c$ is the intensity of the video background and $I'_c$ the resulting video output intensity. $E_c$ is the irradiance calculated by Equation 7.17. $E^0_c$ is the *unoccluded* irradiance that we assume to arrive on the floor in the real world (and thus seen by the camera).

Note that this approach is completely independent from the lighting model (or BRDF). We compute both, occluded and unoccluded irradiance in the virtual world assuming an arbitrary BRDF. The BRDF thus cancels out.

When we substitute Equation 7.17 in Equation 7.18, we get

$$I'_c = I_c \frac{\sum_{i=0}^{N-1} L_c(\vec{D})\, V(\vec{D}) \; < \vec{n}, \vec{D} >}{\sum_{i=0}^{N-1} L_c(\vec{D}) \; < \vec{n}, \vec{D} >} \tag{7.19}$$

Since the color channels $c$ have different values due to the color of the incident light, Equation 7.19 cannot be simplified any more. The main cost lies in the term $V(\vec{D})$ since a ray needs to be shot for each occlusion test.

A reflection of the virtual object (the car) on the floor can be easily added by superposition of a specular part to the diffuse lighting. A reflection ray (or multiple rays, if a glossy reflection is wanted) is generated for the appropriate direction. If the ray hits the car, the derived color from the car shader is added to the diffuse intensity $I'$. The same method can be used for caustics.

### 7.6.5 Ambient Occlusion

As a short excursion, I will describe an interesting variation on this method. There's one case where Equation 7.19 can be further simplified: when the incident radiance $L_c$ is equal from all directions. In this case we can assume

$$I' = I \frac{\sum_{i=0}^{N-1} L_c(\vec{D})\, V(\vec{D}) \; < \vec{n}, \vec{D} >}{\sum_{i=0}^{N-1} L_c(\vec{D}) \; < \vec{n}, \vec{D} >} \tag{7.20}$$

yields

$$I'_c = I_c \, L_{const,c} \frac{\sum_{i=0}^{N-1} V(\vec{D})}{\sum_{i=0}^{N-1} 1} \tag{7.21}$$

with a constant incident radiance $L_{const,c}$ per color channel. Further simplified this results in

$$I'_c = I_c \, L_{const,c} \frac{\sum_{i=0}^{N-1} V(\vec{D})}{N} \tag{7.22}$$

or in simple words: the ratio between background video color and output color is the quotient of unoccluded samples to the total number of samples.

The algorithm for this method looks like following:

```
SampleDirectionTable S[N]
integer n = 0
foreach sample s in S[]
        if s is unoccluded then n++
color c = BackgroundColor·(n/N)
```

The assumption of a constant radiance from all directions is useful when no lightprobe is available. In a constant controlled lighting environment — e.g. in a lab setup for an AR simulation of machine part assembly — there is no need to capture the incident light at real-time. Nevertheless such a simple algorithm can enhance the look of such a simulation with soft shadows and hence improve proper recognition of virtual part locations.

This method is a variation of the *Ambient Occlusion* method [Zhukov98]. Methods of this kind are currently quite popular in the industry for generating soft shadows without explicit information on incident light.

### 7.6.6 Lighting the Virtual Objects

To light the virtual objects (the car), the incident light is calculated from the lightprobe and the occlusion by the virtual objects themselves using Equation 7.17. The irradiance $E$ is then used for shading by an appropriate model. Since the material properties of the virtual objects are known (in comparison to the real objects of the background), shading is straightforward.

We used two shaders for the car: one for the metal parts and a glass shader for the windows. The first shader shows just a simple diffuse lighting model (cosine) and a specular reflection derived from the lightprobe video texture used as reflection map. The glass shader adds transparency to this. This could be simply done by blending with the AR background color supplied by the view compositing. But the background view thru the transparent car windows always passes two glass screens. Also, the car's interior can be seen thru a window. The glass shader thus works as following:

```
color c = diffuse(E) + specular(lightprobe)
ray r = transparency_ray()
if(trace(r))
        c = c + r.color
else
        c = c + background_color
return c
```

The (only minimal) refraction of the car windows glass cannot be taken into account because the AR view compositing does not work with redirected rays (Section 6.2).

### 7.6.7 Results

Frames of the resulting output video of the car example can be seen in Figures 7.23 and 7.24. The latter also shows an outdoor lighting situation.

*Figure 7.23: Results of the car example. The left image shows a background video frame, the right image the final composite. Note the soft shadow cast on the real floor.*

Table 7.1 gives the resulting frame rates. For generating the shadows and lighting the car, 32 samples of the lightprobe were used. No warping of the lightprobe image was used.

The main impact on the resulting frame rate is the size of the floor stand-in rectangle. It has to be large enough to capture all effects of a lighting situation that is not known in advance. On the other side, each (primary) ray hitting it causes expensive spawning of 32 occlusion rays for testing the visibility of light directions.

This example application features only a very simple shading model for the car. We didn't experiment with more complex models. A better shading would result in a more convincing look of the car.

Also no perspective matching of the virtual and the real camera was performed. The human eye is very sensitive to slight inaccuracies on perspective. A camera calibration method (see also Section 8.3.1), in combination with an OpenRT camera shader matching the calibration data, should provide a better result.

The main drawback of the application results from the simple lightprobe device. It captures only the upper hemisphere, which causes the real floor

| Scene | Figure | #CPUs | Resolution | fps |
|---|---|---|---|---|
| Car w/o shadow | - | 6 | 640x480 | 5.5 |
| Car w/o shadow | - | 16 | 640x480 | 14 |
| Car w/shadow | 7.23 | 6 | 640x480 | 0.7 |
| Car w/shadow | 7.23 | 16 | 640x480 | 1.8 |
| Car w/shadow | 7.23 | 24 | 640x480 | 2.9 |

*Table 7.1: Some frame rates for the car example. The car model has about 200000 triangles. All measurements were done on Athlon MP 1800+ CPUs.*

not to be reflected in the car. The fish-eye lens could be tilted to provide a look onto the floor, but light from the back would be missed for the shadows and the reflection of the floor would look very strange because of the failing environment map assumption.

Also, a person walking nearby the lightprobe would result in a very large, grotesque reflection on the car. The low frame rate a achievable with the real-time lightprobe of about 1–2 fps HDR is only suitable for slowly changing light conditions.

*Figure 7.24: Two more example frames. The left image shows the car with different lighting and another surface color. The right image gives an impression of an outdoor application with a rather hard shadow cast by the sun.*

## 7.7 Conclusion and Future Work

The car example shows the applicability of interactive ray tracing for a typical, advanced AR application. Ray tracing is ideal for complex light simulation compared to traditional raster graphics and provides a number of benefits like an easy, modular system design.

The presented real-time lightprobe has a number of drawbacks, but provides a good starting point for further experiments. The design of an omnidirectional real-time lightprobe is still a challenging task left for the future. Depending on the application, the design can be adapted for a special purpose. For the car example, a lightprobe design to overcome the distortion of nearby objects in the reflection is desirable. A setup with a number of cameras (Section 7.3.4) rigged on a car-sized mockup seems to be an interesting approach.

On the other side, for small scale rendering like closeup of models in a design-review application, the use of a (tracked) mirror ball in combination with a HDR camera used for AR view *and* to acquire the lightprobe, could be successful. Results from the movie industry also showed that real HDR acquisition is not always necessary for realistic rendering. For simple AR applications, where exact lighting is not important (like manual part assembly simulation), the Ambient Occlusion method (Section 7.6.5) provides a useful option and is simple to implement.

# Chapter 8

# In-Shader Image Based Visual Hull Reconstruction

In Chapter 5, I showed how subjects can be inserted into the virtual world using video billboards. Billboards have a number of drawbacks due to their 2D nature, though (Section 5.5). A real 3D representation of a person, rendered together with the other (3D) geometry of the virtual scene, would provide a much better integration.

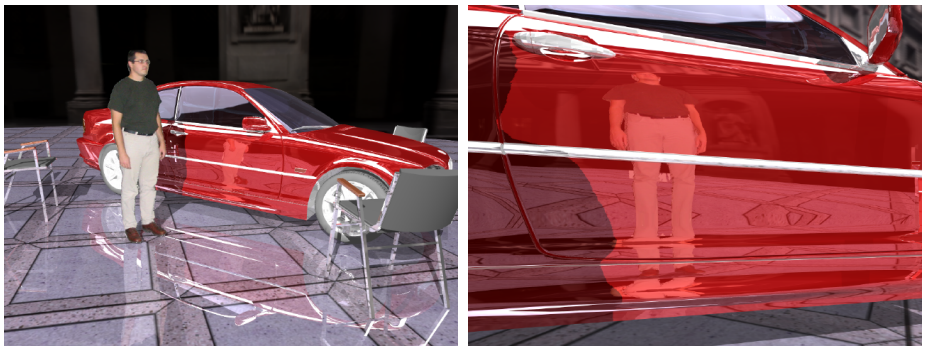


*Figure 8.1: Idea of a visual hull shader. The left image shows a person reconstructed into a VR scene at interactive rates. Correct reflections and shadows provide a seamless integration. The right image shows a closeup of a reflection in the car.*

The simple billboard concept can be extended to 3D: A box is used instead of the flat rectangle and a more complex shader than the billboard shader is attached to the box. In combination with multiple streaming video textures featuring video streams from cameras showing the subject under different

angles, an image-based full 3D reconstruction of a person (or other objects) can be created inside the box (Figure 8.1).

In this chapter, I will provide a background in (interactive) 3D reconstruction methods for actors and introduce the idea of an *image-based visual hull* shader. I will describe the necessary building blocks of a framework for live rendering and supply an OpenRT example implementation of the 3D reconstruction process.

## 8.1 Interactive 3D Reconstruction Methods

A huge number of methods for 3D reconstruction of objects and persons from a number of view images were published in computer graphics and computer vision over the last years. Some of them work in real-time. The methods can be classified by the information in the input images used for reconstruction (*shape-from-x*).

Many methods compute the *visual hull* of an object [Laurentini94], which is basically the 3D intersection of the reprojected silhouettes of the object (*shape-from-silhouette*). For the *photo hull*, a representation more close to the actual shape of the object, photo consistency is taken into account. Here not only the binary silhouette is used, but also the color of the object(*shape-from-texture*).

Other methods remove voxels from a space (like a sculptor) until only the shape of the object remains. This can be based on the silhouette or on the texture (photo-consistency). Examples are *voxel-coloring* [Seitz97] and *space-carving* [Kutulakos98a, Kutulakos98b]. A survey is given in [Slabaugh01].

Nearly all methods are based on two steps: generation of a data structure that represents the subject and using this structure for rendering. The data structures can vary from images over volumetric representations up to true geometry. This concept can be a problem in the demand-driven world of a ray tracer. Often only parts of the reconstructed subject are needed or for a distant view the quality does not need to be the best possible. For a fast interactive method, we do not want to waste CPU time with reconstructing a better model than needed in the actual scene view.

### 8.1.1 Related Work

Giving an extensive survey of (interactive) 3D reconstruction methods is beyond the scope of this thesis. Nevertheless, I want to mention some related work with a special interest in the actor insertion problem and integration

with additional rendering effects. Most of this publications present a complete interactive system.

Among the first interactive systems, and probably the most famous one, is the *image-based visual hull* system of Matusik et al. [Matusik00, Buehler99]. They use 2D rasterization on the view images (hence *image-based*) for the silhouette intersection. The method I present in this chapter is based on this method.

A method for *polyhedral* visual hulls followed [Matusik01]. Here the silhouette is represented by a polygon and the intersection is done geometrically, yielding a real model (a textured polyhedron) of the subject.

The *blue-c* system for virtual collaboration [Gross03] uses an elaborated setup of projectors and cameras to provide a common acting area for several tele-present persons. A 3D representation of the subject based on a point-based variant of [Matusik00] is used [Wuermlin04, Lamboray04].

Li et al. presented a hardware accelerated method for computing a visual hull reconstruction using GPUs [Li03a, Li03b]. A version for computing photo hulls is also available [Li04a].

Hasenfratz et al. describe an actor insertion system based on voxels using GPU hardware for reconstruction [Hasenfratz03, Hasenfratz04]. They also provide (geometrical) interaction examples of the subject with the virtual scene.

A different approach is taken by Theobald et al. [Theobalt03, Theobalt04]. A model of the subject is acquired before and fitted to the actual pose by motion capturing from the camera images. The method is not real-time, but can deliver a smoother and nicer looking subject representation than a real reconstruction method.

## 8.2 Towards a Visual Hull Shader

The basic idea behind a interactive in-shader 3D reconstruction is to extend the billboard concept to 3D. Instead of a flat rectangle, a box is used in the scene description. A special shader is attached to the triangles of the box. Each ray hitting the box invokes a shader call which simulates a volumetric ray traversal inside the box. Video textures (Chapter 4) stream a number of views of an acting area to the shader. The box acts as a 'window' from the virtual scene to this acting area.

Figure 8.2 illustrates the setup. For extracting the actor from the real scene, a segmentation method (see Chapter 5) is used. Because multiple video streams are needed, compression of the image streams on the network is vital.

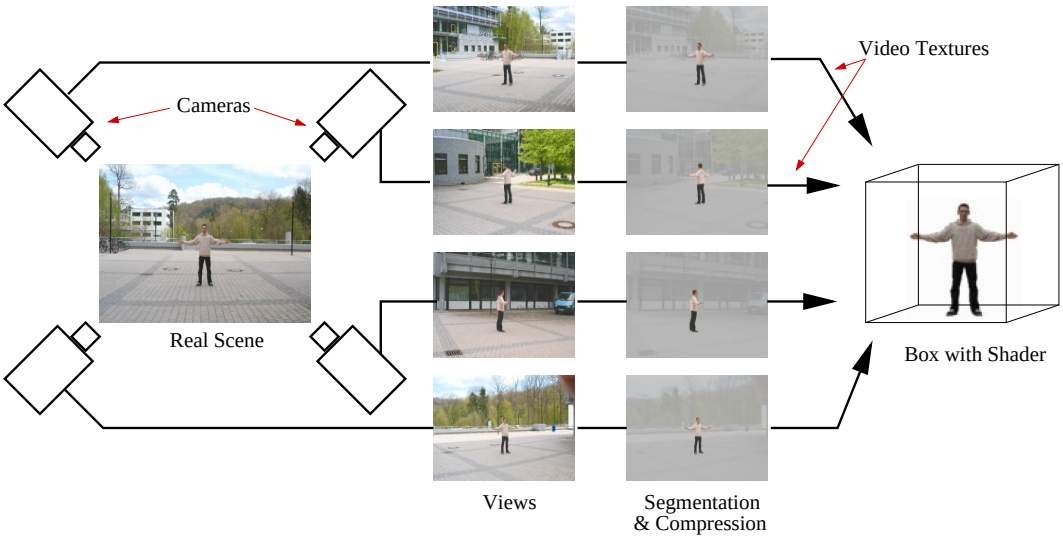



*Figure 8.2: Principle of a 3D reconstruction shader. A subject is captured by several cameras in an acting area. The foreground is segmented and the images are compressed. The video streams are transmitted to the shader assigned to box in the virtual scene.*

The 3D reconstruction algorithm used inside the shader should be carefully chosen to preserve the benefits in a ray tracing framework. The algorithm needs to be

- capable of interactive rates
- demand-driven,
- evaluable for single rays, and
- free of global data structures.

