UC Davis
IDAV Publications

Title
Image-Based Rendering of Range Data with Depth Uncertainty

Permalink
https://escholarship.org/uc/item/7gm811hf

Author
Hofsetz, Christian

Publication Date
2003

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Christian Hofsetz
November 2003
COMPUTER SCIENCE

Image-Based Rendering of Range Data with Depth Uncertainty

Abstract

Image-Based Rendering is an exciting new field, which lies in between Computer Graphics and Computer Vision. We believe that the more we use the knowledge from Computer Vision in our graphics rendering algorithms, the better our final rendered images will be. This dissertation presents a framework to identify what information from computer vision is relevant for rendering and how to render it.

Instead of only using the depth information per pixel, we compute what we call a *depth uncertainty region* around it. We show how to compute this region from an existing 3-D recovering algorithm called range-space search. We also present a new idea that further improve the uncertainty estimation, making it tighter. It is based on the assumption that the estimated depth is close enough to the actual surface.

Two approaches for rendering are presented. The 4-D approach is based on the light field ray space parameterization. The 3-D approach is based on an existing technique of splatting 3-D Gaussian kernels. We show that both techniques achieve good results, but the 3-D approach is faster and it produces sharper images in most cases.

Image-Based Rendering of Range Data with Depth Uncertainty

By

CHRISTIAN HOFSETZ
B.S. (Universidade do Vale do Rio dos Sinos, São Leopoldo, RS, Brazil) 1994
M.S. (Universidade Federal do Rio Grande do Sul, Porto Alegre, RS, Brazil) 1997

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

_____

_____

_____

Committee in charge

2003

Image-Based Rendering of Range Data with Depth Uncertainty

Copyright 2003
by
Christian Hofsetz

To

David,

and Alice,

and Berenice,

and also in memory

of my beloved mother,

Eluza Figueira Hofsetz,

who died of cancer last year.

"When we are no longer able to change a situation - just think of an incurable disease such as inoperable cancer - we are challenged to change ourselves."

Viktor Frankl, Man's Search for Meaning

Acknowledgments

The writing of a dissertation is only the closing chapter of years of research. I am grateful for all the support I have received not only during this step, but also for the support I received during all those years.

First of all, I would like to thank my close family. Thanks to my wife Berenice, for all her support. Also thanks to my daughter Alice and my son David just for being there for me. And since David was born in my last year as a graduate student, I must thank those who helped me to take care of my family during that time: Cláudia, Patrícia, Maria Célia and my mother-in-law Terezinha.

I am indebt to my father and my late mother, who always emphasized the importance of studying and the willingness to fight for my dreams. The price of studying abroad was specially high for my mother. She saw very little Alice, her only grandchild, before passing away. Nevertheless, she always supported us. Finally, I cannot forget my three sisters who were always there for us.

Next, I thank my advisor, Prof. Nelson Max, for his guidance throughout these years. I can't express how proud I am of being one of his students. Nelson Max and Bernd Hamann were the reason I choose UC Davis over four other universities. I was really impressed by how well received my wife and I were, when we visited Davis.

I thank the suggestions of my dissertation committee: Nelson Max, Bernd Hamann and Oliver Staadt. They were also part of my qualifying committee. I also thank the other members of my qualifying committee: Ken Joy and Niels Gronbech Jensen.

Special thanks to George Chen, from ST Microelectronics, for his great ideas. And I must acknowledge the help of my project colleagues from ST Microelectronics, who helped shape our hyperline and hyperpencil research: Kim Ng, Li Hong, Peter McGuinness.

For directly helping me in all my efforts during my research, I thank Yang Liu, Rodrigo Caimi and David Elbery. For helping me writing my dissertation thanks to Oliver Kreylos, David Wiley, T.-J. Kelly.

Also, I cannot forget the help of Kim Reinking, from the CS department; Pat Gomez, Jaime Jones and Maria Terese Tuccori, from CIPIC; and Joana Ribeiro de Abreu,

Marta Elias Ribeiro, and Maria das Gracas Galdino, from CAPES. If my academic life here went smooth, it is due to their support. TGFS, indeed!

I would like to thank the University of North Carolina IBR Group at Chapel Hill for their reading room and car wreck models.

Finally, special thanks for the support I had from all my sponsors:

- My home university, Universidade do Vale do Rio dos Sinos - UNISINOS.
- CAPES - Fundação Coordenação de Aperfeiçoamento de Pessoal de Nível Superior
- ST Microelectronics
- Digital Media Innovation Program (DiMI) from the University of California.

Contents

List of Figures ix

List of Tables xi

Abstract 2

1 Introduction 3
- 1.1 Motivation . . . 3
- 1.2 Previous and Related Work . . . 5
  - 1.2.1 Image-Based Rendering . . . 5
  - 1.2.2 Splatting . . . 6
- 1.3 Terminology . . . 7
- 1.4 Problem Statement . . . 9
  - 1.4.1 Approach . . . 9
  - 1.4.2 Contributions . . . 9
- 1.5 Document Organization . . . 10

2 Depth Uncertainty from Range-Space Search 12
- 2.1 Chapter Organization . . . 13
- 2.2 Recovering 3-D Depth . . . 14
  - 2.2.1 Epipolar Geometry . . . 14
  - 2.2.2 Matching Metrics . . . 17
  - 2.2.3 Multiple-Baseline Stereo Matching . . . 18
  - 2.2.4 Edge Detection . . . 19
  - 2.2.5 Dealing with Occlusion . . . 21
- 2.3 Range-Space Depth Extraction . . . 23
  - 2.3.1 Search and Match . . . 23
  - 2.3.2 Volume Growing . . . 25
  - 2.3.3 Smoothing . . . 27
- 2.4 Delta and Unconstrained Uncertainty . . . 27
- 2.5 Summary . . . 29

3 The 4-D Approach: Light Field Hyperlines 31
- 3.1 Chapter Organization . . . 31

3.2 Overview of Approach . . . 32
3.3 The Dual Representation . . . 33
3.3.1 The Light Field Parameterization . . . 33
3.3.2 Hyperlines . . . 34
3.3.3 Ray-Hyperline Distance Calculation . . . 37
3.4 Camera Hyperline Rendering . . . 39
3.4.1 Dynamic Focal Plane . . . 39
3.5 Geometry Hyperline Rendering . . . 44
3.5.1 Ray Selection . . . 44
3.5.2 Depth Clustering . . . 45
3.5.3 Hole Filling . . . 45
3.5.4 Optimizations . . . 46
3.5.5 Local Ray Space . . . 47
3.5.6 Pseudo Code . . . 49
3.6 Modeling Uncertainty in 4-D Rayspace . . . 50
3.6.1 The Weight Function . . . 51
3.6.2 Occlusion . . . 52
3.7 Summary . . . 53

4 The 3-D Approach: Ellipsoidal Gaussian Kernels **54**
4.1 Overview of Approach and Chapter Organization . . . 55
4.2 The Sampling Problem . . . 55
4.2.1 The Frequency Domain . . . 56
4.2.2 The Meaning of Sampling . . . 58
4.3 Ellipsoidal Gaussian Kernels . . . 61
4.3.1 Gaussian as Low-Pass Filters . . . 61
4.3.2 Orthogonal Projection of a 3-D Gaussian . . . 63
4.3.3 Implicit Ellipsoid . . . 64
4.3.4 Perspective Projection of an Ellipsoid . . . 65
4.3.5 The Perspective Approximation of the 3-D Gaussian . . . 69
4.4 Rendering Depth with Uncertainty using 3-D Ellipsoids . . . 70
4.4.1 The Ellipsoid . . . 72
4.4.2 Sorting . . . 74
4.4.3 Projection . . . 75
4.4.4 Corners . . . 76
4.4.5 The Final Color . . . 78
4.4.6 Normalization . . . 78
4.5 Summary . . . 79

5 Results **82**
5.1 Chapter Organization . . . 82
5.2 Light Field Hyperline . . . 83
5.2.1 Rendering Camera Hyperlines . . . 83
5.2.2 Rendering Geometry Hyperlines . . . 87
5.2.3 Two Problems . . . 89
5.3 Rendering Depth With Uncertainty . . . 93
5.3.1 Delta vs. Unconstrained Uncertainty . . . 95

5.4 The Choice of the Best Approach 101
5.5 Summary 102

6 Conclusions **105**
6.1 Summary 105
6.2 Future Work 107
6.2.1 Uncertainty Region Estimation 107
6.2.2 The 4-D Approach 107
6.2.3 The 3-D Approach 107

Bibliography **108**

List of Figures

1.1 A House Model from three input images 4
1.2 Document organization diagram 10

2.1 Obtaining z' from the traditional depth measure z 13
2.2 Projection of one ray from one image into another image 14
2.3 Epipolar Plane 15
2.4 Epipolar Pencil 15
2.5 Epipolar Lines 16
2.6 Template Window 18
2.7 1-D signal and its first two derivatives 19
2.8 The Sobel Edge Detector 20
2.9 Search intervals 24
2.10 Error region derivation 26
2.11 Volume Growing 27
2.12 Deriving the depth uncertainty from a ground truth depth map 28
2.13 Looking up the depth uncertainty 29

3.1 The Light Slab representation 34
3.2 Representation of camera and geometry as seen in 3-D 35
3.3 The two-plane- parameterization and derivation of dual relationship. 36
3.4 2-D ray space of a 2-D scene as seen by a 1-D camera 37
3.5 Focal plane 40
3.6 Dynamic focal plane reparameterization 42
3.7 Ray selection problems 44
3.8 4-D distance of geometry hyperlines from virtual ray 45
3.9 4-D distance threshold 46
3.10 Perspective foreshortening and area of contribution 47
3.11 Global and Local ray spaces 48
3.12 Pseudocode for geometry hyperline rendering 49
3.13 Example of a Hyperline Pencil 50

4.1 The Sampling Problem 56
4.2 Spatial and Frequency domain of function f 58
4.3 Multiplying f by a Comb Function C 58
4.4 Sampling causes aliasing in the frequency domain 59
4.5 Example of Aliasing 60

4.6 The Ideal Low-Pass Filter . . . 60
4.7 Example of a Gaussian Kernel . . . 62
4.8 Sampled Data . . . 62
4.9 Filtering . . . 63
4.10 Applying the Gaussian Kernel to an Image . . . 64
4.11 The orthogonal projection of an n-D Gaussian is an $(n-1)$-D Gaussian . . . 65
4.12 The Perspective Projection of an Ellipsoid . . . 68
4.13 Ellipsoid perspective projection may be a hyperbola or a parabola. . . . 69
4.14 Perspective approximation idea . . . 70
4.15 The 3-D Algorithm . . . 71
4.16 Depth Uncertainty as 3-D Ellipsoids . . . 72
4.17 Virtual Camera Definition . . . 74
4.18 The Vertex Program . . . 80
4.19 Fragment program to calculate final color of the fragment . . . 81
4.20 Fragment Program to Normalize Alpha . . . 81