Demand-driven means that a reconstruction should be only invoked if the subject (or an effect like a reflection or a shadow) is visible in the actual view. Evaluation of single (primary, secondary, shadow) rays is necessary to fit into the ray tracing concept. Since shadow rays need less effort (no shading calculation), geometry and shading color should be separately calculated within the algorithm. The reconstruction needs a lot of rendering power, the algorithm has to run in a distributed environment. OpenRT does not support dynamic global data structures for shading.

We opted for the *image-based visual hull* (IBVH) algorithm [Matusik00, Buehler99]. Some modifications provide additional speedup and the (global)

data structure (a view-dependent layered depth image [Shade98], used for later occlusion testing in the shading pass) can be replaced by a compute-on-demand scheme. This also meets the requirements of a ray tracer, where (secondary) rays can origin from arbitrary directions and not only the camera view.

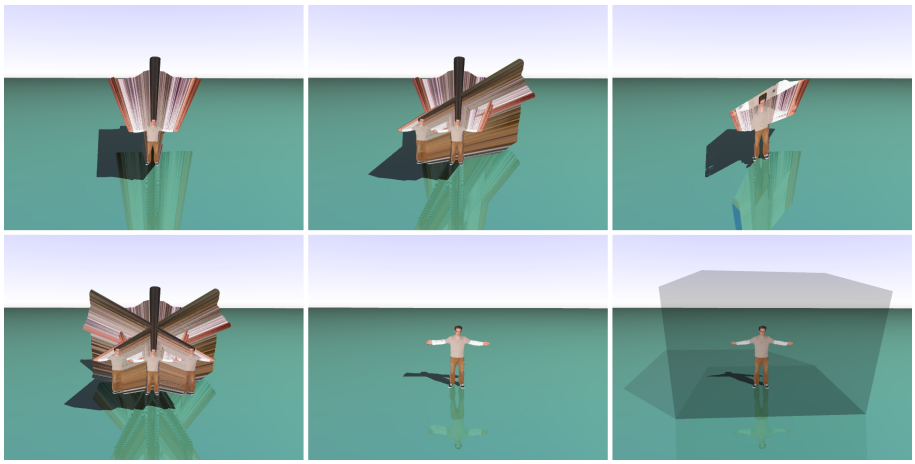


*Figure 8.3: Illustration of the visual hull intersection principle.* **Top row:** *Projection of the view cones. The right image shows the intersection.* **Bottom row:** *The middle image shows the result using three views. In the right image the box object is highlighted.*

Figure 8.3 illustrates the principle of the algorithm. For each view, the silhouette is projected from the camera center into the scene. The cones are intersected and the final intersection forms the resulting visual hull. Figure 8.3 shows that only three views can be sufficient for a good approximation of the subject. In the last image in Figure 8.3, the box object used in the scene description is highlighted to show its size.

## 8.3 Silhouette Acquisition

For live capturing of subjects for 3D reconstruction, an elaborated setup is needed. A number of cameras, observing the acting area, need to be mounted, calibrated, and synchronized. The cameras feed the additional video processing like foreground segmentation and video compression before the silhouettes are sent to the rendering clients.

### 8.3.1 A Calibrated Multi-Camera Setup

The several cameras in a multi-camera setup for interactive 3D reconstruction need to be synchronized in order to acquire a set of images taken at the same point in time.

Conventional analog video cameras can be synchronized by feeding the genlock inputs with the output video signal from one camera. Due to the internal processing speed, the video output is delayed somewhat and the sync phase on the slave cameras need to be adjusted [Poynton03]. A better solution is to use a symmetric feeding of all cameras from a central timebase like a sync generator.

Digital cameras, like the Sony DFW IEEE1394 camera [Sony], can be synchronized externally and triggered by a contiguous pulse signal at the desired frame rate. Appendix B describes the necessary hardware interface (used in [Deutsch02]). Newer IEEE1394 based cameras can often be synchronized using the IEEE1394 bus timing [PointGrey, Anderson98].

Another factor is the effective resolution of the video output. The effective resolution is defined by the number of pixel actually present in the silhouette part. The ratio between foreground and background pixel typically present in a view image is a compromise between camera resolution and acting area size since the cameras are fixed. A high video resolution of the view cameras is thus desired. Typical analog cameras deliver 768x576 pixel, interlaced in two video fields. Since the interlaced fields are acquired at different points in time, only one field can be used for 3D reconstruction. The effective resolution is thus halved in the number of scanlines. Deinterlacing [Poynton03] can be done, but usually just blends the fields (analogous to antialiasing methods) and provides thus only a minor improvement. The use of progressive scan cameras[1] is recommended.

Digital cameras usually provide higher resolutions than 640x480 pixel and also feature progressive scanning. This makes them more interesting for the multi-camera setup purpose. The digital bus system can restrict the mounting possibilities due to maximum cable length, though. Note that the camera bus bandwidth (e.g. IEEE1394 at 400 MBit/s) can limit the frame rate for higher video resolutions.

Besides synchronization, the cameras also need to be setup equally for color and brightness reproduction. Automatic features like auto-exposure and auto-whitebalance have to be switched off. Care should be taken with the lighting of background and subject to ensure proper segmentation and correct exposure.

[1] Some video cameras with analog video output feature both: 50i and 25p.

Another issue is the geometric calibration of the cameras (see also Section 6.1.1). The imaging characteristics of the camera system (sensor and lens) need to be determined (*intrinsic parameters*) to provide a mapping function from a point in 3D to a 2D pixel position in the output image. This is also affected by the camera pose (*extrinsic parameters*). A number of calibrations methods for both parameter sets are available (e.g. [Tsai86]), most work semi-automatic by taking several images of a calibration target. A number of ready-to-use implementations are available (e.g. [OpenCV, Bouguet04]). For 3D reconstruction, the accuracy of the used calibration method is important. Inaccurate calibration results in loss of features of the reconstructed subject.

Special calibration methods for simultaneous calibration of multi-camera setups are available. Here a small target, like a bright LED, is moved around in the view of all cameras to derive a mapping from a point in 3D to all image planes (e.g. [Chen00, Ihrke04]).

### 8.3.2 Foreground Segmentation

To segment the silhouette of the actor from the studio background, a real-time segmentation method is needed. Chroma keying methods are often adopted for this purpose but hard to setup for several simultaneous camera views (Section 5.2.2). Garbage mattes (Section 5.2.1) can help a lot if other cameras and rigs get into the view of a camera. They also reduce processing time since less pixel need to be tested.

A background subtraction (Section 5.2.4) method seems better suited, but is usually less stable due to slight changes in indirect light and other movement in the studio. An optical flow method can provide good results here (e.g. [Bruhn03]).

### 8.3.3 Silhouette Data Compression

Since the available network bandwidth from the video texture servers to the rendering clients is limited, data compression must be used. For the billboard example in Chapter 5 the whole camera image was streamed and segmentation was performed on demand in the shader. For combination with a compression method, it is better to compress the images after segmentation. This results in less data since the background part is not needed. On the other side, the silhouette is not rectangular and the number of foreground pixels varies each frame.

Common schemes as MPEG[2] of JPEG can be adapted. The alternative

[2]Some MPEG version support compression of irregular shaped objects.

is a loss-less compression on the silhouette data. OpenRT provides an compression API to a fast compression library [LZO], which is used for real-time updates of scene data.

The major criteria for choosing a good compression methods is processing speed. It has to be performed in real-time and supply encoding and decoding with an overall speed of more than 20 fps, at least for broadcast needs. Since texture access is demand-driven, evaluation of single texels, directly in the compressed data, would be desirable.

Besides a good compression rate, the compressed data has to meet the requirements of the packetizing schemes for the video textures. Overall compression over the full image is critical if a single, lost network packet can make the whole texture frame useless (see Section 4.3.3).

A scanline based compression approach yields not as good compression rates as a full image method, but provides much better control of texture recovery with lost data packets. This methods also allows faster access to the individual parts of the image. To access a texel, only the current scanline needs to be decompressed (see also Section 8.4.2).

## 8.4 An OpenRT Visual Hull Shader Example

To give an impression of an image-based visual hull shader, I will provide a simple example application in OpenRT [Hoffmann04, Pomi04a]. The focus is held on the reconstruction algorithm and the integration in OpenRT.

### 8.4.1 Data Acquisition

For testing our in-shader reconstruction method, we used still photographs of a subject. Setup of a live capturing system is still complex. Because of our (bad) experience with cameras (Section 7.5.1), we did not want to waste time with setting up a full system and chose a minimal setup with focus on the ray tracing and reconstruction part. Figure 8.4 shows some of the input view photographs. The foreground was segmented manually.

We tested several camera calibration toolkits and calibration methods [OpenCV, ARToolkit, Bouguet04, Linz04]. With all of them we experienced problems with the precision of the calibration parameters. Inexact calibration parameters result in problems with the reconstruction of small features. Often, whole arms or legs of the subject got lost. We finally found a working method based on [Bouguet04] and used a a large calibration target, basically a $1 \times 2$ meter piece of linoleum floor with a (regular) checkerboard pattern ($16.5 \times 16.5$cm squares). For calibration of the intrinsic parameters, about 50

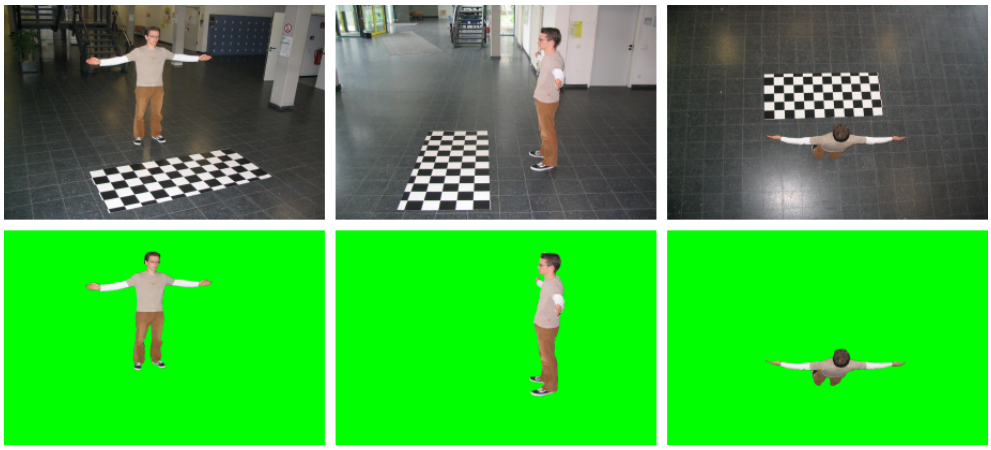


*Figure 8.4: Some of the input view photographs. The original photographs are on top. The linoleum checkerboard target for calibration is clearly visible. The lower row shows the (manually) segmented foreground images used as input for the compression scheme.*

photographs were taken in advance from the target only. [Bouguet04] provides additional output to estimate the reliability of the calculated camera parameters

For the subject views only one photograph is sufficient, provided that most of the target is visible in the image. The subject had to keep in a still pose for several minutes, which is hard, of course. Unavoidable, slight movements of the subject cause loss or merge of smallest features like fingers.

### 8.4.2 The Compression Method

For representing the silhouette we chose a compressed data structure similar to a run-length encoding (RLE) scheme. The data structure allows direct access to single texels without prior decompression of the whole frame and meets thus the demand-driven requirements of a ray tracer.

Figure 8.5 shows the components of the data structure for one frame of a view data stream. A header component contains the resolution of a full view frame and a bounding box for the sub-image actually containing the silhouette. The silhouette is encoded as lists of intervals per scanline. The foreground colors are stored in a single color vector for the whole silhouette. Offset pointers in the interval data structure provide fast access to the color vector. Segmentation and encoding can be done in a single $O(\#pixel)$ pass over the input image. Looking up whether a texel belongs to the silhouette

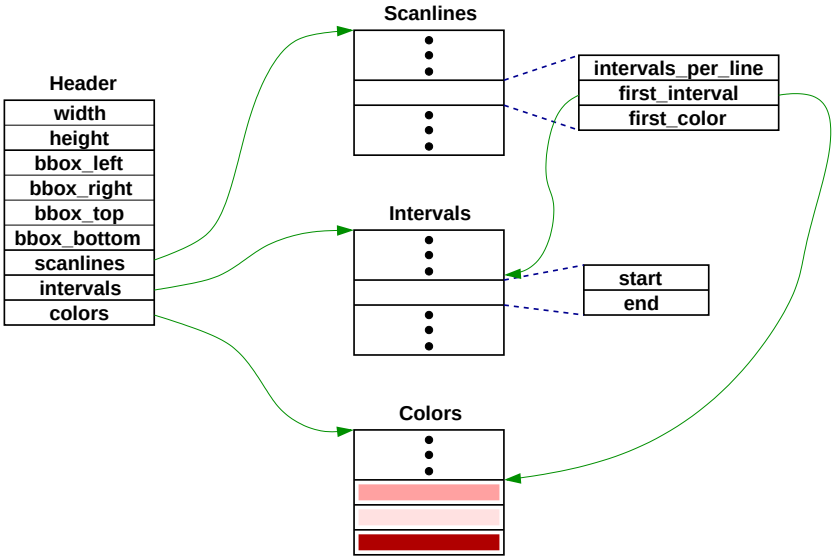



*Figure 8.5: The data structures used for the compressed silhouette data representation. The bounding box accelerates the ray traversal.*

is done by jumping to the requested scanline and searching the intervals. The data structures can be further extended to provide additional data per texel like a blending weight used for view dependent texturing (see [Li04b, Hoffmann04]).

For streaming with video textures the RLE can be reordered to meet the needs of the video texture packetizing scheme. The data for each scanline is then transmitted separately, i.e. a video texture scanline packet consists of the header, a list of the intervals of this scanline and the correspondent part of the color vector.

### 8.4.3 Image Based Ray Traversal

According to [Matusik00, Buehler99], the intersection of a ray with the visual hull is computed in 2D on the view images. The ray segment inside the box object is transformed to the local coordinate system and start and end point are projected into each view. The parametric ray representation allows easy reprojection to 3D when all calculations are done in homogenous coordinates. The ray is rasterized into the view image using a variation of the Cleary-Wyvill algorithm [Cleary88].

The ray intersection is done based on a sorted list of ray intervals. The list contains the values of the ray parameter at the starting and the end

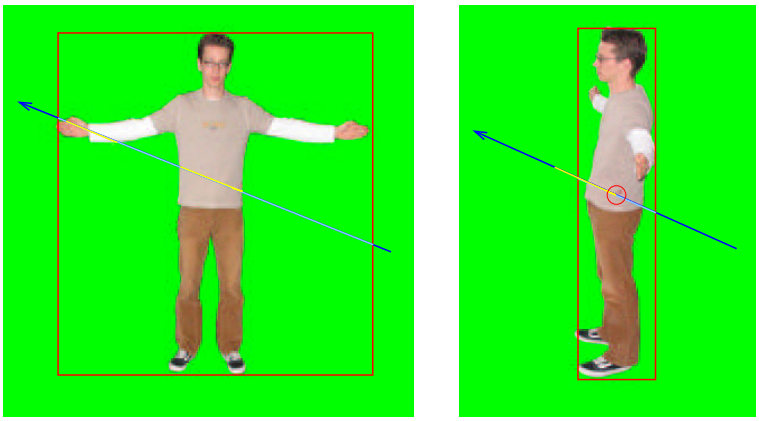


*Figure 8.6: Principle of the visual hull intersection for two silhouette images. The left image shows how a ray (dark blue) gets split into intervals. After clipping to the bounding box (red) of the silhouette (light blue), the intersection of the ray with the silhouette yields two intervals (yellow). In the right image the ray intervals are projected into another view of the subject. After clipping to the bounding box, the yellow interval (which is the projection of the lower interval in the left image) is intersected with the silhouette. The start point marked with the red circle is the nearest intersection point in the remaining interval list.*

point. The initial list contains the entering and the leaving point of the box. For each view, the ray is projected using the camera calibration data. Only the parts with the appropriate ray parameters in the list are rasterized. The intervals are cropped, removed or merged according to the silhouette. Each processed view results in a new interval list. If the list is empty, there is no intersection of the ray with the visual hull and the remaining views need not to be considered. When all views are processed and the list is not empty, the first parameter (the startpoint of the first interval in the sorted list) is the nearest intersection with the visual hull. The point can be reprojected in 3D using the ray parameterization. Figure 8.6 illustrates the method by giving an example using only two view images. The algorithm for the image-based intersection is shown in Figure 8.7.