5.1 Camera Hyperline Rendering with different number of input images . . . 83
5.2 Camera Hyperline Rendering: Ray Selection results . . . 84
5.3 Stanford Dragon: comparison of camera hyperline rendering and light field rendering . . . 85
5.4 Camera Hyperline Rendering: Stanford Dragon . . . 86
5.5 Dynamic focal plane . . . 86
5.6 *'Mate'* Input Images, 450000 points . . . 87
5.7 Comparison between global and local ray space rendering . . . 88
5.8 Local ray space sorting lists . . . 88
5.9 Comparing different choices of the depth window threshold V_d . . . 89
5.10 The UNC Reading Room . . . 90
5.11 Comparing different choices of the 4-D distance threshold D . . . 91
5.12 The 'Car Wreck Scene' . . . 91
5.13 The '*Pisa* Tower' model rendered with different resolutions and viewpoints, using an image-based threshold. . . . 92
5.14 The '*Pisa* Tower' model rendered with different resolutions and viewpoints, using a 4-D distance threshold D . . . 93
5.15 Opacity in Geometry Hyperline Rendering . . . 94
5.16 Uncertainty Map . . . 95
5.17 Uncertainty Map . . . 95
5.18 Hyperpencil Rendering . . . 96
5.19 Depth clustering solving occlusion . . . 97
5.20 Comparison of (a) Hyperpencil rendering and (b) volume splatting. . . . 97
5.21 Comparison of the Delta Uncertainty and the Unconstrained Uncertainty . . . 98
5.22 Contour graphs of the depth size $z_e - z_s$ for the unconstrained uncertainty and the delta uncertainty . . . 98
5.23 Depth size plot (for Y=150,X=0 to 399) . . . 99
5.24 Uncertainty disparity region size for the Delta Uncertainty, $\epsilon = 0.5$ . . . 99
5.25 Reconstructed 3-D points . . . 101
5.26 Comparison of rendering using Delta Uncertainty and Unconstrained Uncertainty. . . . 102

List of Tables

3.1 The Dual Relationship 37

5.1 Summary of rendering methods 104
5.2 Rendering methods: references and figures 104
5.3 Summary of concepts 104

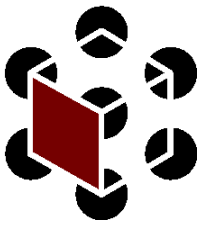


Image-Based Rendering of Range Data with Depth Uncertainty

Author: Christian Hofsetz
Committee: Dr. Nelson Max, Chair
Dr. Bernd Hamann,
Dr. Oliver G. Staadt

Visualization and Graphics Research Group
Center for Image Processing and Integrated Computing
Department of Computer Science
University of California, Davis

Abstract

Image-Based Rendering is an exciting new field, which lies in between Computer Graphics and Computer Vision. We believe that the more we use the knowledge from Computer Vision in our graphics rendering algorithms, the better our final rendered images will be. This dissertation presents a framework to identify what information from computer vision is relevant for rendering and how to render it.

Instead of only using the depth information per pixel, we compute what we call a *depth uncertainty region* around it. We show how to compute this region from an existing 3-D recovering algorithm called range-space search. We also present a new idea that further improve the uncertainty estimation, making it tighter. It is based on the assumption that the estimated depth is close enough to the actual surface.

Two approaches for rendering are presented. The 4-D approach is based on the light field ray space parameterization. The 3-D approach is based on an existing technique of splatting 3-D Gaussian kernels. We show that both techniques achieve good results, but the 3-D approach is faster and it produces sharper images in most cases.

Chapter 1

Introduction

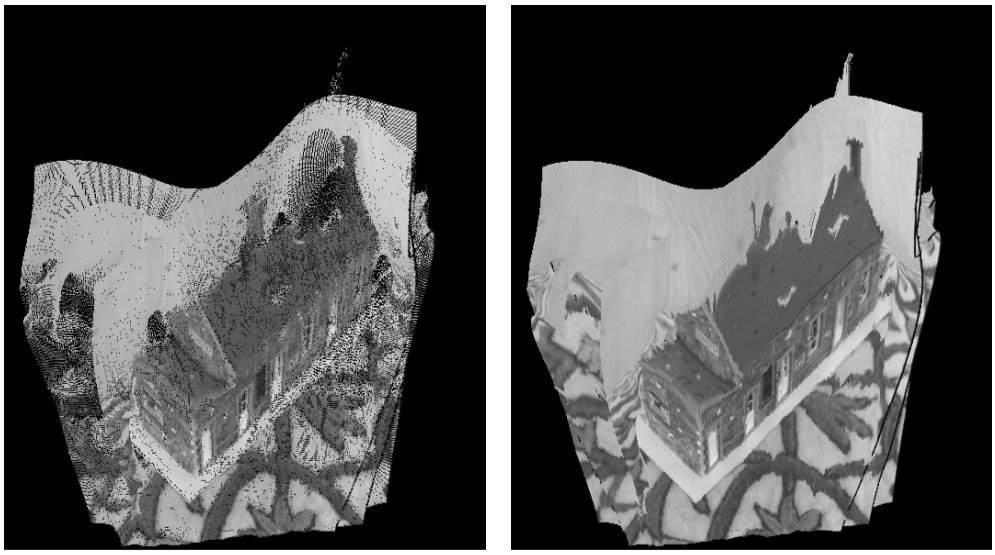
> *"A resposta certa, não importa nada: o essencial é que as perguntas estejam certas." "The right answer, it doesn't matter: the essential is whether the questions are right"*
>
> Mário Quintana

There is no simulator better than the real thing. No synthetic image is more photorealistic than a photograph itself. This is what makes the Image-Based Rendering (IBR) field so exciting. IBR is also unique because it is in the middle of two major research areas: Computer Vision and Computer Graphics. Therefore, it does not come as a surprise that the more we use the knowledge from Computer Vision in our graphics rendering algorithms, the better our final rendered images will be. This document is about going beyond the estimated depth usually available from 3-D depth recovering techniques.

1.1 Motivation

The choice between image-warping and light field techniques is a consequence of how many input images and how much geometry information we have. If we have few input images with geometry, we chose image-warping. If we have a large number of input images we chose light fields. However, in practice, what we usually have is something in between [21]. We may have several cameras available, but the sampling may not be dense enough for light field rendering. Some geometry information may be available, usually

extracted from the input images (using methods like stereo matching), but this information is often inaccurate. So how can we deal with this problem?



(a) (b)

Figure 1.1: A house model, acquired by ST Microelectronics, using three input images. (a) Shows input points acquired through stereomatching techniques, (b) shows input points as a mesh of triangles

First of all, it is important to notice that both light and geometry are just different ways of describing the same information. In fact, we can represent both directly in a 4-D ray space, and this is shown in chapter 3. This allows us to develop a unique rendering framework that handles light and geometry in a similar way.

Also, we have to take into account that acquiring and registering data is far from being perfect and, depending on what kind of geometry reconstruction technique we use, we may have more or less accuracy. For instance, even if we acquired depths with a laser scanner and registered them with their equivalent color images, we would still have problems, especially with occluded objects or with very dark objects that do not reflect the laser beam. Our framework must be flexible enough to deal with this kind of data and also with data that is extracted from several input images using far less accurate techniques such as stereo matching (Figure 1.1), which introduces a high level of uncertainty.

Here we are interested in defining this "uncertainty" and using it in our rendering

framework.

1.2 Previous and Related Work

In this document we present a 4-D approach based on the light field ray space, and a 3-D approach based on volume splatting. In this section we review first, light field rendering in the context of Image-Based Rendering techniques and next, the splatting technique.

1.2.1 Image-Based Rendering

Image-based rendering techniques fall into two main categories: geometry based and light field based. Light field techniques [1][2] do not necessarily need geometry information. We can think of the scene as a space full of rays, a portion of which are recorded by cameras. By resampling the recorded rays according to the geometry of the virtual camera we can reconstruct the image. Geometry information helps to improve the quality of the image [2], although it requires considerable preprocessing. Chai, Tong, Chan and Shum [3] have shown that it is possible to identify, using spectral analysis, the optimal number of image samples and depth planes for anti-aliased light field rendering.

Light field rendering can be categorized as structured or unstructured [9]. Structured light field requires resampling the input images in some uniform way. In this case, camera and ray selection are straightforward, given how the desired image projects into these grids. This allows very fast reconstruction and it can be done in hardware. But this resampling may cause loss of information, especially if the input images are acquired non-uniformly, with different resolutions and camera positions. Unstructured light fields, on the other hand, use the input images directly, which makes camera and ray selection more complex. It is necessary to identify which cameras are closer to each virtual pixel and how they will contribute to the final color. Buehler et al. [5] use several weights to take into account resolution, geometry information, minimal angular deviation and other desired properties.

Geometry based approaches transfer input images to the virtual camera through the use of scene geometry, which can be in the form of per-pixel depth [4] or polygonal

models [6] [7] [8].

The most common method of rendering point clouds is to increase the size of each projected point to multiple pixels [10] [11] [15]. Another way is to ensure that there are enough point samples to match the resolution of all desired virtual views, i.e., if the scene is adequately sampled as shown by Grossman and Dally [16]. Holes can be filled by blending pixels from lower resolutions approximations of the image.

Polygon-based image warping identifies correspondences between the reference images. The source images can be piecewise projectively warped to the destination image and combined according to some weighting function. To handle occlusion, image patches must be warped following a certain order given by the projection of their camera centers in the virtual image [9].

If per-pixel depth is used, a point sample rendering algorithm is necessary in order to handle occlusion and hole filling. Information about how the points are connected is often not available.

Many Image-Based Rendering methods use images with depth (or range images) as primitives for rendering. Images with depth may contain more information about the data than just the color and 3-D position. This information can be exploited using specific data structures and search orders. For example, it is very common to organize those images as a 3-D structure called Layered Depth Images (LDIs) [17] [18]. LDIs can be organized into hierarchical representations [19] to improve rendering performance.

1.2.2 Splatting

The splatting technique is about projecting object primitives to the screen. If the original object primitive is only a point or a voxel, its projected contribution is spread over an area. This area is often called a *kernel* or *footprint*, which can be defined in object-space (for instance, by centering a fixed-size circle on the 3-D point) or in image-space. Each point of the kernel (or its projection) on the screen has color and blending information, usually in the form of opacity. If more than one kernel is projected to the same pixel, a composition scheme is needed, which often depends on the ordering of the splats. Occlusion is dealt by either searching through the objects in a specific order defined by the viewpoint

location, or by identifying which objects are actually visible before blending.

For instance, in the first implementation of the splatting technique for volume rendering [13], a front-to-back or back-to-front order was necessary in order to solve occlusion. Voxels that are closer to the image plane take precedence over farther voxels. If a back-to-front order is used, closer voxels replace voxels farther away. If a front-to-back order is used, the first voxels to project to a pixel overshadow the voxels behind them. A more recent paper, the Surfel technique [12], uses a two-pass solution for this problem. First, the visible pixels are determined by projecting tangent disks of each point (or surfel) to the screen. This is implemented in hardware. The z-buffer will then select the front most disks. The size of the disk depends on the level of detail and the distance of nearby pixels. The projection stores the depth and a pointer to the closest surfel. During the second pass each image pixel will have either a pointer to the closest surfel - in which case the color is interpolated between nearby visible surfels - or no surfel at all.