To improve the performance, we added several details to the basic algorithm from [Matusik00]. Before the ray is rasterized into a view, it is clipped against the bounding box of the silhouette. The bounding box is supplied from the RLE encoding. Only the ray segment near the silhouette is rasterized. This allows for large, sparse input images and for effective processing of a large acting area.

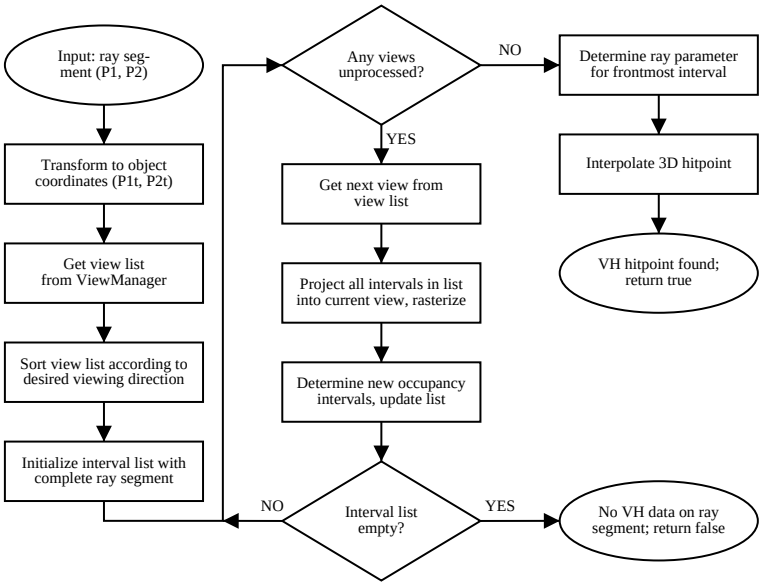



*Figure 8.7: The image-based intersection algorithm for a ray segment with the view images. (Image courtesy of Simon Hoffmann [Hoffmann04])*

We also sort the views for decreasing orthogonality to the ray direction. Using the most orthogonal view first results in fast trimming of the ray parameter range and best precision since the ray parameter is quantized by the rasterization. The reprojected length of a rasterization step onto the ray is maximal for an orthogonal view. The method also yields a low number of processed views for the rejection case due to fast trimming.

Note that the used method can be extended for *photo hulls* [Slabaugh02a, Slabaugh02b], a representation more close to the real subject than the visual hull. The photo hull is based on color consistency rather than binary silhouette intersection.

### 8.4.4 An OpenRT Visual Hull Shader

The intersection method from the previous section can be used as a building block for an OpenRT visual hull shader. The visual hull shader is assigned to a box in the VRML97 scene description. Each ray hitting a face of the box invokes a shader call. The ray is shot from the entering point further inside the box and there will be an intersection with the box itself or with other geometry contained inside the box. Both cases can be distinguished by comparing the shader reference (instance pointer) of the visual hull shader

(C++ *this* pointer) with the shader reference of the hit object. The ray segment between entering point and new hit point is tested with the visual hull intersection method. If there is no visual hull data contained in the ray segment and the ray hits a face of the box object, the ray needs to be traced further behind the box (*transparency ray*). Otherwise the result of the shading process is either the color of the visual hull object, derived by a *view dependent texture mapping* (VDTM) method (see below), or the shading color of other geometry hit in front of the visual hull surface. This allows pixel exact intersection of visual hull data and other ray tracing geometry (see Figure 8.8). Figure 8.9 gives an overview over the algorithm of the visual hull shader.

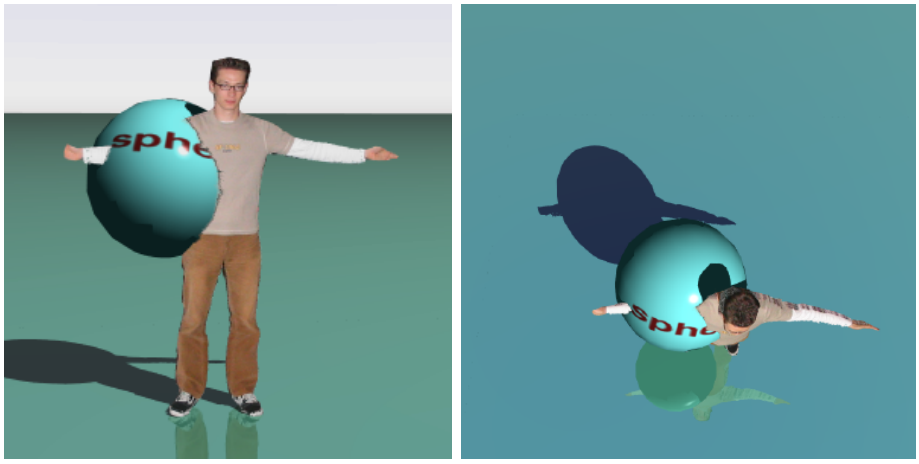


*Figure 8.8: Pixel exact occlusion between the visual hull and other ray tracing objects is achieved by distance comparison of the hitpoint relative to the box intersection point in the visual hull shader.*

The separate shading call for shadow rays in OpenRT (see Chapter 5 and Appendix D) allows a fast solution by performing the visual hull intersection only and neglecting the costly texturing process with additional view occlusion test (see below).

A problem raises when a ray originates inside the box and also hits a (virtual) object inside the box. In this case, the visual hull shader is not invoked, but the visual hull could occlude the ray. The only solution is to observe this case and test the ray segment afterwards against the visual hull. This test is only necessary if the ray origin lies inside the box.

OpenRT does not provide direct access to geometry like a scenegraph library, where an object can be referenced by a name and the object's at-

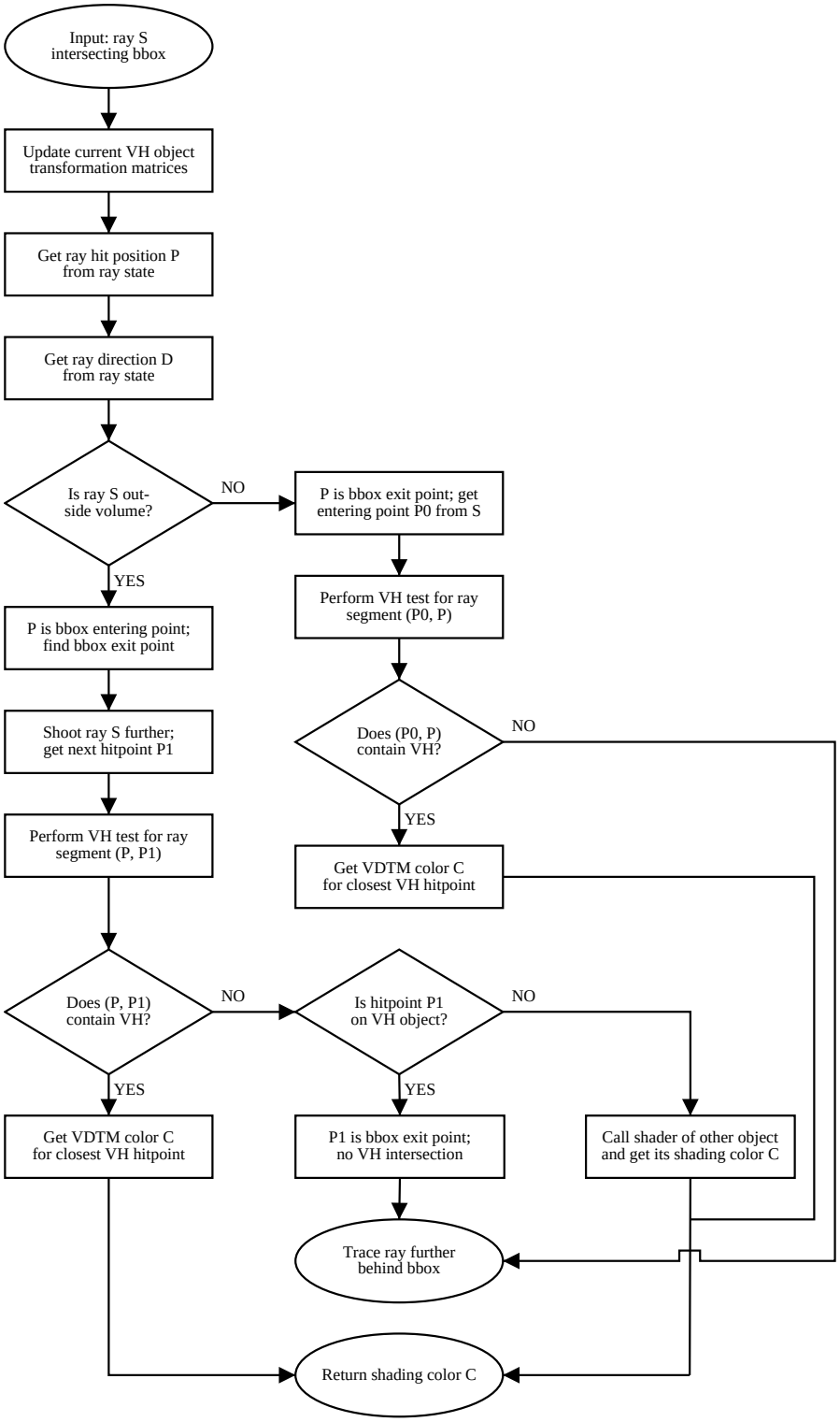



*Figure 8.9: Control flow of the visual hull shader. VDTM: view dependent texture mapping. (Image courtesy of Simon Hoffmann [Hoffmann04])*

tributes can be read out. Thus it is difficult to implement a function for testing whether a certain 3D point is inside the visual hull box object.

We used a trick here. The OpenRT ray data structure is augmented with a flag indicating whether the ray's current origin is inside or outside the box. The flag is updated each time the ray intersects the box and the visual hull shader is called. To determine the initial value (for the primary ray) of the flag, it must be known if the ray tracers' camera is inside or outside the box. This cannot be simply restricted to outside by assuming a small box since the resulting acting area would be very small. Due to the ray projection clipping in Section 8.4.3, the size of the acting area has not much impact on the rendering speed.

To test whether the camera is inside the box, we store the current box object transformation inside the visual hull shader and access it from the rendering object. Each new frame, the camera center point (which is the origin of a primary ray) is tested using the transformation. In the VRML97 scene description the box object is defined as a unity box ($[-0.5, 0.5]^3$ units), scaled and placed by an additional transformation. The point test transforms the point with the object transformation and tests it for inclusion in the unity box. Note that the visual hull box object can be transformed and moved interactively.

All calculations are done in the local coordinate system inside the visual hull box. This coordinate system is defined by the calibration coordinate system and the current transformation of the box object. Since lightsources (and other objects) are defined in scene coordinates, each ray leaving the box must be retransformed.

The concept can be extended to multiple visual hull objects if several shader instances are used to represent a number of subjects independently.

### 8.4.5 View Dependent Texturing

To texture the visual hull surface from the view images, the foreground colors from the several cameras need to be blended to obtain a smooth transition. The diffuse color $C$ can be computed as

$$C = \frac{\sum_{i=0}^{M} V_i \cdot w_i \cdot C_i}{\sum_{i=0}^{M} V_i \cdot w_i} \tag{8.1}$$

where $C_i$ is the color of the texel in the texture of camera $i$, $V_i \in \{0, 1\}$ is the surface visibility of camera $i$, and $w_i$ are the blending weights (*view-dependent texture mapping*, [Debevec98c]). The visibility is important since using the view from a camera that is actually occluded by some part of the

subject can cause *ghosting* artefacts (Figure 8.10). Compared to the two-pass method in [Matusik00, Buehler99], we need to determine the visibility on demand since we do not store any visual hull representation data for occlusion test. The camera occlusion test is done by additional visual hull ray tests. Since the ray test is very costly, only a subset of all cameras is considered for blending. The sorted view list from the traversal algorithm is reused here and only the first $M$ cameras are blended.

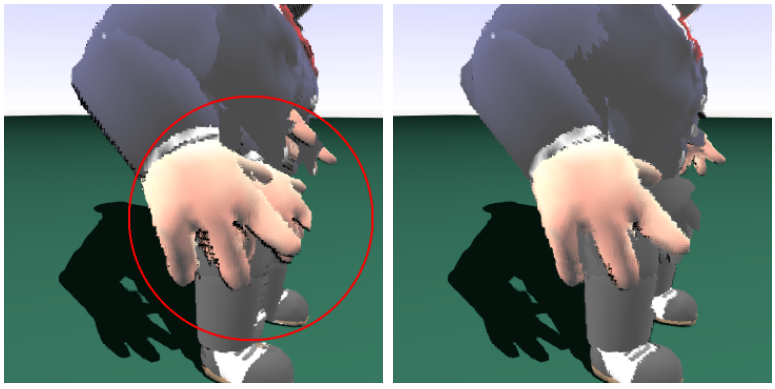


*Figure 8.10: The resulting artefacts when no camera view occlusion test is used. The left image shows the ghosting on the body part when the view occlusion by the arm is not considered. The right image shows the correct rendering with an additional view occlusion test. The object is a reconstruction based on rendered images of a model.*

Choosing good blending weights $w_i$ is a bit difficult. The view direction of the view camera in relation to the virtual camera direction can be used to favor a view that is dominated from the nearest camera direction. This can cause distortion artefacts near the silhouette boundary if the views are far apart. Another possibility is to use the surface normal of the reconstructed visual hull for choosing a view camera. Here the distortion is less, but specular effects caused by the lighting can be completely different. Both information is used in combination for a good compromise [Mukaigawa03]. Further, border areas can be identified using the silhouette information and a distance transformation [Li04b].

The diffuse (texture) color can be modulated by the incident light in the virtual scene for lighting effects. Since no material information (BRDF) for the subject is known, a lighting model must be estimated. A diffuse model is often sufficient. For closeups, an additional specular term for the skin areas can be added (e.g. a Phong highlight). The skin areas can be identified by their typical color.

The incident light is computed by looping over the virtual lightsources in the scene, the usual way in a ray tracing shader. The visual hull also needs to be taken into account for the shadow rays. This concept also allows for *self-shadowing* of the subject, e.g. by casting the shadow of an arm onto the body. The texture should not already contain lighting effects. The studio lighting should be soft and shadowless for later addition of shadows. This also meets the requirements of lighting for the foreground segmentation method.