In order to speed up rendering, it is possible to construct a hierarchical representation of the data. Laur and Hanrahan proposed in [14] a hierarchical splatting algorithm, where the volume data is represented as a pyramid. Each level of the pyramid stores a different size of the footprints. During rendering, more than one level of the pyramid may be used in order to emphasize points closer to the viewpoint.

We are particularly interested in 3-D splatting, since our basic primitive is a 3-D depth region which defines our uncertainty (see chapter 2). Our 3-D rendering technique presented in chapter 4 is based on the work of Zwicker et. al [38] where each point is represented as an ellipsoid. An elliptical kernel is used to approximate the projection of the ellipsoid.

1.3 Terminology

This section contains the definition of some terms that are used in this dissertation. Some of the terms are explained in much more detail in the section they are introduced.

- A **Light field** is originally defined as the radiance at a point in a given direction (defined by a "*ray*"), i.e., a 5-D function. In this document, the term "light field" is used

as its 4-D approximation [1], where the rays are parameterized by their intersection with two parallel planes, sometimes called *camera plane* and *focal* or *object plane*.

- **Ray space** represents the space of the light field function. A 4-D ray space represents the two-plane parameterization of a 4-D light field.
- **Global Ray Space** defines a configuration where the scene is parameterized with only two fixed parallel planes.
- An **Input Image** is a known image used as input to our rendering frameworks. It can be either a photograph or a synthetic image.
- The **Focal plane** is one of the planes of the light field parameterization. It is usually located near where the objects of interest of the scene are. In this way, those objects will be *focused*. The focal plane may be replaced with an approximate geometric model [5].
- **Local Ray Space** defines a configuration where each input image has its own two-plane parameterization scheme, defined by the center of projection and the image plane.
- In Computer Vision *depth* usually means the z distance from the image plane. However, from now on in this document, **Depth** means the distance along the viewing ray. Figure 2.1 shows how to obtain the distance along the viewing ray z' from the traditional depth measure z.
- **Depth Uncertainty:** In this document, uncertainty is specifically *depth* uncertainty, i.e., an asymmetric region around the best estimated depth of a pixel.
- A **Hyperline** is a 2-D entity living in a 4-D ray space. It is not called a plane because it has only 2 degrees of freedom.
- The **Virtual image** is the new view to be reconstructed/synthesized by the rendering algorithm.
- The **Virtual camera** is the camera position for the virtual image.

- A **Virtual ray** is defined by one pixel in the virtual image. This pixel can be represented as a ray from the center of projection to its location in the virtual image.

1.4 Problem Statement

When synthesizing images from real images, IBR techniques often do not take advantage of most of the knowledge gathered from the Computer Vision techniques during the depth recovering step. When a pixel depth is recovered, it is possible to identify how accurate, or how uncertain, it is.

1.4.1 Approach

Our approach can be divided into two sequential steps, reconstruction and rendering. First, the uncertainty information must be acquired during the scene reconstruction step. This information must be available for rendering in the next step. Second, the actual rendering is performed, taking into account this uncertainty.

For the reconstruction step, we have used the depth uncertainty information produced in [22]. Alternatively, we also propose a new method to recover uncertainty based on an existing 3-D depth estimation of the input images.

We have implemented two techniques for rendering uncertainty. The first one considers the depth uncertainty as an entity in 4-D ray space. The second one represents this information as a 3-D object, an ellipsoid. A comparison of these two techniques is presented in the results chapter (chapter 5).

1.4.2 Contributions

The contributions of our research can be summarized as:

1. A new algorithm for light field rendering, called "*camera hyperline rendering*". We assume no knowledge of the geometry. It is based on a 4-D distance from a ray in the input image to a ray in the new image

2. A new Dynamic Focal Plane formula for camera hyperline rendering

3. A new algorithm for point cloud rendering in the context of light field ray space, called "*Geometry Hyperline rendering*".

4. A new algorithm for rendering uncertainty in the context of light field ray space, called "*Hyperpencil rendering*".

5. An adaptation of an existing Volume Splatting algorithm for rendering uncertainty information

1.5 Document Organization

Figure 1.2 illustrates the document organization. After this section, chapter 2 introduces the concept of *depth uncertainty*. This chapter also presents a brief review of computer vision techniques.

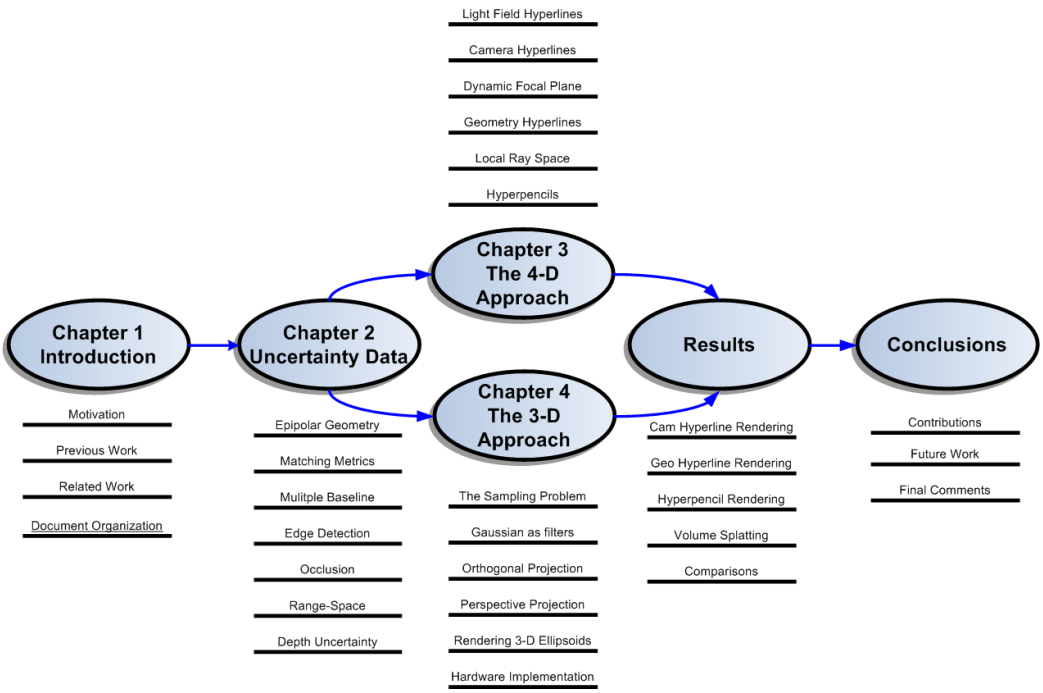


Figure 1.2: Document organization diagram

Chapter 3 presents our 4-D implementation based on light field hyperlines, and chapter 4 our 3-D implementation, based on ellipsoidal Gaussian kernels. Although it may look unusual, we have decided to describe our 3-D implementation after our 4-D version

because it reflects the sequence of progress of our research. The 3-D algorithm was implemented last. Chapter 4 also contains an introduction to the sampling problem and the frequency domain.

Chapter 5 presents the results of our research, including a comparison of the 3-D and 4-D methods for rendering uncertainty. We also show many results from our point-cloud rendering technique that came before the consideration of uncertainty in our framework.

Chapter 6 closes the dissertation. After some final comments, we discuss where we are heading with our research.

Chapter 2

Depth Uncertainty from Range-Space Search

> *"I think that modern physics has definitely decided in favor of Plato. In fact the smallest units of matter are not physical objects in the ordinary sense; they are forms, ideas which can be expressed unambiguously only in mathematical language."*
>
> Werner Karl Heisenberg

Uncertainty is a very vague term, and it may refer to various different aspects of the input images. For example, it may be related to the intrinsic or extrinsic camera parameters. Or it may identify how confident we are with some pixel correspondence between two or more input images - the more homogeneous the area is, the more uncertain is our depth information.

Acquiring and registering data is far from perfect and, depending on what method of geometry reconstruction technique we use, we may have more or less accuracy. The fact is, computer vision algorithms usually know the accuracy of the information they are producing, but most of the time it is not used by image-based rendering algorithms.

We have decided to restrict our study of uncertainty to depth uncertainty. Recall that, unless otherwise specified, depth means *the distance along the viewing ray*. Figure 2.1 shows how to obtain the distance along the viewing ray z' from the traditional depth measure z. For simplicity, in this document we assume that the intrinsic and extrinsic

camera parameters are known, and all cameras are planar pinhole cameras. The depth uncertainty used in our work is generated by the approach developed by Kim, Trivedi and Ishiguro [22]. It is a generalized multiple-baseline stereo synthesis using range-space search, match and render. Here, we use the result generated by the match step (called an error region) and render using hyperline pencils (chapter 3) or ellipsoidal gaussian kernels (chapter 4). The difference between our rendering and the one presented in [22] is that they assume a dense set of input cameras, while ours works with a sparse set, usually three.

It is important to notice that, in theory, any kind of depth range uncertainty can be used with our rendering methods. The only changes needed are the parameters necessary to control the rendering itself.

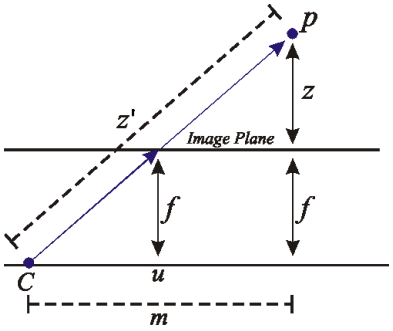


Figure 2.1: Obtaining z' from the traditional depth measure z. Given a point p in object space, $z' = \sqrt{(z+f)+m^2}$, where $m = \frac{u(z+f)}{f}$ and C is the center of projection.

2.1 Chapter Organization

Section 2.2 review some concepts and techniques of computer vision, such as stereo matching, epipolar geometry, match metrics, edge detection mechanisms and multiple-baseline stereo matching. Next, in section 2.3 the range-space approach is reviewed, which is divided into searching and matching, volume growing and smoothing.

Finally, in section 2.4 we present an alternative solution to obtain the uncertainty depth from a set of images.

A pair of images is called either (1) left and right images or (2) image i and image j, where image i is the reference image and image j is where the corresponding pixel is

being searched.

2.2 Recovering 3-D Depth

Recovering depth from pictures is a very active research field that produces many new publications each year. Therefore, we have to restrict our review here in order to approach only the subjects necessary to explain the range-space approach.

2.2.1 Epipolar Geometry

In order to extract the 3-D scene from 2-D pictures we have to look for correspondences between two or more images. We know that a point in one image can be represented as a 3-D ray. This ray intersects the object at some unknown depth. The problem here is to identify where this point appears in another image, from a different point of view. Figure 2.2 shows an example. One point in the image plane of C_2 corresponds to a ray. From the point of view of C_1, the actual 3-D point may be projected anywhere on a line, in its image plane, that is the projection of this ray. This line is called an *epipolar line*. One point in one camera corresponds to an epipolar line in another camera. The line that connects the center of projections of the two cameras is called a *baseline*. The intersections of the baseline with the image planes are called *epipoles*.