### 8.4.6 Surface Normals

Sometime, surface normals for assisting in the shading process are desired. Since no geometry is present in the reconstruction process and the evaluation of the visual hull surface is done by discrete sampling points, reconstruction of normals is hard. The use of the (unfiltered) normals of the visual hull geometry is limited because the visual hull is not smooth, at least when only a few views are used for reconstruction. Artefact in shading can be the result. Specular effects on a subject are not possible. Nevertheless, rough normals can assist in the shading process as blending weights for view dependent texturing [Mukaigawa03]. The process of normal computation should be invoked on demand if needed for shading.

We get only one surface point per ray intersection. To compute a normal at this point, the 'neighborhood' in the visual hull must be examined. An obvious method is to intersect additional rays with the visual hull and to use a gradient by fitting a tangent plane to the visual hull surface. For a rough normal estimation, two additional rays, displaced slightly in two directions perpendicular to the ray direction, are sufficient. The two additional surface hits form a triangle with the hitpoint of the original ray. The normal of this triangle can be used as approximation to a surface normal. Problems occur if the additional rays miss the visual hull near the borders. An iterative method with variable displacement values and directions can be used, but would result in a bad performance.

Matusik et al. [Matusik02] describes another method, working directly in the image space of the views. In their reconstruction method, a polygonal representation of the silhouette is given. The interval list structure (Section 8.4.3) is augmented by a pointer to the view responsible for changing the interval start point. The hitpoint on the visual hull is a border point in this view. The normal is defined by the polygon edge and the camera center of the view. Since we have no edge lines in our method, we can compute a 2D normal of a border point by using a $3x3$ Sobel filter kernel (Figure 8.11).

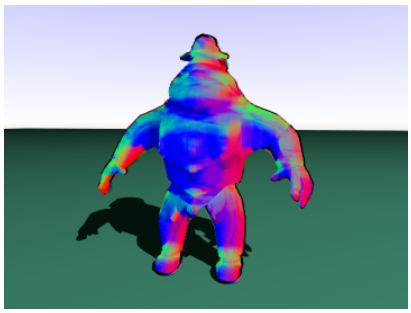


*Figure 8.11: Color coded visual hull surface normals derived with the 2D Sobel filter method. The method is noisy due to the discreet sampling of the visual hull and the limited number of possible values of a small filter kernel used on a binary image.*

### 8.4.7 Results

Example images, rendered using the visual hull shader, can be found in the Figures 8.1, 8.3, 8.8 and 8.12. Table 8.1 shows the according frame rates.

Since the reconstruction process is demand-driven, the frame rate is heavily affected by the number of necessary visual hull intersections and thus the number of pixels showing the subject or its effects (reflection or shadow). For closeups, the frame rate drops to a few frames per second. The setup with three subjects yields about the same frame rate like the one subject example because the number of total visual hull pixels is about the same in the shown images.

We used nine photographs per subject at full resolution of the digital camera (2048 × 1536 pixel). Our experiments showed that smaller view images cause only minimal speedup in reconstruction. It should be considered that the typical foreground/background ratio is about 10–15 percent. The bounding box test provides an immense speedup by achieving a low number of rasterization steps.

Segmentation was performed manually by painting the background green. The RLE encoder performed the chroma test from Section 5.4.4 in a single pass together with the encoding. The whole process of compressing a view frame with 1024×768 pixels takes about 21ms with unoptimized code (Athlon MP 1800+ CPU).

For shading, simple view dependent texturing in combination with a diffuse lighting model (one point lightsource) was used. The additional normal calculation with the Sobel method showed no further impact on the frame rate.

| Scene | Figure | #CPUs | Resolution | fps |
|---|---|---|---|---|
| 1 subject w/o lighting | - | 10 | 320x240 | 34 |
| 1 subject with lighting | 8.8 | 4 | 640x480 | 4.4 |
| 1 subject with lighting | 8.8 | 10 | 640x480 | 10.5 |
| 1 subject with lighting | 8.8 | 16 | 640x480 | 15.5 |
| 1 subject, closeup | 8.12 right | 16 | 640x480 | 2.1 |
| 1 subject and car | 8.1 | 16 | 320x240 | 5.3 |
| 1 subject and car | 8.1 | 16 | 640x480 | 1.6 |
| 1 subject and car | 8.1 | 24 | 320x240 | 9.6 |
| 1 subject and car | 8.1 | 24 | 640x480 | 3.3 |
| 3 subjects and car | 8.12 left | 16 | 640x480 | 2.1 |
| 3 subjects and car | 8.12 left | 24 | 640x480 | 3.6 |

*Table 8.1: Some frame rates for the visual hull shader examples. Nine (eight at 45 degree around and one from above) views were used per subject. View input resolution is* $2048x1536$ *pixel. The speedup with smaller view resolutions is only minimal. All measurements are done on Athlon MP 1800+ CPUs.*

## 8.5 Conclusion and Future Work

The implementation of a visual hull shader for OpenRT showed how simple the inclusion of a reconstruction method can be done in a ray tracing framework. The shader allows reconstruction of subjects (and objects) at interactive frame rates and fits well into the demand-driven concept of a ray tracer. No changes to the existing OpenRT framework are needed. Rendering effects like shadows and reflections come up automatically. The visual hull shader can be seen as a, more evolved, replacement for the billboard shader of Chapter 5. It overcomes all drawbacks of video billboards mentioned there. The price is, of course, the lower rendering speed due to the much more complex calculations for 3D reconstruction.

GPU accelerated methods can yield higher speeds, but are more restricted in terms of integration when it comes to reflections and lighting effects. Since the reconstruction in GPU based methods is view-dependent, the whole calculation needs to be repeated for reflections and shadow maps. Here, the demand-drive concept of ray tracing shows its benefits. No unnecessary calculations are done in a ray tracing framework. The shaders of the surrounding scene need not to know about the special visual hull shader and can treat it like any other shader.

*Figure 8.12: Some more examples of the visual hull shader. The left image shows three persons reconstructed separately using three instances of the visual hull shader. The car model has about 200000 triangles. The right image shows a closeup shot.*

For the future, we still owe a full featured live system with video streaming. A more sophisticated compression scheme is needed to cope with the available network bandwidth. A multi-processor, shared memory machine (e.g. [Altix]) should provide lower video latency and less trouble with the video textures (see Section 4.3.5).

Combination of the visual hull shader with the AR view compositing method of Chapter 6 can provide new applications, like video-based telepresence where a reconstructed subject is rendered into a video background with a real person. The integration of both methods is simple and straightforward. The same holds for an integration with the global illumination system (IGI2, [Benthin03]).

The shading and texturing of the reconstructed visual hull can be improved much more. We only used simple view dependent texture blending, so far. More sophisticated blending schemes will provide better results, in particular for closeups. Our method could also be extended for rendering photo hulls [Slabaugh02a].

Though the concept of a virtual camera, rendering the scene at arbitrary camera positions, provides new possibilities to the studio director, the rendering quality of the blended textures on the visual hull reconstruction do not meet broadcast quality demands [Grau01]. To overcome this problem, a *hybrid* system, which combines the reconstruction with traditional compositing, can be used. The 3D reconstruction of the subject would only be used for rendering the effects like shadows and reflections, while the primary image of the silhouette is taken by a tracked, mobile studio camera and 2D

composited in the traditional way (Chapter 5). A number of fixed, low quality cameras are used for the visual hull reconstruction. The tracked primary camera(s) should not be used for reconstruction since the tracking precision is often too low to meet the demands of the visual hull algorithm.

# Chapter 9

# Final Summary

The aim of this thesis is to prove the applicability of (distributed) interactive ray tracing for mixed reality rendering. On the basis of the OpenRT system, I showed how two extensions (streaming video textures and AR view compositing), build on top of the OpenRT framework and without changes in the existing OpenRT API, enable applications from both ends of the Milgram continuum. I described how existing applications can be improved in terms of rendering quality and realism (lighting from video, video billboards, integrated 3D reconstruction) [Pomi04b]. The ray tracing environment provides a number of benefits, but also some drawbacks today, to the application programmer and to the end user. Just as ray tracing has been the main technology for achieving realistic images over the years, *interactive* ray tracing will be the main source of high quality rendering in real-time applications like VR and MR.

I introduced the concept of *in-shader compositing* which integrates the compositing of real and synthetic elements in MR applications into the shading process. Compared to traditional compositing performed in a (2D) post-process, the in-shader concept provides a number of benefits. Also the overall system design is simpler since there is no need to carry any additional information (like alpha or depth) from the renderer to the post-compositing process. In-shader compositing fits perfectly into a ray tracing framework like OpenRT. Since the programmability of modern GPUs steadily improves, the in-shader compositing concept can also be used in rasterization based systems.

Interactive MR applications, even video-based, often rely on additional hardware, which cannot be pulled off the shelf most of the time. The real-time lightprobe device or a complex multi-camera setup with proper calibration and synchronization are good examples for this. Since computer graphics

is part of the computer science department, people often think computer graphics is only programming (and debugging, of course). But the previous examples show that also engineering skills (mechanical, electrical, optical) are mandatory to build a complete system.

To close this thesis, I will give a brief discussion of the benefits and drawback of using interactive ray tracing in mixed reality applications and provide an outlook to the future.

## Benefits from Interactive Ray Tracing for Mixed Reality Rendering

The modular framework of a ray tracing, which provides independent scene description, primary ray generation (camera), ray shooting and shading, allows an easy and straightforward implementation. The OpenRT API, supporting all of the above areas and being close to OpenGL and RenderMan SL [Apodaca00, Apodaca90], provides additional help for both, porting existing applications and creating new ones.

In a pure software ray tracing environment, the complexity of the shading process is only limited by the computing power of a CPU. Hardware limits and resource conflicts, like on GPUs, are not present. Also the instruction set of a CPU is not specialized compared to GPUs and provide general purpose constructs like loops and branching. This enables the implementation of advances shading methods and global illumination methods, a key technology to photorealistic mixed reality applications. The independence of geometry intersection and shading provides an alternative to the sophisticated multi-pass rendering algorithms still necessary on graphics hardware. Also the evolved programmability of graphics boards (*fragment shaders*) is still ways from the flexibility of software ray tracing.

The demand-driven approach of ray tracing ensures that no more computing power is used as necessary for a specific output. This is in particular valuable for complex and expensive shading methods in MR. The demand-driven 3D visual hull reconstruction is a good example.

Ray tracing is logarithmic in the number of scene primitives, compared to the linear complexity of rasterization (z-buffer) algorithms. This enables the application of larger and more complex models. This is very important for industrial VR/AR applications, since model simplifications, as practiced today to enable rendering of real CAD data, is not necessary any more.

## Drawbacks, at least today...

The main drawback of interactive ray tracing is the low speed (compared to GPUs) in todays implementations. Often single CPUs or small dual- and quad-processor machines are not sufficient to provide a high enough frame rate at the desired image resolution. Distributing the rendering process on a number of machines (cluster) helps for the rendering speed due to the parallel nature of the ray tracing algorithm, but introduces a higher latency at the same time. The higher latency is caused by the (asynchronous) network transfer of the rendering data. For highly interactive applications, in particular with video included in the rendering process, this can be distracting to the user. Also, MR applications increase the overall network load by adding video streaming.

The network also introduces problems in scalability and thus sets a limit for connecting more and more rendering clients to achieve higher frame rates. Multicast provides a scalable alternative, but raises new problems like unreliability causing visible dropouts. Large shared memory machines can provide an alternative and seems to be the only option if reliable and fast video-based rendering is necessary (e.g. virtual TV studio broadcast requirements). It should be noted that distributed graphics hardware systems suffer from the same problems. Here, expensive specialized hardware (e.g. pixel bus [Matysczok04]) is often the only solution.

Ray tracing is not capable of rendering 2D features, like text or lines. These are often used primitives for user interaction and annotations in todays AR applications. Of course, 3D objects can be used instead, but often imply (unwanted) perspective hints. Ray tracing can be easily extended for rendering 2D features by adding traditional rasterization methods. Since the 2D objects needed in AR applications are not complex, software 2D rendering would be sufficient.

## The Future

With the availability of interactive ray tracing, an alternative to traditional raster graphics hardware for interactive applications was provided. Many existing applications could be improved by migration to this new technology, in terms of more realism and quality, but also of software design advantages and modularity of a ray tracer.

It should be noted that most of todays AR applications feature rather simple, flat 2D graphics. This is often due to the limited possibilities of the used rendering techniques. User interaction researchers often state, that

these simple rendering are sufficient, but I believe, that a better rendering quality can improve many applications and can be more appealing. A simple framework to allow the implementation of complex AR/MR applications with a high quality would help here. A middleware, based on OpenRT and specialized for AR/MR applications with a number of additional, modular features (e.g. tracking), can provide an easy starting point for implementation. The adaption of the widely used ARToolkit[1] could be a beginning.

The availability of (inexpensive) shared memory machines with a high number of CPUs in the future will provide the necessary rendering power with a smaller effort (financial, power and space) than todays commodity clusters. OpenRT has already proven its ability to render very huge models with advanced shading effects on shared memory machines (e.g. SGI [Altix]). The problems of video distribution for the implementation of video texture would be much simplified on a shared memory machine. Video dropouts in the textures cannot occur. All other benefits of ray tracing remain, of course.

The SaarCOR [Woop05, Schmittler03] architecture can provide a hardware alternative to pure software ray tracing an a cluster. The prototype promises good scalability and programmable shading at a better quality compared to today GPUs and with a much simpler chip technology, lower clock rate and less power consumption. 'Single chip' ray tracers will provide new ways to image quality in wearable AR technology at low power consumption.

Though OpenRT has a good load balancing to keep all rendering CPUs busy, there's still no mechanism for a guaranteed rendering frame rate. Like with GPU based approaches, rendering speed could be controlled by dynamically changing the effective rendering resolution and scaling the output image afterwards. Guaranteed frame rates are important for interactive applications and vital for real-time systems like virtual TV studios.

The shaders and applications in this thesis do not take fully advantage from today CPU features, in particular the SSE[2] extension (SIMD). Writing shaders with parallel SSE code can provide a maximum rendering power, but also still needs assembler programming. OpenRT already features SSE based ray shooting and an SEE capable shading API extension, but the user needs still to provide his shaders in SSE code. SSE and MMX[3] can also help to speed up additional processing like background segmentation and HDR acquisition.

Most of the MR applications presented in this thesis should only be seen as 'starting points'. Further research is necessary to evolve them to industry

[1] OpenRT does not support the OpenGL modelview/projection matrix model, hence ARToolkit cannot be just compiled with OpenRT.

[2] SSE: Super Scalar Extension.

[3] MMX: Multimedia Extensions: The integer counterpart to floating-point SSE.

applicable systems. Better technologies in the future will make it possible to build better lightprobe devices for acquiring incident light, which is one of the keys to realistic AR rendering. An integration of IBL into the global illumination IGI2 system [Benthin03] based on OpenRT will further simplify building of applications.

Actor insertion based on 3D reconstruction provides an interesting option to traditional systems, but also still a number of technical challenges. The OpenRT visual hull shader needs still to be tested in a large, interactive setup. The nearly automatic integration with all rendering features available in a ray tracer will make this effort rewarding.

# Zusammenfassung

Das Ziel dieser Arbeit war die Untersuchung der Tauglichkeit von (verteiltem) interaktivem Ray-Tracing für Mixed-Reality-Anwendungen. Dies erfolgte anhand des OpenRT Systems.

Neben einer Einführung in die Technik interaktiven Ray-Tracings (Kapitel 2) und einem Überblick über Verfahren zur Erstellung von video-basierten Mixed-Reality-Anwendungen (Kapitel 3) mit Schwerpunkt auf dem Rendering-Vorgang, werden anhand einer Reihe von Beispielanwendungen von beiden Enden des Milgramschen Kontinuums die Möglichkeiten ray-tracing-basierten Mixed-Reality-Renderings illustriert.