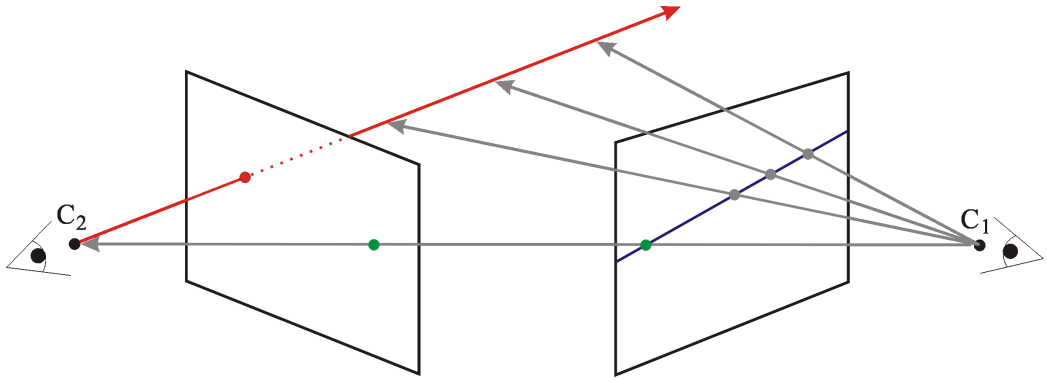


Figure 2.2: Projection of one ray from one image into another image. The epipoles are indicated by green dots. C_1 and C_2 are the center of projections of the images.

A point in 3-D defines an *epipolar plane* (Figure 2.3), and the intersection of an epipolar plane with the image planes defines an epipolar line in each. All the possible

epipolar lines may be obtained by rotating a plane along the baseline. The set of the planes passing through the baseline is called an *epipolar pencil* (Figure 2.4) .

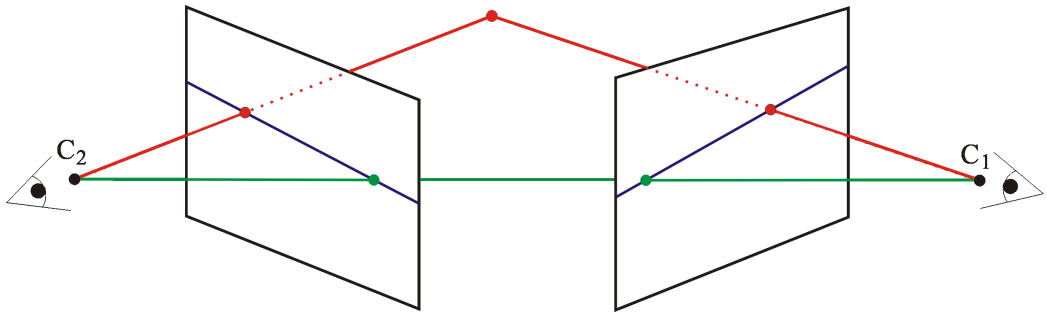


Figure 2.3: Epipolar Plane

All epipolar lines in one image intersect at the epipole. If the intersection of the baseline lies on the image plane, it will form a pencil of lines passing through the epipole. If the baseline is parallel to one of the image planes the epipolar lines in that plane will all be parallel.

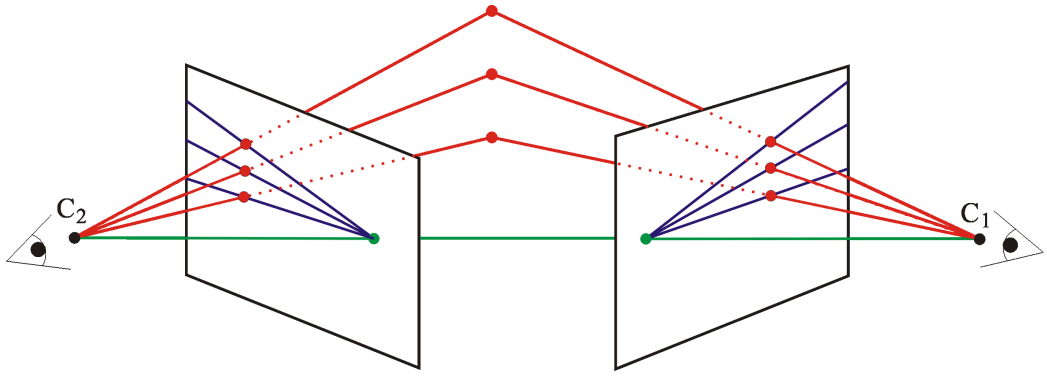
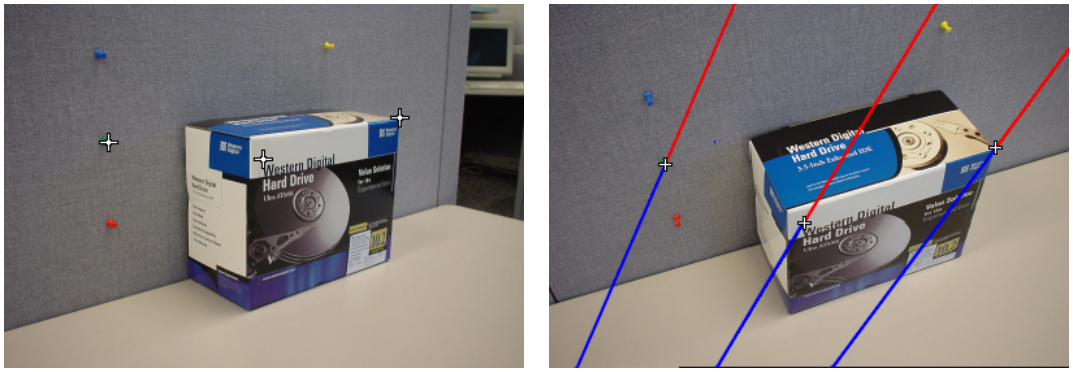


Figure 2.4: Epipolar Pencil

Figure 2.5 shows an example of how points in one image are represented as lines in another image. If the camera parameters are correct, we can see that the matching point in the other image lies necessarily in this line, if it is not occluded. This is also known as *the epipolar constraint* because it constrains the search of the matching point to this line only.

The knowledge of the epipolar geometry is very useful for recovering depth. First of all, the epipolar geometry is dependent on the camera intrinsic and extrinsic parameters, which is important to help calibrate cameras. It can also be used to calculate unknown



(a) Selected pixels (b) and their epipolar lines

Figure 2.5: (b) shows the epipolar lines of three selected pixels from (a)

camera parameters. Second, it is independent of the underlying 3-D scene. If the cameras are fixed, the scene may change, but the epipolar geometry will remain the same. Finally, due to the epipolar constraint, the searching and matching of one point from C_2 in C_1 is reduced to a 1-D search along the epipolar line.

In some special cases where the baseline is parallel to both image planes, the epipoles are at infinity and, thus, all epipolar lines are parallel. Search is even more simplified in this setup if the lines are parallel to the image axes, since the search can be done directly on the horizontal lines of the image. It is possible to take advantage of this setup even in the general case, for any pair of views. In this case we apply a transformation called *rectification*. It projects both images to a virtual image plane parallel to the baseline. This process is not always desirable since it requires resampling of the input images, which may cause loss of information.

Now, given a rectified pair of images, let P be a 3-D point in the scene that is projected to x_L,y_L in the left image and x_R, y_R in the right image. Let C_R and C_L be the center of projection of the right and left images, respectively. The *disparity* is defined as:

$$\delta = \frac{x_R - x_L}{|C_R - C_L|} \tag{2.1}$$

In some references the disparity is not normalized, i.e., $\delta = x_R - x_L$. In Image-Based Rendering the *generalized disparity* is often used instead and it is calculated as:

$$\delta_{generalized} = \frac{|\vec{r}|}{|P - C_L|} \tag{2.2}$$

where $\vec{r}$ is the vector from C_L to the 3-D position of the pixel (x_L,y_L).

2.2.2 Matching Metrics

In order to calculate the correspondence between two images we have to measure how different one image is from the other. We also must assume that most pixels from one image are seen from the other.

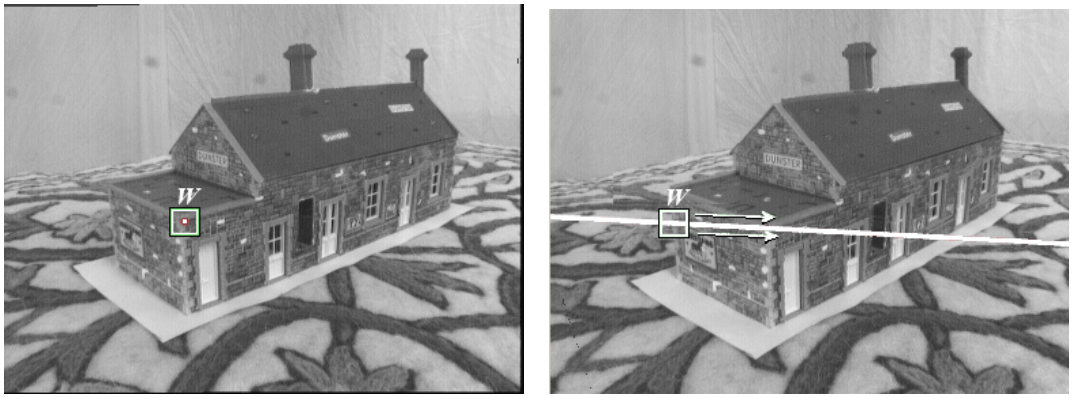
There are many approaches to compare a set of images (usually two). The most common are local methods and global methods [30]. Local methods calculate the correspondence between a small area around the pixels and select the pixel with the minimum associated cost. Once we find the correspondence, the depth is calculated by triangulation. Global methods calculate a global disparity function based on an energy-minimization framework, with an energy penalty for change in disparity between neighboring pixels.

Even though local methods are more prone to ambiguity in their solution, they are simpler and faster. Due to the increased complexity of finding an error region using a global method (instead of only the disparity), we concentrate here on local methods.

A naive approach would be to compare the color of one pixel in the reference image to the color of the current candidate pixel in the other image. However, some colors may be duplicated along the epipolar line, and there is usually noise in both pictures. Therefore, the comparison is actually done with a template window around both pixels (Figure 2.6). The procedure to find the best correspondence is:

1. Fix pixel p_i in reference image i, define its template region W_i
2. For every pixel [1] p_j in the epipolar line on image j, $i \neq j$
   place a template window W_j around p_j, calculate its correspondence with W_i
3. Select p_j with the best template match, calculate depth

[1] The points on the epipolar line are rarely at pixel centers, so the selection of the candidate pixel is more complicated. Usually, resampling is required to generate W_j.



Reference image i Image j

Figure 2.6: Search best match from image j to pixel in image i by looking into a template window W

To compare templates we have to calculate how similar (or rather how different) they are. In practice, the most used metric is the *Sum of Squared Difference*, or SSD. The SSD of W_i and W_j is given by:

$$d_{SSD} = \sum_{(m,n) \in W_i, W_j} [W_i(m,n) - W_j(m,n)]^2$$

where $W_i(m,n)$ and $W_j(m,n)$ are the colors of the pixels in the locations m and n of W_i and W_j, respectively.

Another way of calculating the difference between templates is the *Sum of Absolute Difference*. It is faster[2] since it is not necessary to calculate the square:

$$d_{SAD} = \sum_{(m,n) \in W_i, W_j} |W_i(m,n) - W_j(m,n)|$$

2.2.3 Multiple-Baseline Stereo Matching

In the general case where we have not two but several images to consider, it is possible to concurrently search along multiple epipolar lines in each image other than the reference. The idea here is that global mismatches can be minimized by summing all SSDs values from multiple stereo pairs. Thus, the difference between all template windows and the reference template window is the sum of all SSDs, called the SSSD [31]:

[2]Some architectures (like the Pentium IV) have one assembler instruction to calculate the absolute value.

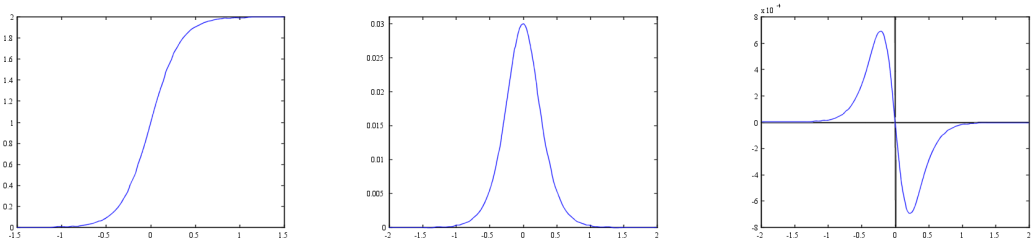
$$d_{SSSD} = \sum_{k=1}^{n} \sum_{(m,n) \in W_i, W_j} [W_i(m,n) - W_j^k(m,n)]^2$$

In the simple stereo scenario, searching can be done in image space. When multiple images are involved we have to search at intervals on the ray, where the projection of each interval is at most one pixel-width away from the previous pixel in all images j.

2.2.4 Edge Detection

Color matching is usually not enough to estimate depth. In order to solve the ambiguities from the epipolar line searching we often need to extract some features from the image. The most interesting features the human eye notices in a picture are the discontinuities in intensity. Those discontinuities are the edges, or the boundaries of the objects in the scene.

Edges can be identified as sudden changes in intensity from one pixel to a neighboring one. Let us analyze a simple continuous function in 1-D, Figure 2.7a. It represents a rapidly change in intensity, or a 1-D edge. It turns out we can identify it by either looking for the maximum or minimum of the first derivative (2.7b) or, alternatively, by finding the zero crossings of its second derivative (2.7c).



(a) $f(x)$ (b) $f'(x)$ (c) $f''(x)$

Figure 2.7: 1-D signal and its first two derivatives

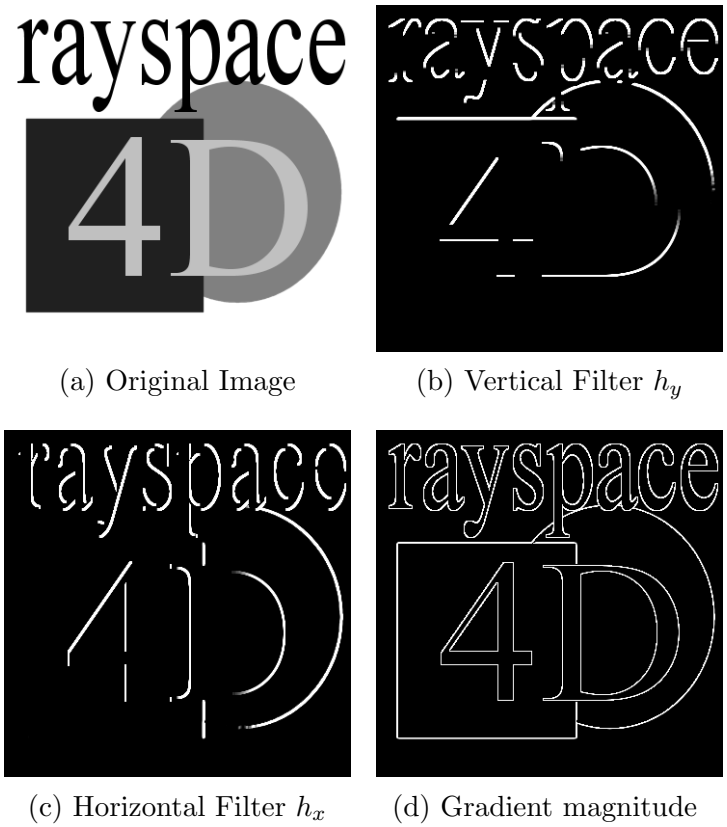
Gray Scale Images

Most gray scale methods of detecting edges can be grouped into two categories: gradient (first order derivative) and Laplacian (second order derivative) [27]. The most common is the Sobel edge detector, a gradient-based method.

The Sobel method applies two convolution kernels (or masks) to the pixels, one to find the horizontal edges (h_x) and the other to find the vertical edges (h_y) [28].

$$h_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad h_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

The result of applying these two kernels can be represented as a gradient vector $g = [g_x, g_y]$, which has a magnitude $\sqrt{g_x^2 + g_y^2}$ and a direction[3] $tan^{-1}(g_y/g_x)$. Figure 2.8 shows a sample picture and the results (actually absolute values) of applying the horizontal and vertical filters, as well as the final image of the gradient magnitude.



(a) Original Image (b) Vertical Filter h_y

(c) Horizontal Filter h_x (d) Gradient magnitude

Figure 2.8: The Sobel Edge Detector

[3] $atan2(g_y, g_x)$ is needed in practice, because of the 180^0 ambiguity in tan^{-1}, and the possibility of dividing by zero.

Color Images

Color-edge detection is more complicated because we have three color channels to evaluate (R, G and B). Ruzon and Tomasi [29] divide the color-edge detection techniques into three categories:

- Output fusion methods
- Multi-dimensional gradient methods
- Vector Methods

Output fusion methods are the most commonly used for color-edge detection. The idea is to apply the traditional gray scale edge detection techniques to each color channel, and from each result gradient create three edge maps. Finally we combine (or fuse) all three maps into a single one.

Multi-dimensional gradient methods create a single gradient vector derived from the gradient of all channels and then calculate the edge map based on this single gradient.

Finally, vector methods consider each pixel in the image as a vector of some kind, which is not the gradient. For instance, Ruzon et. al [29] use the CIE-lab color space. This color space is perceptually uniform, which means that the Euclidian distance between two colors has a perceptual meaning.

2.2.5 Dealing with Occlusion

If the viewpoints for two images are close enough together, the occlusion problem in stereo matching algorithms can be ignored, since almost all points from one image will be visible in the other image [32]. This is why occlusion is not a problem for scene extraction from sequence of images from a moving camera (the problems here are the camera parameters). Most of the time, though, solving occlusion is not trivial.

Brown, Bruschka and Hager in [34] divide the algorithms for handling occlusion in stereo computation into three classes: methods that detect occlusion, methods that reduce the contribution of occluders and methods that actually model the geometry of the occlusion.

One example of a method that detects occlusion was presented by Chang et. al in [33]. The idea is to look for inconsistencies between the depth maps. The left depth map is obtained by searching the epipolar lines in the right image, and the right depth map is obtained by searching the epipolar lines in the left image. Now, for each image, we warp every pixel to the other image. Next, the warped depths are compared with the depths in the map of the other image. If the depths are not similar within a given threshold, we assume that this is due to occlusion. This is not always the case, since it can be caused by many other factors, like specular reflections or noise. Another way to detect occlusions is to use the edge map as a guide to smooth the final depth map. In this case, depths that are outliers are discarded.

Kanade and Okutomi [35] introduced a method that changes the template window size iteratively, and, therefore, reduces the contribution of any possible occluder. They used a statistical model to calculate what they call the *uncertainty*[4] of a window. Initially the window is small (3x3). Its size changes one direction at a time. If, for one specific direction (left, right, up, down), the resulting uncertainty surpasses a given threshold, the window will not expand in that specific direction. The process stops when the window cannot be expanded anymore for any direction. For instance, let's say we start to increase the window by one pixel to the left. If the uncertainty value of the new window does not exceed the threshold, the expansion is kept (the new window is then 4x3). Next, we expand to the right and repeat the process. If the new uncertainty value of the new window (5x3) is below the threshold we keep this window. Otherwise, the window maintains its previous 4x3 size. We then check the other directions and we stop if the window did not change its size after checking all directions.

Finally, methods that explicitly model the geometry of the occlusion (like slope, object boundaries and/or other 3-D properties) are usually dynamic programming techniques [36]. Voxel carving techniques, such as Voxel Coloring [37] also belong to this category, since the volume carving process itself is guided by a *visibility constraint* that guarantees a single visibility ordering of the voxels. The constraint requires that no scene point lies in the convex hull that contains all input camera centers. In this way, it is possible

[4]This is not the same as the uncertainty discussed in this dissertation.

to visit the voxels in a way such that if a voxel m occludes a voxel n, m will necessarily be visited before n.

Section 2.3.1 shows how the range-space solution for occlusion uses both a color-edge map and an adaptive template window (which is circular, not rectangular).

2.3 Range-Space Depth Extraction

The range-space approach presented in [22] and [23] is a technique for searching and rendering an *error region* from a set of multiple input images. It is called *range-space search* because the searching is performed in world space, not in image space. In this document the rendering step of the original algorithm will not be presented.

The technique can be divided into three independent steps:

1. Search and Match
2. Volume Growing
3. Smoothing

2.3.1 Search and Match

The searching and matching step is basically a search for pixel correspondences between images. Traditionally it's done in image space by searching the best correspondence of one pixel in one image in the epipolar line of the other images. Nevertheless, here the actual search is done in world space. In this case the search step is not obvious since there is no actual grid. In the following sections we will describe the searching and matching of the pixels from a *reference image* i to all input images j, where $i \neq j$.

Searching

The search is done as follows: for each pixel from image i we create a 3-D frustum, centered at the camera's center of projection. The frustum discretization is guided by the pixel grid of all input images j. The size of a pixel in one image defines an interval in the ray-frustum. The next search position in the ray is given by the smallest interval from all

input images (Figure 2.9). Therefore, it is guaranteed that the next search position will project to a new location in each image j that is at most one-pixel far away from the current candidate pixel in this image, and exactly one pixel in at least one of the images. The size of each interval may be different.