Kapitel 5 beschreibt die Problematik des Einfügens (realer) Personen in synthetische Szenen, wie sie etwa in der Technik virtueller Fernseh-Studios auftritt. Ein Video-Billboard Shader ermöglicht eine 2D Integration von Personen in die Ray-Tracing Szene, inklusive der Generierung von ray-tracing typischen Effekten wie Reflektionen und Schatten.

Da Billboards jedoch auch mit Nachteilen, die in ihrem 2D-Charakter begründet liegen, behaftet sind, wird in Kapitel 8 eine 3D Lösung beschrieben. Ein OpenRT-Shader ermöglicht mittels einer bild-basierter Visual-Hull Technik echtes dreidimensionales Compositing. Diese Vorgehensweise erlaubt alle Effekte wie korrekte Reflektionen, Schatten und pixelgenaue Verdeckung.

In Kapitel 7 wird auf das interaktive Einfügen synthetischer Objekte in einen Video-Hintergrund eingegangen. Eine eigens dafür konstruierte Spezial-Kamera erlaubt die Erfassung einer realen Beleuchtungs-Situation. Die von ihr gelieferten Informationen erlauben die realistische Beleuchtung der synthetischen Objekte und die Generierung von (synthetischen) Schatten, die für die Wahrnehmung einer solchen kombinierten Szene wichtig sind.

Um Mixed-Reality-Anwendungen zu unterstützten, wurde das OpenRT System in zwei Punkten erweitert: *Videotexturen* erlauben das Einbeziehen von Live-Bildern der 'echten' Welt direkt in den Rendering-Prozess. Kapitel 4 beschreibt Methoden, einen Video-Datenstrom auf die einzelnen Rendering-Rechner zu verteilen und die Synchronität sicher zu stellen.

Ein weiterer Baustein, speziell für Augmented-Reality-Anwendungen, ist eine verteilte *View Compositing-Methode*, um gerenderte Objekte auf überzeugende Weise in einem Video-Hintergrund mit Hilfe von Schatten und Reflektionen zu verankern (Kapitel 6). Hier kommt ein Differential-Rendering Verfahren zum Einsatz.

Beide Bausteine wurden ohne die OpenRT Bibliothek zu verändern als Plugins realisiert (als sog. OpenRT Rendering-Objekt). Eine Verwendung der beschriebenen Methoden in Verbindung mit Graphik-Hardware (GPU) basierten Rendering-Methoden ist ebenfalls denkbar.

Ein zentrales Konzept dieser Arbeit ist das *In-Shader Compositing*. Hierbei werden die notwendigen Berechnungen um reale und synthetische Elemente zu verbinden im Rahmen des Shading-Prozesses mitberechnet. Ein nachträglicher Compositing-Schritt entfällt. Gerade in Verbindung mit einem Ray-Tracer hat In-Shader Compositing eine Reihe von Vorteilen, so wird zum Beispiel die Qualität des Compositing automatisch dem jeweiligen Bildausschnitt angeglichen.

Diese Arbeit bestätigt auch, dass sich das Erstellung von Mixed-Reality-Anwendungen nicht nur auf Software beschränkt, sondern auch umfangreiche Kentnisse und Verständnis einer ganzen Reihe anderer Technologien, wie zum Beispiel Kameratechnik oder Netzwerktechnik, notwendig ist.

## Schlussfolgerung

Diese Arbeit zeigt, dass interaktives Mixed-Reality-Rendering auf der Basis von OpenRT möglich ist und im Vergleich zu (Raster-) Graphik-Hardware basierten Methoden Vorteile hat. Diese liegen in der unbegrenzten Flexibilität der reinen Software-Implementierung von OpenRT und der hohen Modularität des Ray-Tracing Verfahrens begründet.

Die Nachteile der Verwendung von interaktivem Ray-Tracing für Mixed-Reality-Anwendungen liegen in den derzeit geringeren erzielbaren Bildraten im Vergleich zu Graphik-Hardware. Auch ist der Aufwand des verteilten Renderns im Vergleich zu einer einzelnen Graphikkarte ungleich höher. Preiswertere Shared-Memory-Maschinen können hier in Zukunft Abhilfe schaffen. Latenzzeiten und Netzwerkprobleme würden damit auf ein Minimum reduziert werden.

# Appendix A

# The CTools Suite

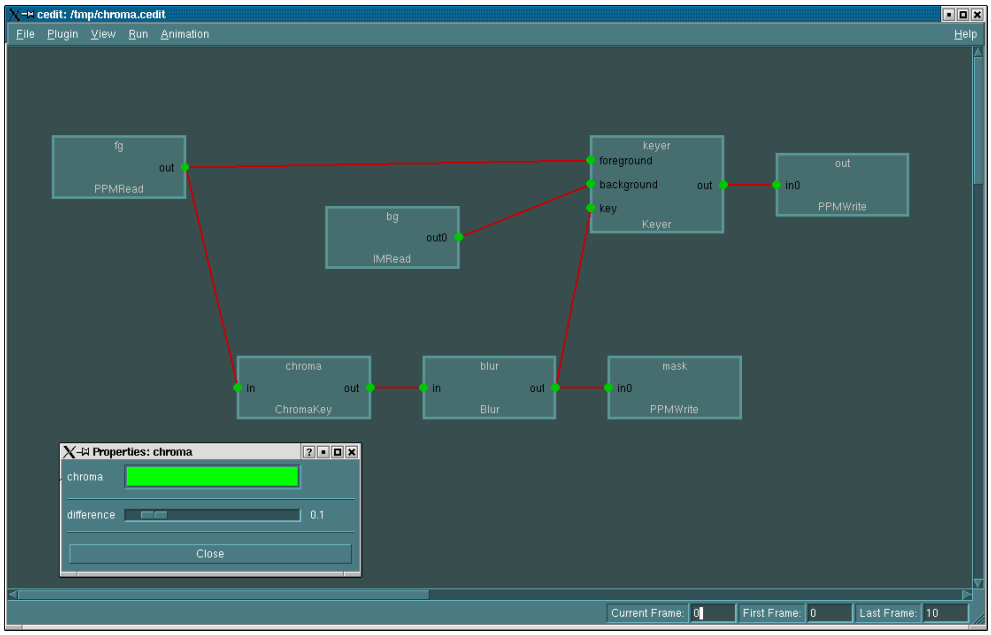



*Figure A.1: The* `CEdit` *GUI front-end allows easy creation and manipulation of CTools graphs.*

To simplify testing of image and video manipulation algorithms like chroma keying and background subtraction (see Section 5.2) or high dynamic range image sampling (see Section 7.6.3), I developed a framework inspired by graph based systems used in the creative industry for film and video compositing.

The CTools (for *Compositing Tools*) suite consists of a library and two front-end applications: a Qt [Qt] based graph editor (`CEdit` (see Figure A.1) and a batch processing command line tool (`CRun`). The library provides functions for connecting nodes (plugins) as separately compiled shared libraries, controls the graph data flow of image buffers and allows to save and load the graph to disk. The versatile image buffer supports a number of encoding formats up to floating point buffers for high dynamic range image manipulation. The buffer access mechanism in the plugins provides a transparent interface to write generic algorithm running with the same code on LDR or HDR data. Plugins export their parameters including range description to allow automatic generation of a GUI dialog for the plugin. Support for a number of parameter types (color, integer, float, mode, etc.) is provided. The graph can be run iteratively for video sequences.

# Appendix B

# A Triggering Interface for Sony DFW Cameras

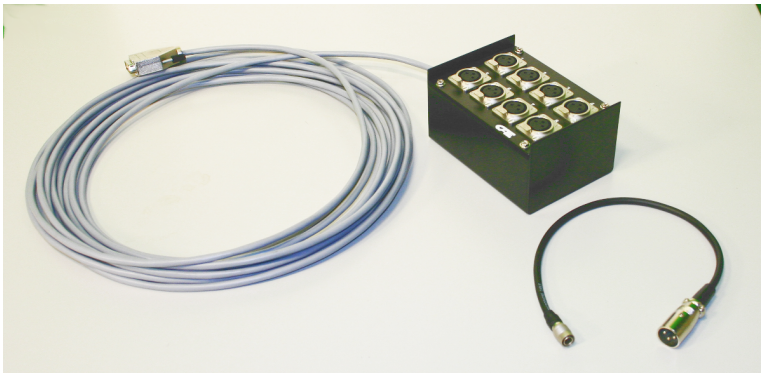


*Figure B.1: A simple home-brew triggering box for Sony DFW (and other) cameras built by the author.*

For simultaneous capturing of images with a number of cameras (e.g. for the 3D Visual Hull reconstruction described in Chapter 8), it is necessary to ensure that all cameras expose at the same time. While some newer IEEE1394 bus cameras provide bus support for synchronous contiguous run modes (running at a fixed frame rate, typically only a small number of fixed rates are available, e.g. 30 fps, 15 fps, 7.5 fps, 3.75 fps), it is often desirable to run at an frame rate that adapts closely to the processing speed of the reconstruction algorithm.
The Sony DFW-V500 (and many comparable cameras) provides an external hardware input for triggering. A simple hardware interface allows to trigger

the cameras at speeds up to 30 fps from a PC. Figure B.1 shows my trigger box for up to eight cameras. It is connected to the parallel port (LPT) of the PC. Writing a single byte to the parallel port allows to trigger a single camera or a group of cameras dependent on the bits set in the byte. A Linux device driver provides an alternative `ioctl` based interface. The cameras are connected via standard XLR extension cables. The interface is opto-insulated.

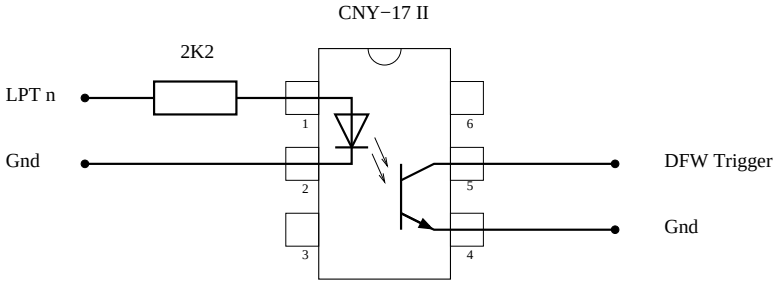



*Figure B.2: The schematics of one of the eight trigger circuits. An opto-coupler (CNY-17 II) provides opto-insulation for minimizing ground loop interference via the IEEE1394 cabling.*

# Appendix C

# The OpenRT Video Texture API

The OpenRT video texture API is an extension to the OpenRT shading API (see [OpenRT]). In the following, I give a brief overview over the most important API calls for shader programming. An example shader is given in Appendix D. Please refer to Section 4.3 for an overview of the architecture of the video texture subsystem.

```
static VTex *VTexManager::openStream(uint id)
static VTex *VTexManager::openFile(char *path)
```

The function can be called directly on the (singleton) `VTexManager` class. The return value is a pointer to a `VTex` video texture object. `openStream(uint id)` opens a multicast stream. The appropriate multicast group is joined if it is not already available on the client host. `openFile(char *path)` opens a local file in a custom video texture file format (a header and one or multiple texture images) for testing purpose. If a stream or file is currently not available, these functions return a `NULL` pointer.

```
RTuint VTex::getWidth()
RTuint VTex::getHeight()
```

Called on a video texture object, these functions return the width and height of the texture image.

```
R3 VTex::getTexel(RTuint u, RTuint v)
R3 VTex::getTexelNearest(RTFloat u, RTFloat v)
R3 VTex::getTexelLinear(RTFloat u, RTFloat v)
```

To access a texel of the current texture frame, the function `getTexel(RTuint u, RTuint v)` provides direct access. For convenience two functions with texture filtering (nearest texel and linear interpolation) with $u, v \in [0, 1)$ are available.

```
RTvoid *VTex::getBuffer()
```

To use the video texture system for streaming arbitrary data structures like the sample tables in Section 7.6, a number of functions provide a direct access to the raw data. More advanced functions can cope with the network packet loss problem to ensure the integrity of the data.

# Appendix D

# An OpenRT Video Billboard Shader Example

In the following, I will give a commented OpenRT shader code example: the chroma keying billboard shader from Section 5.4. The OpenRT video texture API calls are explained in Appendix C. Please see the OpenRT API reference [OpenRT] for an explanation of the OpenRT calls.

```
#include "OpenRTS/RTS.hxx"
#include "OpenRTS/RTS++.hxx"
#include "OpenRTS/RTShader.hxx"
#include "VTex/vtex.hxx"
```

Some OpenRT standard include files. The file `vtex.hxx` contains the OpenRT video texture API calls.

```
class ChromakeyBillboardShader:public RTShader
{
  int id;
  float chromascale;
```

The local variables: `id` is the video texture stream id and `chromascale` is the scaling factor for chroma keying (see Section 5.4.4).

```
public:

  virtual RTvoid Register()
  {
    Init();
```

```
    RTShader::Register();

    rtsDeclareParameter("texture_id", PER_SHADER,
                        offsetof(RTChromaBillboardShader, id),
                        sizeof(id));

    rtsDeclareParameter("chromascale", PER_SHADER,
                        memberoffset(chromascale),
                        sizeof(chromascale));
  }
```

The shader parameters are bound to the local variables.

```
  virtual RTvoid Shade(RTState *const state)
  {
    R3 texcoord;
    rtsFindST(state, texcoord);
```

The 2D texture coordinates for the hitpoint are determined.

```
    VTex *vtex = VTexManager::openStream(id);
    if(!vtex) return;
```

The reference to the desired video texture stream is obtained. If the stream is currently not in use, `NULL` is returned.

```
    R3 color = vtex->getTexelLinear(texcoord.s, texcoord.t);
    float green = color.g * chromascale;
```

The texture color is interpolated from the video texture. The green value is scaled accordingly to provide a more stable chrome keying criteria (see Section 5.4.4).

```
    if(green > color.r && green > color.b)
    {
      RTState transparence;
      rtsInitState(state, &transparence);
      rtsTransparencyRay(state, &transparence);
      rtsTrace(&transparence, color);
    }

    rtsReturnColor(state, color);
  }
```

The chroma keying background condition is evaluated. If it holds, a transparency ray is shot and the color of the hit object is obtained. If the condition fails, the video texture color is preserved since this is a foreground pixel.

```
  virtual bool LightTransparency(RTState *const state)
  {
    R3 texcoord;
    rtsFindST(state, texcoord);

    VTex *vtex = VTexManager::openStream(id);
    if(!vtex) return;

    R3 color = vtex->getTexelLinear(texcoord.s, texcoord.t);
    float green = color.g * chromascale;

    if(green > color.r && green > color.b)
      return false;
    else
      return true;
  }
}
```

The `LightTransparency()` function is called for shadow rays. For foreground pixel in the video texture it returns `true`, which means the billboard is opaque at this point.

```
rtDeclareShader(ChromakeyBillboardShader, ChromakeyBillboard);
```

Finally an export name for the shader is declared. The shader can be used in a VRML file by the name `ChromakeyBillboard`. The shader can be compiled separately or together with other shaders in a shared library. The full shader name is defined as `shadername@library.so`.