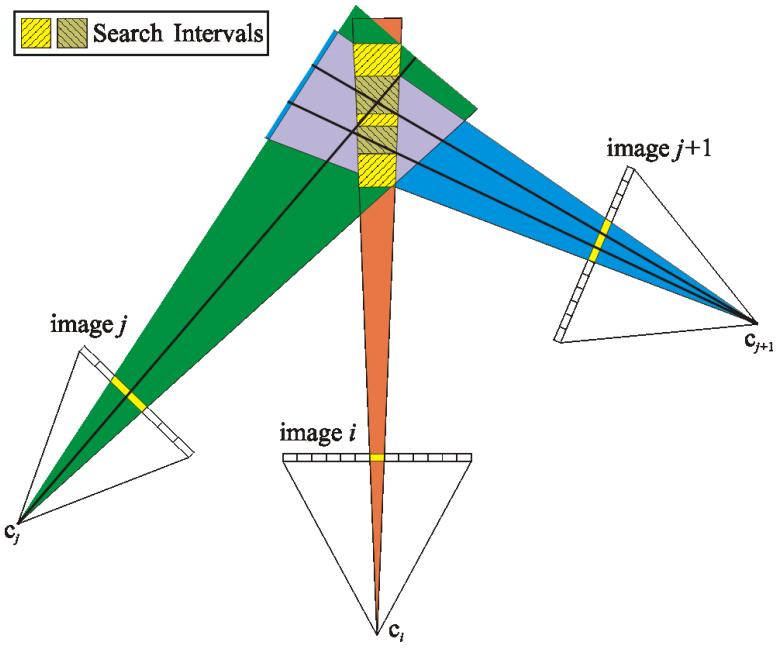


Figure 2.9: Search intervals in the frustum may have different sizes.

Matching

Unlike most image-space techniques, the template window used to calculate the correspondence here is not rectangular. The template is a projection of a sphere, centered in the current interval in the frustum, onto all images j. The resulting template in the image-space will be circular. The diameter of the sphere must be bigger than the size of the current interval in order to better average the noise in the images. Notice that because the template depends on the projection of this sphere, it will be different for each image. The closer the image plane of j is to this interval, the more pixels are considered in the distance calculation. In order to have the same number of pixels in all windows, resampling is often needed.

Now, in order to find correspondences we have to calculate two SSD values for each position searched in the frustum. One is based on the color images (color match)

and the other is based on the color-edge images (color-edge match, see 2.2.4) calculated in a pre-processing step. The result is a matching curve for each type of image (color and color-edge images).

Let us take a close look at the characteristics of the color matching curves. Every time a pixel from i is compared to a pixel from j we calculate its SSD value. If the colors are similar, the result will be close to zero. An abrupt change of color in the epipolar line will cause a peak. If this change makes the templates more similar in color, this may result in a global or local minimum. Each peak that causes a local minimum defines what we call here an error region.

Figure 2.10 shows an example of a matching curve containing only one error region. It shows the matching error for each possible depth value. In this curve, the lower the matching error, the more similar the corresponding pixels are. The vertical dotted lines represent depth samples, at equal disparity spacing. Recall that between two successive z samples the projection of the corresponding pixel from i differ by one pixel in at least one of the input views j, and by at most one pixel in any of the views. An error region starts at z_s which is the depth where the error falls below a given threshold η. The region ends at z_e, which is where the error raises above η plus the smallest matching error in that region located at depth z_b. Note that the definitions of z_s and z_e are not symmetrical.

2.3.2 Volume Growing

In practice, there are usually many error regions in a given color matching curve. The error region containing the minimum global error is often not the best match. In order to identify the best error region for each input pixel from i we need to propagate the best matches we have throughout i, based on the color-edge matching curve. The best matches will be our *seeds*. A pixel p can become a seed if one of its local minimum peaks coincides with a local minimum from the color-edge matching curve. After the seeds are chosen, the color-edge matching curves are not used anymore.

Alternatively, the seeds can also be obtained by any other 3-D depth recovery technique, since what we are interested here are in the error regions (section 2.4 shows how the searching for this error region can actually be done in reverse, starting from an

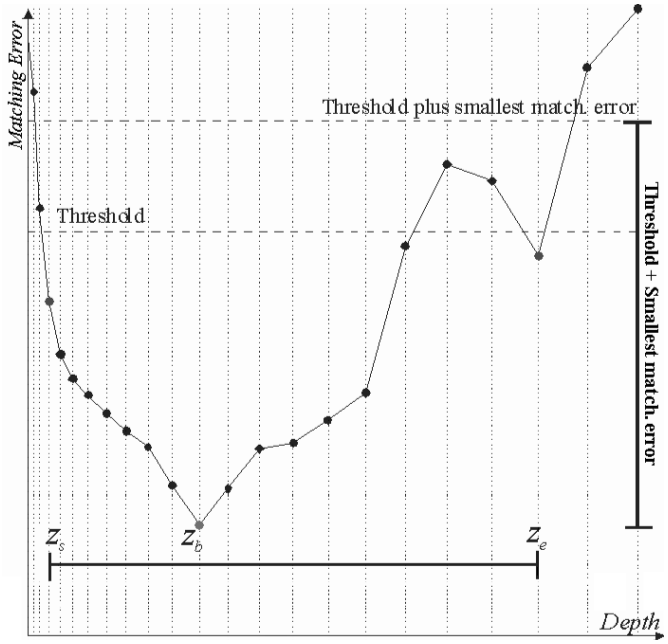


Figure 2.10: Example of error region derivation from a sample matching curve [22]

estimated depth).

The process of finding the best error regions for all pixels is iterative. The seeds will help to determine the best error region from adjacent pixels that are not themselves seeds. Those areas are usually homogeneous in color. Once the best error region from a pixel is identified, it also becomes a seed. The process ends when all pixels have their best error regions identified.

A seed can only propagate to one of its 8 neighbors. In order to assure continuity, it can only propagate if (1) the best z is contained in an error region of the neighbor pixel and (2) the best z is at most one search interval away from the local minimum in the chosen error region. In the case where a non-seed pixel is surrounded by one ore more seeds, the one closer to the viewpoint of the non-seed pixel is chosen. One extra advantage of this process is that it solves occlusion automatically; occluded regions from other images will not have any influence here, since their contribution will not help to select an error region.

Figure 2.11 illustrates this process.



Figure 2.11: Three snapshots in time of the volume growing process as the seeds propagate throughout the image

2.3.3 Smoothing

Finally, the best depth of each error region is smoothed by calculating the depth average of a window for every pixel in the image. In order to avoid that pixels belonging to a different surface contribute to this average, the depth of a neighbor pixel is used for averaging only if it lies within the error region of the pixel.

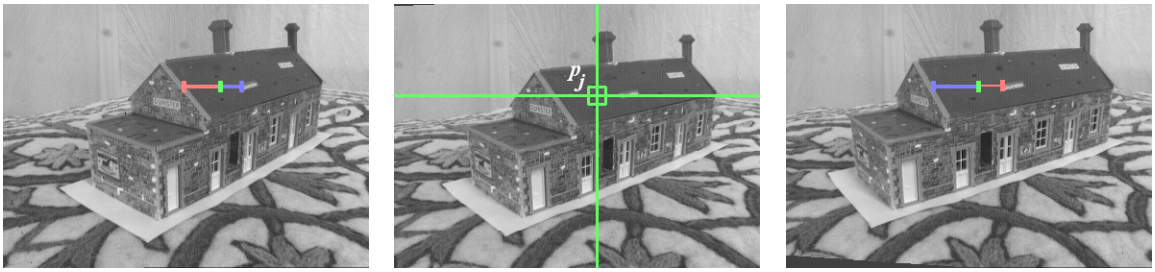
The smoothing process is repeated until the absolute change in the depth is smaller than a given threshold. Here, the larger the window size, the faster it will converge, but the final extracted surface will not be as smooth as it would be with a smaller window. This is because the process of averaging a large window decreases the individual contribution of all depths from all pixels in that window, which allows for large variation of depths in the resulting depth map.

2.4 Delta and Unconstrained Uncertainty

The result of the smoothing step of the range-space approach is used as input to our uncertainty rendering systems. The error regions that are selected by this processes are now called *depth uncertainty* regions. They will be the input of our 4-D and 3-D frameworks described in the next chapters.

The range-space search technique reviewed here is one of the possible techniques to calculate the depth uncertainty. Since both the 4-D and 3-D rendering approaches are very sensitive to how accurate the best estimated depth z_b is (since the main contribution of their splats comes from this depth), sometimes a different technique to estimate this depth may work better. Moreover, it can also be used as a seed in the volume growing step.

Alternatively, it is possible to search the depth uncertainty starting from z_b. First, we compute a 3-D depth map for each pixel of the input images using any kind of reconstruction method and use this depth as a ground truth. Later, through epipolar line searching, we find out how uncertain is this depth - the more homogeneous the region is, the higher the uncertainty depth range.



(a) Image j (b) Image i (c) Image $j+1$

Figure 2.12: Deriving the depth uncertainty from a ground truth depth map

Figure 2.12 illustrates this process. From a reference image i, we know the estimated generalized disparity[5] δ_i of a point p_i. Now, resembling the way that the range-space searching works, we look into all epipolar lines of images j, using an ellipse as a template.

In this way, the uncertainty region is computed *after* the depth estimation. Unlike the range-space approach, the uncertainty region grows in both directions (Figure 2.13), since the best depth is already known, which is equivalent to a local minimum, and the unknowns are the boundaries of the region.

There is an important difference between our algorithm and the original range-space approach. We limit our search for a given distance threshold ϵ along the estimated disparity, i.e., the uncertainty region will be inside the region defined by $\Delta = \delta_i \pm \epsilon$. For future reference, we call this technique *Delta Uncertainty*. Also, for comparison, we call the uncertainty region acquired from the original range-space search *Unconstrained Uncertainty*[6].

The threshold ϵ only works if the unknown actual depth is within the region defined by Δ. This is our assumption. Even though homogeneous areas may have a really high

[5] In this section, disparity always means *generalized* disparity.

[6] We call it *unconstrained* because the depth region obtained by this method may have any size. This can be observed particularly in homogeneous areas. The matching curve in those areas is flat, and it may never reach η.

uncertainty, as defined by the original range-space search algorithm, this assumption is true most of the time. The estimated depth of those areas are inferred from nearby features (like edges), where we have very low uncertainty. Therefore, the actual depth of a pixel in one of the homogeneous areas is usually not too far from its estimated depth.

In our implementation, we used $\epsilon = 0.5$ when the input images have a relatively high difference in angles, and $\epsilon = 1.0$ when they have similar configurations, such as the classic binocular stereo-matching setup.