# Appendix E
# The Studio Lab

*Figure E.1: The (Mixed Reality) Studio Lab at the Computer Graphics Department at Saarland University. A high ceiling headroom, a mounting grid for lighting fixtures, cameras and chroma green backdrops simplify mixed reality experiments.*

Since I was the first PhD student when the new Computer Graphics Department at Saarland University was opened in October 1999, I had the chance to set up a new Studio Lab for mixed reality experiments and demos. After a search of more than one year and a lot of political and bureaucracy obstacles, a suitable place was found in a corner of the entrance hall of the computer science building. A newly constructed wall separates the lab from the rest of the hall. The high ceiling headroom of 3.7m allows to hang lighting fixtures. The studio walls and ceiling were painted dark gray for control of spill light. A lighting grid consisting of 50mm aluminium tubes was installed under the ceiling and supports hanging of lighting fixtures, cameras and backdrops.

For lighting we choose cool fluorescent fixtures (NesyFlex [Nesys]) and a number of small fresnel spots (Desisti Magis and Dedolights) since the studio has no air conditioning. Green chroma keying fabric [Gerriets] for backdrops and floor is available. A JVC GY-DV500 professional video camera, a video rack with monitors, video recorders and scan conversion for VGA, various mounting/grip equipment, plenty of power connections and a screen projection make the Studio Lab a universal place for demos and experiments.
A special thanks goes to Simon Hoffmann and Stefan Schüffler who helped me a lot in building and equipping the Studio Lab.

*Figure E.2: A sphere rendered into the studio with the method described in Section 7.6.*

# Bibliography


[3DV] *http://www.3dvsystems.com.* 3DV Systems Technology, Israel.

[Agarwal03] *Sameer Agarwal, Ravi Ramamoorthi, Serge Belongie, and Henrik Wann Jensen.* Structured Importance Sampling of Environment Maps. *ACM Transactions on Graphics*, Vol. 22, No. 3, pages 605–612, July 2003.

[Altix] *http://www.sgi.com/products/servers/alitx/350.* SGI Altix 350 Server, Silicon Graphics Inc., US.

[Anderson98] *Don Anderson. FireWire(R) System Architecture: IEEE 1394A.* Addison-Wesley, Second edition, 1998.

[Apodaca90] *Anthony A. Apodaca and M. W. Mantle.* RenderMan: Pursuing the Future of Graphics. *IEEE Computer Graphics & Applications*, Vol. 10, No. 4, pages 44–49, July 1990.

[Apodaca00] *Anthony A. Apodaca and Larry Gritz. Advanced RenderMan. Creating CGI for Motion Pictures.* Morgan Kaufmann, First edition, 2000.

[Appel68] *Arthur Appel.* Some Techniques for Shading Machine Renderings of Solids. *SJCC*, pages 27–45, 1968.

[ARToolkit] *http://www.hitl.washington.edu/artoolkit/.* ARToolkit Library, Washington University, US.

[Azuma95] *Ron Azuma.* A survey of augmented reality. *Computer Graphics (SIGGRAPH'95 Proceedings, Course Notes #9: Developing Advanced Virtual Reality Applications)*, pages 1–38, 1995.

[Azuma01] *Ronald Azuma, Yohan Baillot, Reinhold Behringer, Steven Feiner, Simon Julier, and Blair MacIntyre.* Recent Advances in Augmented Reality. *IEEE Computer Graphics Applications*, Vol. 21, No. 6, pages 34–47, 2001.

[Battiato03] *Sebastiano Battiato, Alfio Castorina, and Massimo Mancuso.* High dynamic range imaging for digital still camera: an overview. *Journal of Electronic Imaging*, Vol. 12, No. 3, pages 459–469, July 2003.

[Bayer76] *Bryce E. Bayer.* Color imaging array. US Patent 3971065, 1976.

[BBC RD] *http://www.bbc.co.uk.* British Broadcast Company, Research and Development Department, UK.

[Bekaert01] *Philippe Bekaert.* Extensible Scene Graph Manager, August 2001. http://www.cs.kuleuven.ac.be/∼graphics-/XRML/.

[Ben-Ezra00] *Moshe Ben-Ezra.* Segmentation with invisible keying signal. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 568–575, 2000.

[Benthin03] *Carsten Benthin, Ingo Wald, and Philipp Slusallek.* A Scalable Approach to Interactive Global Illumination. *Computer Graphics Forum*, Vol. 22, No. 3, pages 621–630, 2003. (Proceedings of Eurographics).

[Benthin04] *Carsten Benthin, Ingo Wald, and Philipp Slusallek.* Interactive Ray Tracing of Free-Form Surfaces. In *Proceedings of Afrigraph 2004*, pages 99–106, November 2004.

[Bergh99] *Frans van den Bergh and Vali Lalioti.* Software Chroma Keying in an Immersive Virtual Environment. In *SAICSIT'99, Annual Conference.* South African Institute of Computer Scientists and Information Technologies, 1999.

[BFI] *http://www.bfioptilas.de.* Optical Technology Supply, BFI Optilas, Germany.

[Bimber03a] *Oliver Bimber, Anselm Grundheimer, Gordon Wetzstein, and Sebastian Knodel.* Consistent Illumination within optical see-through augmented environments. In *Proceed-*

*ings of IEEE/ACM International Symposium aon Augmented and Mixed Reality 2003 (ISMAR'03)*, pages 1–8. ACM Press, 2003.

[Bimber03b] *Oliver Bimber and Ramesh Raskar.* Alternative Augmented Reality Approaches: Concepts, Techniques, and Applications. Eurographics Conference 2003, Tutorial, 2003.

[Bouguet04] *http://www.vision.caltech.edu/bouguetj/calib_doc/.* Camera Calibration Toolbox for Matlab by Jean-Yves Bouguet.

[Box99] *Harry C. Box. The Gaffer's Handbook – Film lighting practices, equipment and electrical distribution.* Focal Press, Second edition, 1999.

[Brinkmann99] *Ron Brinkmann. The Art and Science of Digital Compositing.* Morgan Kaufmann Publishers Inc., First edition, 1999.

[Bruhn03] *Andres Bruhn, Joachim Weickert, Christian Feddern, Timo Kohlberger, and Christoph Schnörr.* Real-Time Optic Flow Computation with Variational Methods. In *Computer Analysis of Images and Patterns, Proceedings 10th International Conference CAIP 2003*, pages 222–229. Lecture Notes in Computer Science, Vol. 2756, Springer, Berlin, August 2003.

[Buehler99] *Chris Buehler, Wojciech Matusik, Leonard McMillan, and Steven Gortler.* Creating and Rendering Image-Based Visual Hulls. Technical Report MIT/LCS/TR-780, Massachusetts Institute of Technology, 1999.

[Carey97] *Rikk Carey, Gavin Bell, and Chris Marrin.* ISO/IEC 14772-1:1997 Virtual Reality Modelling Language (VRML97), April 1997. http://www.vrml.org/Specifications/VRML97.

[Chen00] *Xing Chen, James Davis, and Philipp Slusallek.* Wide Area Camera Calibration Using Virtual Calibration Objects. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2000*, pages 520–527, 2000.

[Chuang01] *Yung-Yu Chuang, Brian Curless, David H. Salesin, and Richard Szeliski.* A Bayesian Approach to Digital Matting. In *2001 Conference on Computer Vision and Pattern Recognition (CVPR 2001)*, pages 264–271, December 2001.

[Chuang02] *Yung-Yu Chuang, Aseem Agarwala, Brian Curless, David H. Salesin, and Richard Szeliski.* Video Matting of Complex Scenes. *ACM Transactions on Graphics*, Vol. 21, No. 3, pages 243–248, July 2002.

[Cleary88] *John G. Cleary and Geoff Wyvill.* An Analysis of an Algorithm for Fast Ray-Tracing using Uniform Space Subdivision. *The Visual Computer*, Vol. 4, No. 2, pages 65–83, 1988.

[Cook84] *Robert Cook, Thomas Porter, and Loren Carpenter.* Distributed Ray Tracing. *Computer Graphics (Proceeding of SIGGRAPH 84)*, Vol. 18, No. 3, pages 137–144, 1984.

[Debevec] *http://www.debevec.org.* Paul Debevecs Website at the ICT, US.

[Debevec97] *Paul E. Debevec and Jitendra Malik.* Recovering High Dynamic Range Radiance Maps from Photographs. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 369–378, August 1997.

[Debevec98a] *Paul Debevec.* Rendering Synthetic Objects Into Real Scenes: Bridging Traditional and Image-Based Graphics With Global Illumination and High Dynamic Range Photography. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 189–198, July 1998.

[Debevec98b] *Paul Debevec.* Rendering with natural light. In *SIGGRAPH '98: ACM SIGGRAPH 98 Electronic art and animation catalog*, page 166. ACM Press, 1998.

[Debevec98c] *Paul E. Debevec, Yizhou Yu, and George D. Borshukov.* Efficient View-Dependent Image-Based Rendering with Projective Texture-Mapping. In *Eurographics Rendering Workshop 1998*, pages 105–116, June 1998.

[Debevec00] *Paul Debevec, Tim Hawkins, Chris Tchou, Haarm-Pieter Duiker, Westley Sarokin, and Mark Sagar.* Acquiring the Reflectance Field of a Human Face. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 145–156, July 2000.

[Debevec02a] *Paul Debevec.* Image-Based Lighting. *IEEE Computer Graphics & Applications*, Vol. 22, No. 2, pages 26–34, March/April 2002.

[Debevec02b] *Paul Debevec, Andreas Wenger, Chris Tchou, Andrew Gardner, Jamie Waese, and Tim Hawkins.* A Lighting Reproduction Approach to Live-Action Compositing. *ACM Transactions on Graphics*, Vol. 21, No. 3, pages 547–556, July 2002.

[Debevec02c] *Paul Debevec, Andreas Wenger, Chris Tchou, Andrew Gardner, Jamie Waese, and Tim Hawkins.* A Lighting Reproduction Approach to Live-Action Compositing. *ACM Transactions on Graphics*, Vol. 21, No. 3, pages 547–556, July 2002.

[Deutsch02] *Benjamin Deutsch.* Image Based Visual Hull Rendering. Diploma Thesis, Saarland University. March 2002.

[Dietrich03] *Andreas Dietrich, Ingo Wald, Carsten Benthin, and Philipp Slusallek.* The OpenRT Application Programming Interface – Towards A Common API for Interactive Ray Tracing. In *Proceedings of the 2003 OpenSG Symposium*, pages 23–31, Darmstadt, Germany, 2003. Eurographics Association.

[Dietrich04a] *Andreas Dietrich, Ingo Wald, and Philipp Slusallek.* Interactive Visualization of Exceptionally Complex Industrial Datasets. In *ACM SIGGRAPH 2004, Sketches and Applications*, August 2004.

[Dietrich04b] *Andreas Dietrich, Ingo Wald, Markus Wagner, and Philipp Slusallek.* VRML Scene Graphs on an Interactive Ray Tracing Engine. In *Proceedings of IEEE VR 2004*, pages 109–116, March 2004.

[Dosch] *http://www.doschdesign.com.* Libraries of textures, environments, HDR lightprobes and models, Dosch Design Gmbh, Germany.

[DP97] *DP.* Virtuelle Studiotechnik – Detaillierte technische Übersicht. Digital Production, August 1997.

[DP00a] *DP.* Kollision zweier Welten – Virtuelle Studios von GMD am Beispiel 'Service Wohnen'. Digital Production, May 2000.

[DP00b] *DP.* Virtual Studio Equipment. Digital Production, May 2000.

[Drettakis97a] *George Drettakis, Luc Robert, and Sylvain Bougnoux.* Interactive Common Illumination for Computer Augmented Reality. In *Eurographics Rendering Workshop 1997*, pages 45–56, June 1997.

[Drettakis97b] *George Drettakis and Francois Sillion.* Interactive update of global illumination using a line-space hierarchy. *Computer Graphics*, Vol. 31, No. Annual Conference Series, pages 57–64, 1997.

[Exif] *http://www.exif.org.* Exchangeable image file format for Digital Still Cameras: EXIF, Japan Electronic Industry Development Association (JEIDA), Japan.

[FourCC] *http://www.fourcc.org.* Video Codec and Pixel Format Information.

[Fournier93] *Alain Fournier, Atjeng S. Gunawan, and Chris Romanzin.* Common Illumination between Real and Computer Generated Scenes. In *Proceedings of Graphics Interface '93*, pages 254–262, Toronto, ON, Canada, May 1993.

[Gerriets] *http://www.gerriets.com.* Stage and Studio Fabrics, Gerriets GmbH, Germany.

[Gibson00] *Simon Gibson and Alan Murta.* Interactive Rendering with real world illumination. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 365–376, June 2000.

[Gibson03a] *Simon Gibson and Alan Chalmers.* Photorealistic Augmented Reality. Eurographics Conference 2003, Tutorial, 2003.

[Gibson03b] *Simon Gibson, Jon Cook, Toby Howard, and Roger Hubbold.* Rapid Shadow Generation in Real-World Lighting Environments. In *Eurographics Symposium on Rendering: 14th Eurographics Workshop on Rendering*, pages 219–229, June 2003.

[Glassner95] *Andrew S. Glassner. Principles of digital image synthesis*, volume 1+2. Morgan Kaufmann Publishers, Inc., First edition, 1995.

[Goral84] *Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile.* Modeling the Interaction of Light Between Diffuse Surfaces. *Computer Graphics (SIGGRAPH'84 Proceedings)*, Vol. No. 18, pages 212–222, July 1984.

[Gortler96] *Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen.* The Lumigraph. *Computer Graphics*, Vol. 30, No. Annual Conference Series, pages 43–54, 1996.

[Grau01] *Oliver Grau, Marc Price, and Graham A. Thomas.* Use of 3-D Techniques for Virtual Production. pages *Invited paper presented at SPIE conference on Videometrics and Optical Methods for 3D Shape Measurement*, 2001.

[Greene86] *Ned Greene.* Environment mapping and other applications of world projections. *IEEE Computing Graphics Applications*, Vol. 6, No. 11, pages 21–29, 1986.

[Greger98] *Gene Greger, Peter Shirley, Philip M. Hubbard, and Donald P. Greenberg.* The Irradiance Volume. *IEEE Computer Graphics and Applications*, Vol. 18, No. 2, pages 32–43, March 1998.

[Gross03] *Markus Gross, Stephan Würmlin, Martin Naef, Edouard Lamboray, Christian Spagno, Andreas Kunz, Esther Koller-Meier, Tomas Svoboda, Luc Van Gool, Silke Lang, Kai Strehlke, Andrew Vande Moere, and Oliver Staadt.*

blue-c: A Spatially Immersive Display and 3D Video Portal for Telepresence. In *Proceedings of ACM SIGGRAPH 2003*, pages 819–827, 2003.

[Guenther04] *Johannes Guenther, Ingo Wald, and Philipp Slusallek.* Realtime Caustics using Distributed Photon Mapping. In *Rendering Techniques 2004, Proceedings of the Eurographics Symposium on Rendering*, pages 111–121, June 2004.

[Haeberli93] *Paul Haeberli and Mark Segal.* Texture Mapping As A Fundamental Drawing Primitive. In Michael F. Cohen, Claude Puech, and Francois Sillion, editors, *Fourth Eurographics Workshop on Rendering*, pages 259–266, 1993.

[Haller03] *Michael Haller, Stephan Drab, and Werner Hartmann.* A real-time shadow approach for an augmented reality application using shadow volumes. In *VRST '03: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 56–65. ACM Press, 2003.

[Hanrahan91] *Pat Hanrahan, D. Salzman, and L. Aupperle.* A rapid hierarchical radiosity algorithm. *Computer Graphics*, Vol. 24, No. 4, pages 197–206, 1991.

[Hasenfratz03] *Jean-Marc Hasenfratz, Marc Lapierre, Jean-Dominique Gascuel, and Edmond Boyer.* Real-Time Capture, Reconstruction and Insertion into Virtual World of Human Actors. In *Vision, Video and Graphics*, pages 49–56. Eurographics, Elsevier, 2003.

[Hasenfratz04] *Jean-Marc Hasenfratz, Marc Lapierre, and Francois Sillion.* A Real-Time System for Full Body Interaction. *Virtual Environments*, pages 147–156, 2004.

[HDRC] *http://www.hdrc.com.* HDRC High Dynamic Range CMos Video Camera, IMS Vision GmbH, Germany.

[HDRShop] *http://www.debevec.org/hdrshop.* HDRShop, ICT, US.

[Heidrich98] *Wolfgang Heidrich and Hans-Peter Seidel.* View-independent environment maps. In *SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 39–46, August 1998.

[Heidrich99] *Wolfgang Heidrich*. High-quality Shading and Lighting for Hardware-Accelerated Rendering. PhD thesis, Universität Erlangen, Germany, 1999.

[Hoffmann03] *Simon Hoffmann*. Building a High Dynamic Range Video Camera. Fortgeschrittenenpraktikum, Saarland University. July 2003.

[Hoffmann04] *Simon Hoffmann*. Interactive Reconstruction and Rendering of Image-Based Visual Hulls in a Distributed Ray Tracing Framework. Diploma Thesis, Saarland University. December 2004.

[Hughes04a] *Charles E. Hughes, Jaakko Konttinen, and Sumanta Pattanaik*. The Future of Mixed Reality: Issues in Illumination and Shadows. In *Proceedings of I/ITSEC 2004, Orlando*, December 2004.

[Hughes04b] *Charles E. Hughes, Erik Reinhard, Jaakko Konttinen, and Sumanta Pattanaik*. Achieving Interactive-time Realistic Illumination in Mixed Reality. In *Proceedings of the 24th Army Science Conference (Poster), Orlando*, November 2004.

[IEEE1394] *http://www.ieee1394.org*. The IEEE1394 Interface Standard (aka Apple FireWire or Sony i-Link).

[Ihrke04] *Ivo Ihrke, Lukas Ahrenberg, and Marcus Magnor*. External camera calibration for synchronized multi-video systems. *Journal of WSCG*, Vol. 12, No. 1–3, pages 537–544, January 2004.

[Jacobs04] *Katrien Jacobs and Celine Loscos*. Classification of Illumination Methods for Mixed Reality. In *Eurographics State of the Art Reports*, Grenoble, September 2004.

[Jai] *http://www.jai.com*. Video cameras, JAI Camera Solutions, Japan.

[Jensen96] *Henrik Wann Jensen*. Global Illumination using Photon Maps. In Xavier Pueyo and Peter Schröder, editors, *Eurographics Rendering Workshop 1996*, pages 21–30. Eurographics, Springer, June 1996.

[Kang03] *Sing Bing Kang, Matthew Uyttendaele, Simon Winder, and Richard Szeliski.* High Dynamic Range Video. *ACM Transactions on Graphics*, Vol. 22, No. 3, pages 319–325, July 2003.

[Keith96] *Jack Keith. Video Demystified. A Handbook for the Digital Engineer.* Harris Semiconductor, Second edition, 1996.

[Keller97] *Alexander Keller.* Instant Radiosity. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 49–56. ACM SIGGRAPH, Addison Wesley, August 1997.

[Keller98] *Alexander Keller.* Quasi-Monte Carlo Methods for Realistic Image Synthesis. PhD thesis, University of Kaiserslautern, 1998.

[Kelly00] *Doug Kelly. Digital Compositing. In Depth.* Coriolis, 2000.

[Kollig03] *Thomas Kollig and Alexander Keller.* Efficient illumination by high dynamic range images. In *Proceedings of the 14th Eurographics workshop on Rendering*, pages 45–50. Eurographics Association, 2003.

[Kutulakos98a] *Kiriakos Kutulakos and Steven M. Seitz.* What Do $n$ Photographs Tell Us About 3D Shape? Technical Report 692, Computer Science Department, University of Rochester, NY, US, May 1998.

[Kutulakos98b] *Kiriakos N. Kutulakos and Steven M. Seitz.* A Theory of Shape by Space Carving. Technical Report TR692, Computer Science Department of University of Rochester, 1998.

[Lafortune93] *Eric Lafortune and Yves Willems.* Bidirectional Path Tracing. In *Proceedings of the 3rd International Conference on Computational Graphics and Visualization Techniques (Compugraphics)*, pages 145–153, 1993.

[Lamboray04] *Edouard Lamboray, Stephan Würmlin, and Markus Gross.* Real-time Streaming of Point-based 3D Video.

In *Proceedings of the IEEE Virtual Reality 2004 Conference*, pages 91–98. IEEE Computer Society Press, 2004.

[Laurentini94] *Aldo Laurentini*. The Visual Hull Concept for Silhouette Based Image Understanding. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 16, No. 2, pages 150–162, 1994.

[Lee] *http://www.leefilters.com*. Lee Gel Filters, UK.

[Lensch01] *Hendrik P. A. Lensch, Jan Kautz, Michael Goesele, Wolfgang Heidrich, and Hans-Peter Seidel*. Image-based reconstruction of spatially varying materials. Technical Report MPI-I-2001-4-001, Max-Plank-Institut für Informatik, Saarbrücken, Germany, 2001.

[Levoy96] *Marc Levoy and Pat Hanrahan*. Light Field Rendering. *Computer Graphics*, Vol. 30, No. Annual Conference Series, pages 31–42, 1996.

[Li03a] *Ming Li, Marcus Magnor, and Hans-Peter Seidel*. Hardware-Accelerated Visual Hull Reconstruction and Rendering. pages *Proceeding of Graphics Interface (GI'03), Halifax, Canada*, June 2003.

[Li03b] *Ming Li, Marcus Magnor, and Hans-Peter Seidel*. Online Accelerated Rendering of Visual Hulls in Real Scenes. *Journal of WSCG*, Vol. 11, No. 2, pages 290-297, 2003.

[Li04a] *Ming Li, Marcus Magnor, and Hans-Peter Seidel*. Hardware-Accelerated Rendering of Photo Hulls. *Proceeding of Eurographics (EG'04) Grenoble, France*, pages 635–642, September 2004.

[Li04b] *Ming Li, Marcus Magnor, and Hans-Peter Seidel*. A Hybrid Hardware-accelerated Algorithm for High Quality Rendering of Visual Hulls. *Proceedings of Graphics Interface (GI'04)*, pages 41–48, May 2004.

[LightGen] *http://www.ict.usc.edu/~jcohen/lightgen/lightgen.html*. The LightGen Plugin for HDRShop, US.

[Linux1394] *http://www.linux1394.org*. Linux IEEE1394, IEEE1394 Support for Linux.

[Linz04] *Christian Linz.* Implementing a camera calibration toolbox for Linux. Fortgeschrittenenpraktikum, Saarland University. July 2004.

[Liu03] *X.Q. Liu and Abbas El Gamal.* Synthesis of High Dynamic Range Motion Blur Free Image From Multiple Captures. *IEEE Transactions on circuits and systems (TCASI)*, Vol. 50, No. 4, pages 530–539, April 2003.

[Lohse02] *Marco Lohse, Michael Repplinger, and Philipp Slusallek.* An Open Middleware Architecture for Network-Integrated Multimedia. In *Protocols and Systems for Interactive Distributed Multimedia Systems, Joint International Workshops on Interactive Distributed Multimedia Systems and Protocols for Multimedia Systems, IDMS/PROMS 2002, Proceedings*, volume 2515 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 2002.

[Loscos99] *Céine Loscos, Marie-Claude Frasson, George Drettakis, Bruce Walter, Xavier Granier, and Pierre Poulin.* Interactive Virtual Relighting and Remodeling of Real Scenes. In *Eurographics Rendering Workshop 1999*, June 1999.

[Luther98] *Arch C. Luther. Video Camera Technology.* Artech, First edition, 1998.

[LZO] *http://www.dogma.net/DataCompression/LZO.shtml.* LZO-Compression Library by Markus Oberhume.

[Madden93] *Brian Madden.* Extended intensity range imaging. Technical Report MS-CS-93-96, University of Pennsylvania, GRASP Laboratory, 1993.

[Mann94] *Steve Mann and Rosalind W. Picard.* On being 'undigital' with digital cameras: Extending Dynamic Range by Combining Differently Exposed Pictures. Technical report, M.I.T. Media Lab Perceptual Computing Section (also IS&T's 48th Annual Conference, Society for Imaging Science and Technology, pages 422-428, Washington D.C., May 1995), 1994.

[Mann96] *Steve Mann.* 'Pencigraphy' with AGC: Joint parameter estimation in both domain and range of functions in

same orbit of the projective-Wyckoff group. In *IEEE International Conference on Image Processing (ICIP-96)*, September 1996.

[Mann97] *Steve Mann and Rosalind W. Picard.* Video Orbits of the Projective Group: A Simple Approach to Featureless Estimation of Parameters. *IEEE Transactions on Image Processing*, Vol. 6, No. pages 9, September 1997.

[Mantiuk04] *Rafal Mantiuk, Grzegorz Krawczyk, Karol Myszkowski, and Hans-Peter Seidel.* Perception-motivated high dynamic range video encoding. *ACM Transactions on Graphics*, Vol. 23, No. 3, pages 733–741, 2004.

[Marmitt04] *Gerd Marmitt, Heiko Friedrich, Andreas Kleer, Ingo Wald, and Philipp Slusallek.* Fast and Accurate Ray-Voxel Intersection Techniques for Iso-Surface Ray Tracing. In *Proceedings of Vision, Modeling, and Visualization (VMV)*, November 2004.

[Marschner97] *Stepeh R. Marschner and Donald P. Greenberg.* Inverse lighting for photography. In *Proceedings of the IS&T/SID Fifth Color Imaging Conference*, Society for Imaging Science and Technology, pages 262–265, 1997.

[Masselus02] *Vincent Masselus, Philip Dutré, and Frederik Anrys.* The Free-form Light Stage. In *Rendering Techniques 2002: 13th Eurographics Workshop on Rendering*, pages 247–256, June 2002.

[Masselus03] *Vincent Masselus, Pieter Peers, Philip Dutré, and Yves D. Willems.* Relighting With 4D Incident Light Fields. *ACM Transactions on Graphics*, Vol. 22, No. 3, pages 613–620, July 2003.

[Matusik00] *Wojciech Matusik, Chris Buehler, Ramesh Raskar, Steven J. Gortler, and Leonard McMillan.* Image-Based Visual Hulls. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 369–374. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[Matusik01] *Wojciech Matusik, Chris Buehler, and Leonard McMillan.* Polyhedral Visual Hulls for Real-Time Rendering. *Proceedings of Eurographics Workshop on Rendering*, pages 115–126, 2001.

[Matusik02] *Wojciech Matusik, Chris Buehler, Leonard McMillan, and Steven Gortler.* Efficient View-Dependent Sampling of Visual Hulls. pages *MIT LCS Technical Memo 624*, 2002.

[Matysczok04] *Carsten Matysczok and Andrew Wojdala.* Rendering of Highly Polygonal Augmented Reality Applications on a Scalable PC-Cluster Architecture. In *IEEE and ACM International Symposium on Mixed and Augmented Reality ISMAR 2004*, pages 254–255, November 2004.

[Milgram94] *Paul Milgram and Fumio Kishino.* A taxonomy of mixed reality visual displays. In *IEICE Transactions on Information Systems*, volume 12, December 1994.

[Mitsunaga99] *Tomoo Mitsunaga and Shree K. Nayar.* Radiometric Self Calibration. In *Proccedings on Computer Vision and Pattern Recognition, Vol. I*, pages 374–380, 1999.

[Moeller99] *Tomas Moeller and Eric Haines. Real-Time rendering.* AK Peters, Ltd., First edition, 1999.

[Mukaigawa03] *Yasuhiro Mukaigawa, Daisuke Genda, Ryo Yamane, and Takeshi Shakunaga.* Color Blending based on Viewpoint and Surface Normal for Generating Images from Any Viewpoint using Multiple Cameras. *Proceedings of the IEEE Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI2003)*, pages 95–100, 2003.

[Nayar00] *Shree K. Nayar and Tomoo Mitsunaga.* High Dynamic Range Imaging: Spatially Varying Pixel Exposures. In *CVPR*, pages 1472–1479, 2000.

[Nelson99] *Brad Nelson and Philipp Slusallek.* Virtual Light Meter. Project Sketch, Standford Immersive Television Project, Stanford, 1999.

[Nesys] *http://www.nesys.de.* NesyFlex Fluorescent Lighting Fixtures, Nesys Lichtsysteme, Germany.

[Nielsen01] *Frank Nielsen*. On Representing Spherical Videos. IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Technical sketch, 2001.

[Nielsen02] *Frank Nielsen*. High Resolution Full Spherical Videos. In *Proceedings of ITCC2002, International Conference on Information Technology: Coding and Computing*, pages 260–267, 2002.

[Nikon] *http://www.nikon.com*. Cameras and accessories, Nikon Inc., Japan.

[Nishino04] *Ko Nishino and Shree K. Nayar*. Eyes for relighting. *ACM Transactions on Graphics*, Vol. 23, No. 3, pages 704–711, 2004.

[NMM] *http://www.networkmultimedia.org*. NMM, Network Integrated Multimedia Middleware for Linux, Saarland University, Germany.

[Ohta99] *Yuichi Ohta and Hideyuki Tamura. Mixed Reality - Merging Real and Virtual Worlds*. Springer (Berlin), 1999.

[OpenCV] *http://www.intel.com/research/mrl/research/opencv/*. Intel Open Source Computer Vision Library (OpenCV), Intel Corp., US.

[OpenEXR] *http://www.openexr.net*. OpenEXR File Format, Industrial Light and Magic, US.

[OpenGL] *http://www.opengl.org*. The OpenGL Specification, SGI, US.

[OpenRT] *http://www.openrt.org*. The OpenRT Documentation.

[Orad] *http://www.orad.com*. Orad Virtual Studio Technology, Israel.

[Patow03] *Gustavo Patow and Xavier Pueyo*. Inverse Rendering Problems. *Computer Graphics Forum*, Vol. 22, No. 4, pages 663–687, December 2003.

[Peters03] *Benjamin Peters*. High Quality Chroma Keying. Fortgeschrittenenpraktikum, Saarland University. October 2003.

[PointGrey] *http://www.ptgrey.com.* Digital Cameras and Modules, Point Grey Research Inc., US.

[Pomi99] *Andreas Pomi.* Dynamisches Radiosity mit Line-Space-Hierarchie und Movement Prediction. Diploma Thesis. Technical University of Darmstadt, Germany. January 1999.

[Pomi03] *Andreas Pomi, Gerd Marmitt, Ingo Wald, and Philipp Slusallek.* Streaming Video Textures for Mixed Reality Applications in Interactive Ray Tracing Environments. In *Proceedings of Virtual Reality, Modelling and Visualization (VMV)*, pages 261–270. AKA, Berlin, November 2003.

[Pomi04a] *Andreas Pomi, Simon Hoffmann, and Philipp Slusallek.* Interactive In-Shader Image-Based Visual Hull Reconstruction and Compositing of Actors in a Distributed Ray Tracing Framework. In *1. Workshop VR/AR, Chemnitz, Germany*, September 2004.