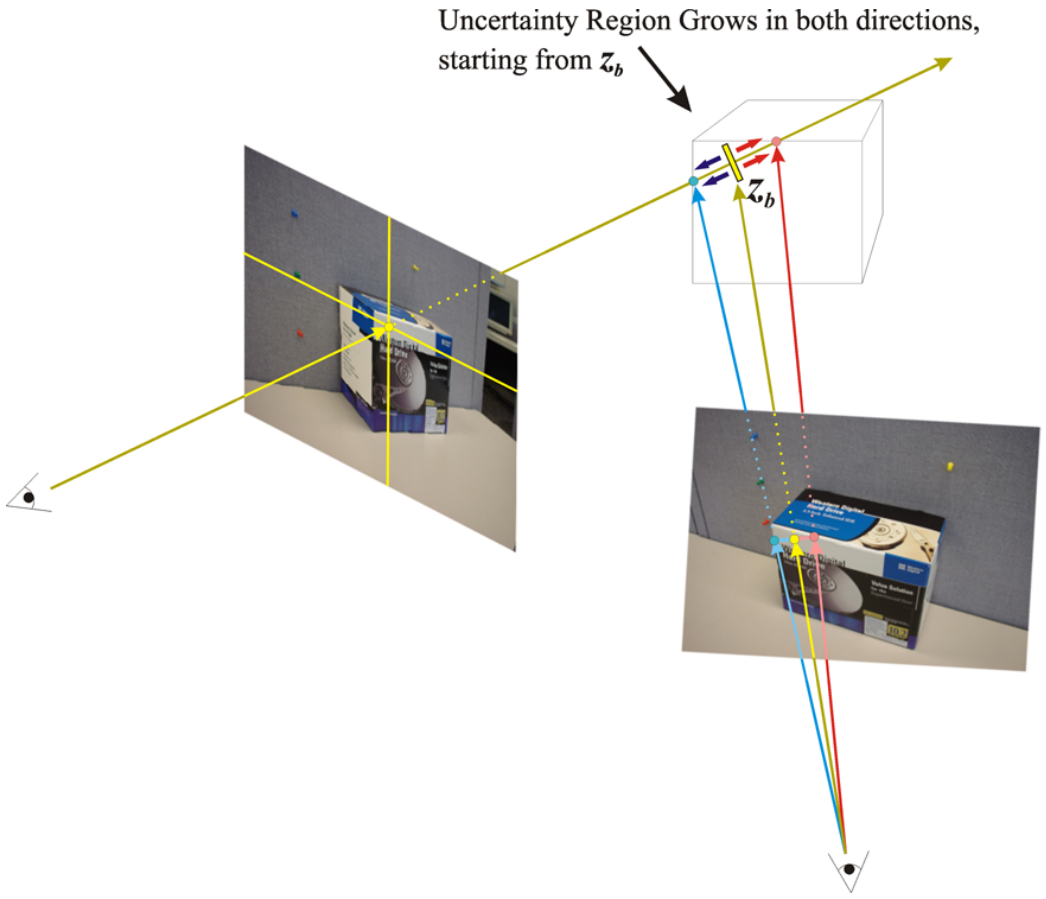


Figure 2.13: The process of looking up the depth uncertainty starts at the estimated depth and it grows in both directions of the ray

2.5 Summary

This chapter presented, first, a review of some basic concepts in computer vision. We showed that the epipolar geometry is important to create additional constraints to help

the searching and matching of pixels from one image to another. The match itself is based on specific metrics, one example being the sum of the squared color distances from a window of pixels. This window is located around the current pixel being queried in the epipolar line. Additional constraints may be obtained by feature extraction, usually from edges. Edge detection is commonly based on the derivatives of the image. They help to identify sudden intensity changes in the surrounding area of a pixel.

Next, we reviewed the range-space method, which is a technique to extract (and render) error regions from multiple images. After calculating the color-edge map images, two matching curves are assigned to each pixel in every input image. One for the color images and another for the color-edge map images. Then, we select the best matches - the one that an error region from the color-edge curve coincides with an error region from the color matching curves - to be the seeds. A volume growing phase propagates these seeds the homogeneous areas of the image. In the end, the final depth map is smoothed. An alternative to the range-search method is to estimate the depths using any 3-D reconstruction algorithm and use this depth as a ground truth to calculate the depth uncertainty.

Chapter 3

The 4-D Approach: Light Field Hyperlines

"You can't invent a design. You recognise it, in the fourth dimension. That is, with your blood and your bones, as well as with your eyes."

David Herbert Lawrence

In this chapter, we propose a new approach that combines light field and geometry into one common framework by thinking of rays as being emitted by scene points rather than being recorded by cameras. This mathematically equivalent treatment of rays is based on a dual space representation of the light field which formalizes the original idea of 4-D ray space.

3.1 Chapter Organization

A description of the light field duality, including the definition of camera and geometry hyperlines and the computation of the closest distance from a hyperline to a ray is presented in section 3.3. Next, camera hyperline rendering and dynamic focal planes are described in section 3.4. Geometry hyperline rendering, including both global and local ray space modeling are in section 3.5. Finally, the rendering of data with uncertainty information is the scope of section 3.6.

3.2 Overview of Approach

We model images with depth as point clouds. The points are called scene points. A light field is the set of rays either recorded by the cameras, or emitted by scene points. Both cameras (actually, projection centers) and scene points can be thought of as bundles of rays. Their roles are symmetric with respect to the rays. The essence of the proposed framework is treating the 4-D light field as a dual space of the 3-D world space so that the point-ray-camera relationship has a dual appearance. The concept of hyperline is defined as the intersection of two 4-D hyper planes in the dual light field. A hyperline has two degrees of freedom. It is called a "line" because two hyperlines intersect in a single a hyper point. Then, both cameras and scene points appear as hyperlines, and rays appear as hyper points on hyperlines. Hyperlines representing cameras (camera hyperlines) obtain their colors from the images and, therefore, they are mostly heterogeneous. Hyperlines representing scene points (geometry hyperlines) inherit their colors from the latter, thus are mostly homogeneous. In the proposed framework, ray selection becomes hyper point selection from hyperlines (either kind or both), as opposed to 3-D line selection.

Under this framework, rendering reduces to finding and blending the closest appropriate hyperlines to each virtual ray. Occlusion is handled by clustering nearby hyperlines. Hole filling is done simply by extending the contribution of each hyperline in ray space by a given distance threshold (note that we are not using normals or any other BRDF parameters).

We derive the light field duality concept using the 2-plane parameterization. Based on that, we propose a novel point cloud rendering technique for image-based rendering based on Light Field hyperlines. The blending weights are functions of the 4-D distance of light field rays from a desired output viewing ray.

We also address the problem of input images that do not have accurate depth information. As seen in chapter 2, the searching and matching steps of stereovision reconstruction provide more information than just the best matching depth. Thus, the reconstruction step produces a depth region, including the best matching depth. We use this information to create an entity in 4-D that aggregates all possible 3-D points in this depth

region (hyperline pencil). The rendering process takes into account proximity of the best matching depth and the size of this region.

3.3 The Dual Representation

A plenoptic function describes all the visible light from any viewing position [20]. This means that ideally, we should be able to obtain the color $\bar{C} = (C_r, C_g, C_b)$ for every given position (x, y, z), orientation (θ, ϕ) and at any given time T, *i.e.*, $L(x, y, z, \theta, \phi, T) \rightarrow \bar{C}$. However, it is hard to obtain a complete sample of the plenoptic function. In practice, only subfunctions of the generic plenoptic function are considered. For instance, McMillan and Bishop [20] represented it as a panoramic image centered at some fixed position and time.

We adopt the representation of Levoy and Hanrahan [1]. We show in this chapter that not only light rays, but also geometry, may be represented with this parameterization. We define geometry as a cloud of points located on the surfaces of objects in the scene subspaces [21]. We also present the dual representation between a 3-D object space and a 4-D ray space, originally presented by Chen, Hofsetz, Max, Hong, Liu and McGuinness [47]. We explain the concept of hyperlines, which represent points in 3-D either from camera centers of projection or from objects.

3.3.1 The Light Field Parameterization

The light field can be thought of as a parameterization of a plenoptic function at a fixed time using two parallel 2-D planes (Figure 3.1). Each ray has a 4-D representation (s, t, u, v) where (s, t) and (u, v) are the coordinates of the intersections of this ray with those planes. Moreover, s, t, u and v are restricted to an area that defines a convex quadrilateral in each plane. This defines a light slab, which can be intuitively thought as a beam of light entering one quadrilateral and exiting the other quadrilateral. Note that this simplification of the plenoptic function requires that the virtual camera be always located behind the $u-v$ plane.

There are many ways to obtain the light field. For instance, we can create a

structured light field where the input cameras' centers of projection are uniformly placed in one of the planes (camera plane) and the image planes are coincident with the other plane (focal plane) [1] [2]. In this case, light field rendering is easy and fast. For each output pixel the (s, t, u, v) of the virtual ray is found. Then, the closest four cameras positioned in the $s-t$ plane are found and a set of closest rays (4 or 16) in $u-v$ from those cameras are blended to produce the color of the pixel in question. The problem with this parameterization is that either the images have to be uniformly acquired or rays in all input images have be resampled to this uniform structure. And a large amount of memory is needed to store the additional and redundant information necessary to create the full camera matrix. The resampling process also causes some degree of loss of information because it cannot, in general, reproduce the input images [5].

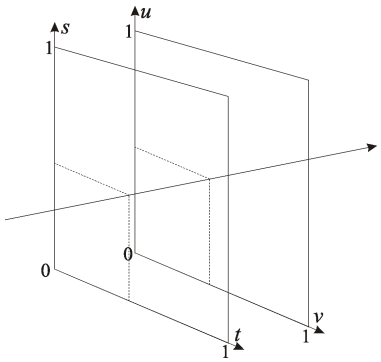
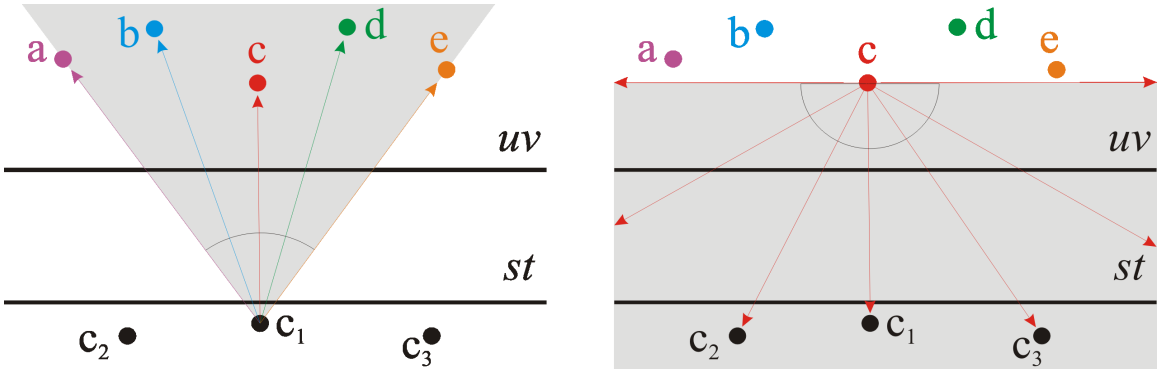


Figure 3.1: The Light Slab representation [1]

In our framework, an input camera image is represented as a finite and discrete set of rays (Figure 3.2a). Each ray may have a different color. Note that not only cameras but also geometry can be represented with the light field parameterization. If we consider Lambertian surfaces, we can think of each geometry point generating an infinite number of rays of the same color, where a subset of those rays intersect with our $s-t$ and $u-v$ planes (Figure 3.2b).