[Pomi04b] *Andreas Pomi and Philipp Slusallek.* Interactive Mixed Reality Rendering in a Distributed Ray Tracing Framework. In *IEEE and ACM International Symposium on Mixed and Augmented Reality ISMAR 2004, Student Colloquium*, November 2004.

[Porter84] *Thomas Porter and Tom Duff.* Compositing digital images. In *In Proceedings of the 11th Annual International Conference on Computer Graphics and Interactive Techniques*, pages 253–259, 1984.

[Poynton03] *Charles Poynton. Digital Video and HDTV. Algorithms and Interfaces.* Morgan Kaufmann, First edition, 2003.

[Press99] *William H. Press, Saul A. Teukolsky, William T. Vetterl, and Brian P. Flannery. Numerical Recipes in C. The Art of Scientific Computing.* Cambridge University Press, Second edition, 1999.

[Purcell02] *Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan.* Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics*, Vol. 21, No.

3, pages 703–712, 2002. (Proceedings of SIGGRAPH 2002).

[Qt] *http://www.trolltech.com*. The Qt Grahics User Interface Toolkit, Trolltech Inc., Oslo, Norway.

[Ramamoorthi01a] *Ravi Ramamoorthi and Pat Hanrahan*. An Efficient Representation for Irradiance Environment Maps. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 497–500, August 2001.

[Ramamoorthi01b] *Ravi Ramamoorthi and Pat Hanrahan*. A Signal-Processing Framework for Inverse Rendering. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 117–128, August 2001.

[Raskar01] *Ramesh Raskar, Greg Welch, Kok-Lim Low, and Deepak Bandyopadhyay*. Shader Lamps: Animating Real Objects With Image-Based Illumination. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 89–102. Springer-Verlag, 2001.

[Repplinger01] *Michael Repplinger*. L1394-Library: Design and Implementation of a FireWire Library for Linux. Fortgeschrittenenpraktikum, Saarland University. May 2001.

[Robertson99] *Mark A. Robertson, Sean Borman, and Robert L. Stevenson*. Dynamic Range Improvement Through Multiple Exposures. In *Proceedings of the IEEE International Conference on Image Processing*, volume 3, pages 159–163. IEEE, October 1999.

[Sato99a] *Imari Sato, Yoichi Sato, and Katsushi Ikeuchi*. Acquiring a Radiance Distribution to Superimpose Virtual Objects onto a Real Scene. *IEEE Transactions on Visualization and Computer Graphics*, Vol. 5, No. 1, pages 1–12, 1999.

[Sato99b] *Imari Sato, Yoichi Sato, and Katsushi Ikeuchi*. Illumination Distribution from Shadows. In *CVPR*, pages 1306–1312, 1999.

[Schmittler02] *Jörg Schmittler, Ingo Wald, and Philipp Slusallek.* SaarCOR – A Hardware Architecture for Ray Tracing. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 27–36, 2002.

[Schmittler03] *Jörg Schmittler, Alexander Leidinger, and Philipp Slusallek.* A Virtual Memory Architecture for Real-Time Ray Tracing Hardware. *Computer and Graphics, Volume 27, Graphics Hardware*, pages 693–699, 2003.

[Schmittler04a] *Jörg Schmittler, Tim Dahmen, Daniel Pohl, Christian Vogelgesang, and Philipp Slusallek.* Ray Tracing for Current and Future Games. In *Proceedings of 34. Jahrestagung der Gesellschaft für Informatik*, pages 149–153, 2004.

[Schmittler04b] *Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang J. Paul, , and Philipp Slusallek.* Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In *Proceedings of Graphics Hardware*, pages 51–65, 2004.

[Schoedl00] *Arno Schoedl, Richard Szeliski, David H. Salesin, and Irfan Essa.* Video Textures. In *Proceedings of SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 489–498. ACM Press, August 2000.

[Schoeffel99] *Frank Schoeffel and Andreas Pomi.* Reducing Memory Requirements for Interactive Radiosity using Movement Prediction. In Dani Lischinski and Greg Ward Larson, editors, *Rendering Techniques '99*, pages 225–234. Springer, June 1999.

[Seitz97] *Steven Seitz and Charles Dyer.* Photorealistic Scene Reconstruction by Voxel Coloring. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1067–1073, June 1997.

[Servetto02] *Sergio Servetto, Rohit Puri, Jean-Paul Wagner, Pierre Scholtes, and Martin Vetterli.* Video Multicast in (Large) Local Area Networks. In *Proceedings of IEEE INFOCOM*, June 2002.

[Shade98] *Jonathan Shade, Steven Gortler, Li Wei He, and Richard Szeliski.* Layered depth images. In *SIGGRAPH '98:*

*Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 231–242. ACM Press, 1998.

[Shirley97] *Peter Shirley and Kenneth Chiu.* A low distortion map between disk and square. *Journal of Graphics Tools*, Vol. 2, No. 3, pages 45–52, 1997.

[Sillion95] *Francois Sillion.* A unified hierarchical algorithm for global illumination with scattering volumes and object clusters. *IEEE Transactions on Visualization and Computer Graphics*, Vol. 3, No. 1, pages 240–254, 1995.

[Slabaugh01] *G. Slabaugh, W. Bruce Culbertson, Thomas Malzbender, and Ron Shafer.* A Survey of Methods for Volumetric Scene Reconstruction from Photographs. pages *International Workshop on Volume Graphics 2001, Stony Brook, New York*, June 2001.

[Slabaugh02a] *Greg Slabaugh, Ron Schafer, and Mat Hans.* Image-Based Photo Hulls. *1st International Symposium on 3D Processing, Visualization, and Transmission*, pages 704–708, 2002.

[Slabaugh02b] *Greg Slabaugh, Ron Schafer, and Mat Hans.* Image-Based Photo Hulls. Technical Report HPL-2002-28, Hewlett-Packard Labs, 2002.

[Slater95] *Mel Slater, Martin Usoh, and Yiorgos Chrysanthou.* The influence of dynamic shadows on presence in immersive virtual environments. In *Selected papers of the Eurographics workshops in Virtual environments '95*, pages 8–21. Springer, 1995.

[Sloan02] *Peter-Pike Sloan, Jan Kautz, and John Snyder.* Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments. *ACM Transactions on Graphics*, Vol. 21, No. 3, pages 527–536, 2002.

[Smith96] *Alvy Ray Smith and James F. Blinn.* Blue screen matting. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 259–268. ACM Press, 1996.

[Smits94] *Brian Smits, Jim Arvo, and Donald Greenberg.* A clustering algorithm for radiosity in complex environments. *Computer Graphics*, Vol. 28, No. Annual Conference Series, pages 435–442, 1994.

[Sony] *http://www.sony.net.* Sony DFW-V500 Camera Specification, Sony Corp., Japan.

[Spheron] *http://www.spheron.com.* SpheronCam HDR, Spheron GmbH, Germany.

[Starlog81] *Starlog. Magicam Explores the Cosmos.* Starlog Press, 1981.

[State94] *Andrei State, Gentaro Hirota, David T. Chen, William F. Garrett, and Marc A. Livingston.* Superior augmented reality registration by integrating landmark tracking and magnetic tracking. In *Computer Graphics 30, Annual Conference Series (1994)*, pages 429–438, 1994.

[Stevens98] *W. Richard Stevens. UNIX Network Programming. Networking APIs: Sockets and XTI.* Prentice-Hall, Second edition, 1998.

[Swaminathan00] *Rahul Swaminathan and Shree K. Nayar.* Nonmetric Calibration of Wide-Angle Lenses and Polycameras. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 22, No. 10, pages 1172–1178, 2000.

[Szecsi04] *Laszlo Szecsi, Mateu Sbert, and Laszlo Szirmay-Kalos.* Combined Correlated and Importance Sampling in Direct Light Source Computation and Environment Mapping. *Computer Graphics Forum*, Vol. 22, No. 3, pages 585–593, 2004. (Proceedings of Eurographics).

[Theobalt03] *Christian Theobalt, Joel Carranza, Marcus Magnor, and Hans-Peter Seidel.* A Parallel Framework for Silhouette-Based Human Motion Capture. In *Proceedings of Virtual Reality, Modelling and Visualization (VMV)*, pages 207–214. AKA, Berlin, November 2003.

[Theobalt04] *Christian Theobalt, Joel Carranza, Marcus Magnor, and Hans-Peter Seidel.* 3D Video - Being Part of the Movie.

*ACM Computer Graphics*, Vol. 38, No. 3, pages 18–20, August 2004.

[Tsai86] *Roger Y. Tsai.* An efficient and accurate camera calibration technique for 3D machine vision. In *Proceedings of Computer Vision and Pattern Recognition*, 1986.

[Ultimatte] *http://www.ultimatte.com.* Chroma keying technology, Ultimatte Corp., US.

[Unger03] *Jonas Unger, Andreas Wenger, Tim Hawkins, Andrew Gardner, and Paul Debevec.* Capturing and Rendering with Incident Light Fields. In *Eurographics Symposium on Rendering: 14th Eurographics Workshop on Rendering*, pages 141–149, June 2003.

[Video4Linux] *http://www.video4linux.net.* The Video For Linux Project.

[Viera93] *Dave Viera. Lighting for Film & Electronic Cinematograpy.* Wadsworth Inc., First edition, 1993.

[Vlahos78] *Petro Vlahos.* Comprehensive Electronic Compositing System. US Patent 4100569, 1978.

[Waese01] *Jamie Waese and Paul Debevec.* A Real Time High Dynamic Range Light Probe. SIGGRAPH 01, Technical Sketch, 2001.

[Wagner02] *Markus Wagner.* Development of a Ray-Tracing-Based VRML Browser and Editor. Diploma Thesis, Saarland University. 2002.

[Wald01a] *Ingo Wald and Philipp Slusallek.* State-of-the-Art in Interactive Ray-Tracing. In *State of the Art Reports, Eurographics 2001*, pages 21–42, 2001.

[Wald01b] *Ingo Wald, Philipp Slusallek, and Carsten Benthin.* Interactive Distributed Ray Tracing of Highly Complex Models. In Steven J. Gortler and Karol Myszkowski, editors, *Rendering Techniques 2001*, pages 274–285, 2001.

[Wald01c] *Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner.* Interactive Rendering with Coherent

Ray Tracing. *Computer Graphics Forum*, Vol. 20, No. 3, pages 153–164, 2001. (Proceedings of Eurographics).

[Wald02a] *Ingo Wald, Carsten Benthin, and Philipp Slusallek.* OpenRT - A Flexible and Scalable Rendering Engine for Interactive 3D Graphics. Technical report, Saarland University, 2002.

[Wald02b] *Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek.* Interactive Global Illumination using Fast Ray Tracing. *Rendering Techniques*, pages 15–24, 2002. (Proceedings of the 13th Eurographics Workshop on Rendering).

[Wald03a] *Ingo Wald, Carsten Benthin, and Philipp Slusallek.* Interactive Global Illumination in Complex and Highly Occluded Environments. In *Proceedings of the 2003 Eurographics Symposium on Rendering*, pages 74–81, Leuven, Belgium, June 2003.

[Wald03b] *Ingo Wald and Tim Dahmen. OpenRT User Manual.* Computer Graphics Group, Saarland University, 2003. http://www.openrt.de.

[Wald04a] *Ingo Wald.* Realtime Ray Tracing and Interactive Global Illumination. PhD thesis, Computer Graphics Group, Saarland University, April 2004.

[Wald04b] *Ingo Wald, Andreas Dietrich, and Philipp Slusallek.* An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. In *Rendering Techniques 2004, Proceedings of the Eurographics Symposium on Rendering*, pages 81–92, June 2004.

[Wald04c] *Ingo Wald, Johannes Günther, and Philipp Slusallek.* Balancing Considered Harmful – Faster Photon Mapping using the Voxel Volume Heuristic. *Computer Graphics Forum*, Vol. 22, No. 3, pages 595–603, 2004. (Proceedings of Eurographics).

[Ward94] *Gregory J. Ward.* The RADIANCE Lighting Simulation and Rendering System. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94*, Computer Graphics

Proceedings, Annual Conference Series, pages 459–472. ACM SIGGRAPH, ACM Press, July 1994.

[Ward96] *Greg Ward.* Real Pixels. In James Arvo, editor, *Graphics Gems II.* Academic Press, 1996.

[Ward00] *Peter Ward, Alan Bermingham, and Chris Wherry. Multiskilling for Television Production.* Focal Press, First edition, 2000.

[Watt92] *Alan Watt and Mark Watt. Advanced Animation and Rendering Techniques. Theory and Practice.* Addison-Wesley, 1992.

[Wenger03] *Andreas Wenger, Tim Hawkins, and Paul Debevec.* Optimizing Color Matching in a Lighting Reproduction System for Complex Subject and Illuminant Spectra. In *Eurographics Symposium on Rendering: 14th Eurographics Workshop on Rendering*, pages 249–259, June 2003.

[Wexler02] *Yonatan Wexler, Andrew. W. Fitzgibbon, and Andrew. Zisserman.* Image-based environment matting. In *Proceedings of the 13th Eurographics Workshop on Rendering*, pages 279–290. Eurographics Association, 2002.

[Whetten95] *Brian Whetten.* A Reliable Multicast Protocol. pages *Theory and Practice in Distributed Systems. Lecture Notes on Computer Science*, 1995.

[Whitted80] *Turner Whitted.* An Improved Illumination Model for Shaded Display. *CACM*, Vol. 23, No. 6, pages 343–349, June 1980.

[Wojdala98] *Andrzej Wojdala, Marek Gruszewski, K. Dudkiewicz, and M. Donotek.* Real-time depth-of-field algorithm for virtual studio. *MGV (Machine Graphics and Vision)*, Vol. 7, No. 1/2, pages 5–14, 1998.

[Wojdala00] *Andrzej Wojdala, Marek Gruszewski, and Ryszard Olech.* Real-time shadow casting in virtual studio. *MGV (Machine Graphics and Vision)*, Vol. 9, No. 1/2, pages 315–329, 2000.

[Woo97] *Mason Woo and Jackie Neider et al. Open GL. Programming Guide.* Addison-Wesley, Third edition, 1997.

[Woop05] *Sven Woop, Jörg Schmittler, and Philipp Slusallek.* RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. In *Proceedings of SIGGRAPH '05*, Computer Graphics Proceedings, Annual Conference Series. ACM Press, August 2005. To appear.

[Wuermlin04] *Stephan Wuermlin, Edouard Lamboray, and Markus Gross.* 3D video fragments: Dynamic point samples for real-time free-viewpoint video. *Computers & Graphics, Special Issue on Coding, Compression and Streaming Techniques for 3D and Multimedia Data*, Vol. 28, No. 1, pages 3–14, January 2004.

[Wyckoff61] *Charles W. Wyckoff.* An experimental extended response film. Technical report, Edgerton, Germeshausen & Grier, Inc., Boston, Massachusetts, March 1961.

[Yu99] *Yizhou Yu, Paul Debevec, Jitendra Malik, and Tim Hawkins.* Inverse Global Illumination: Recovering Reflectance Models of Real Scenes From Photographs. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 215–224, August 1999.

[Zhukov98] *Sergey Zhukov, Andrei Iones, and Grigorij Kronin.* An Ambient Light Illumination Model. In *Rendering Techniques '98*, pages 45–56, 1998.

[Zonker99] *Douglas E. Zonker, David M. Werner, Brian Curles, and David H. Salesin.* Environment Matting and Compositing. In *Proceedings of ACM SIGGRAPH 1999*, pages 205–214, 1999.