3.3.2 Hyperlines

Under the light field parameterization, we can think of its 4-D representation as a space filled with rays, a 4-D ray space. We can then define a dual relationship between



(a) (b)

Figure 3.2: c_1, c_2 and c_3 are cameras; a,b,c,d,e are geometry points. Both camera and geometry can be represented with Light Field parameterization. Cameras are represented as a finite, discrete and heterogeneous set of rays (a), while geometry points are represented as a continuous infinite and homogenous set of rays (b).

object space and ray space. Figure 3.3 shows a line Q in 3-D space coming from one object point $P = (x, y, z)$ and intersecting both planes at (s, t) and (u, v), can be represented as a point $P = [s, t, u, v]^T$ in a 4-D ray space, which we name the ray space in this paper. From simple trigonometry, or similar triangles, we have:

$$\frac{s-x}{u-x} = \frac{t-y}{v-y} = \frac{z-z_{st}}{z-z_{uv}}$$

which has two equivalent matrix forms:

$$\begin{bmatrix} z_{uv}-z_{st} & 0 & s-u \\ 0 & z_{uv}-z_{st} & t-v \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} sz_{uv}-uz_{st} \\ tz_{uv}-vz_{st} \end{bmatrix} \tag{3.1}$$

$$\begin{bmatrix} z-z_{uv} & 0 & z_{st}-z & 0 \\ 0 & z-z_{uv} & 0 & z_{st}-z \end{bmatrix} \begin{bmatrix} s \\ t \\ u \\ v \end{bmatrix} = \begin{bmatrix} x(z_{st}-z_{uv}) \\ y(z_{st}-z_{uv}) \end{bmatrix} \tag{3.2}$$

Equation 3.2 describes, in fact, the intersection of two 4-D hyperplanes, and thus it represents a *hyperline*, which is a 2-D entity in a 4-D space. A general form of equation 3.2 is:

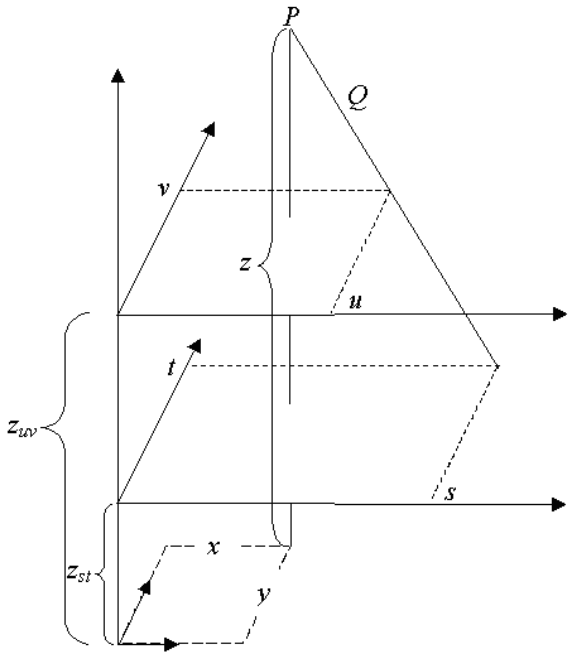


Figure 3.3: The two-plane- parameterization and derivation of dual relationship.

$$\begin{bmatrix} a & 0 & b & 0 \\ 0 & a & 0 & b \end{bmatrix} \begin{bmatrix} s \\ t \\ u \\ v \end{bmatrix} = \begin{bmatrix} c \\ d \end{bmatrix} \tag{3.3}$$

where a and b cannot be zero at the same time.

Using this relationship, we define a *camera hyperline* as the ray space representation of a finite and heterogeneous bundle of rays in object space around camera centers. Also, a *geometry hyperline* is the ray space representation of an infinite and homogeneous bundle of rays in object space around scene points. Furthermore, both camera and geometry hyperlines can be represented with equation 3.3. A geometry hyperline and a camera hyperline intersect at a 4-D point which is the ray where the geometry point is seen by the camera.

It is also important to notice that because we think of the geometry hyperlines as infinite entities, we are not restricted to the light slab quadrilateral constraints. Assuming we have a camera positioned anywhere behind the $u - v$ plane, there always is exactly one ray that links the camera's center of projection to the geometry point in front of the $u - v$

plane. This ray can be used for virtual view synthesis.

The dual relationship between object space and ray space is summarized in Table 3.1.

Object Space	**Ray Space**
Point	Hyperline
Line	Point
Bundle of lines from surface point	Geometry Hyperline
Bundle of lines from camera center	Camera Hyperline
Depth	Slope

Table 3.1: The Dual Relationship

Figure 3.4 shows a 2-D scene as seen by a 1-D camera, and its respective 2-D ray space. Notice that the smaller the slope, the closer the hyperline is to the $u - v$ plane. All hyperlines that have the same depth in object space are parallel in ray space.

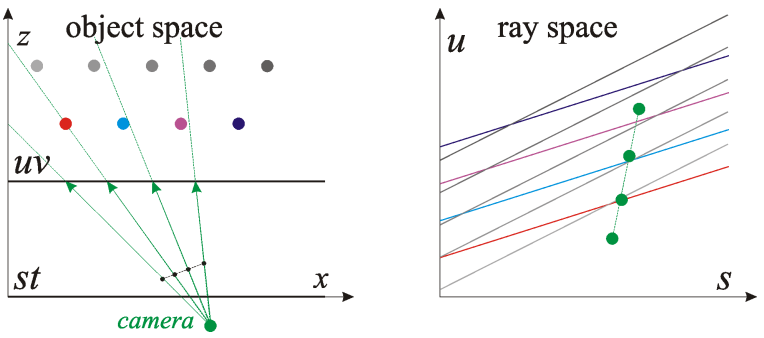


Figure 3.4: 2-D ray space of a 2-D scene as seen by a 1-D camera

3.3.3 Ray-Hyperline Distance Calculation

The distance between a hyperline $l_i = (a_i, b_i, c_i, d_i)$ and a virtual ray $r = (s_r, t_r, u_r, v_r)$ is the smallest distance between the any ray $r_i = (s_i, t_i, u_i, v_i)$ on l_i from r. One intuitive way to solve this is to minimize the squared 4-D distance between all r_i and r:

$$d^2 = (s_i - s_r)^2 + (t_i - t_r)^2 + (u_i - u_r)^2 + (v_i - v_r)^2 \tag{3.4}$$

subject to $a_i s_i + b_i u_i = c_i$ and $a_i t_i + b_i v_i = d_i$. The solution is

$$r_i = \begin{bmatrix} \frac{b_i^2 s_r - a_i b_i u_r + a_i c_i}{a_i^2 + b_i^2} \\ \frac{b_i^2 t_r - a_i b_i v_r + a_i d_i}{a_i^2 + b_i^2} \\ \frac{a_i^2 u_r - a_i b_i s_r + b_i c_i}{a_i^2 + b_i^2} \\ \frac{a_i^2 v_r - a_i b_i t_r + b_i d_i}{a_i^2 + b_i^2} \end{bmatrix}$$

and

$$d_{\min} = \sqrt{\frac{(a_i s_r + b_i u_r - c_i)^2 + (a_i t_r + b_i v_r - d_i)^2}{a_i^2 + b_i^2}} \tag{3.5}$$

From now on, we use a quadruple (a, b, c, d) to represent a hyperline. Obviously, a hyperline represented by 3.3 has two degrees of freedom. The intersection of two hyperlines is that of four hyper planes, thus a hyper point (zero degrees of freedom), unless the hyperlines are parallel. The dual relationship between world space and light field is described by the following three remarks:

1. World space and light field are dual spaces: points in one space correspond to lines or hyperlines in the other space, and vice versa. For example, the point P in world space corresponds to the hyperline in the light field whose equation is given by equation 3.2; and the point Q in the light field corresponds to the line in world space whose equation is given by equation 3.1.

2. A bundle of lines in world space correspond to a set of co-hyperlinear points in the light field. For example, when x, y, z are fixed while s, t, u, v are varying, equation 3.1 describes a bundle of lines in world space all passing through P, and equation 3.2 describes a set of co-hyperlinear points in R^4.

3. A set of collinear points in world space corresponds to a bundle of hyperlines in the light field. For example, when s, t, u, v are fixed while x, y, z are varying, equation 3.1 describes a set of collinear points and equation 3.2 describes a bundle of hyperlines all passing through the 4-D point Q.

We would like to point out the relationship between the slope k of a hyperline and the depth z of the corresponding 3-D point: $k = -a/b = (z - z_{uv})/(z - z_{st})$. Particularly,

for two 3-D points of same depth, their hyperlines are parallel. This observation was also made by Gu et al. in [24] where they characterized a bundle of lines in the world space as a 2-D affine subspace in the light field. They did not explicitly make use of the duality between the 3-D line bundle and the 4-D hyperline.

3.4 Camera Hyperline Rendering

Light field rendering using camera hyperlines is rather straightforward. First, convert input images to camera hyperlines. Second, for each virtual ray, select hyperlines by minimizing a cost function. Third, for the same virtual ray, choose and then blend rays from the just selected hyperlines. Notice since there is a one-to-one mapping between a camera hyperlines and an image plane, there is no need to perform resampling on the camera hyperlines.

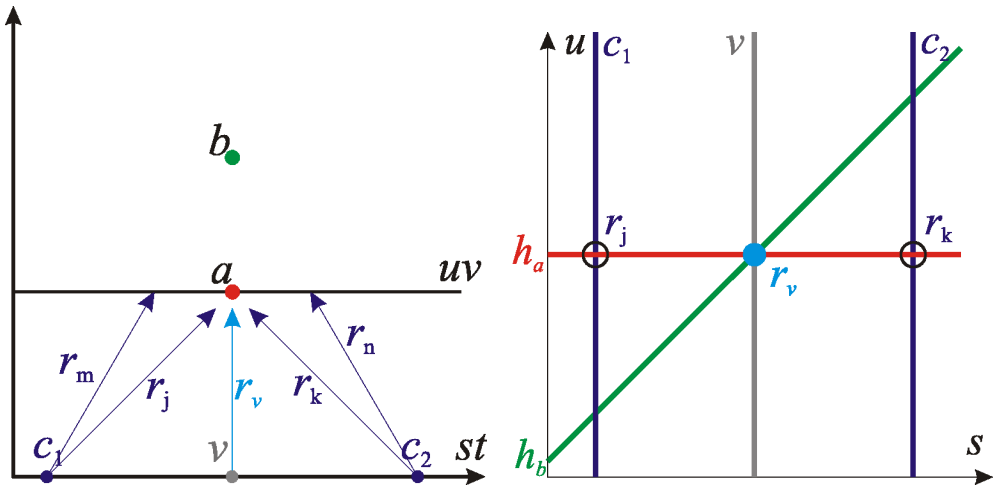
Based on 3.5 the first few source rays with the shortest 4-D distance are chosen, and blended according to the inverse distance. To see the geometrical meaning of this solution, place all cameras on the $s-t$ plane, i.e., $z_c = z_{st}$. As a result, $x_c = s_i$, $y_c = t_i$. The ray with the shortest distance is $r_i = (c_i/a_i,\ d_i/a_i, u_r, v_r) = (s_c, t_c, u_r, v_r)$ which coincides with r on the $u-v$ plane as they have the same $u-v$ coordinates. Since all selected rays pass through (u_r, v_r, z_{uv}), their costs are really dependent on their locations on the $s-t$ plane. This is the basis for the quadrilinear interpolation adopted in [1].

3.4.1 Dynamic Focal Plane

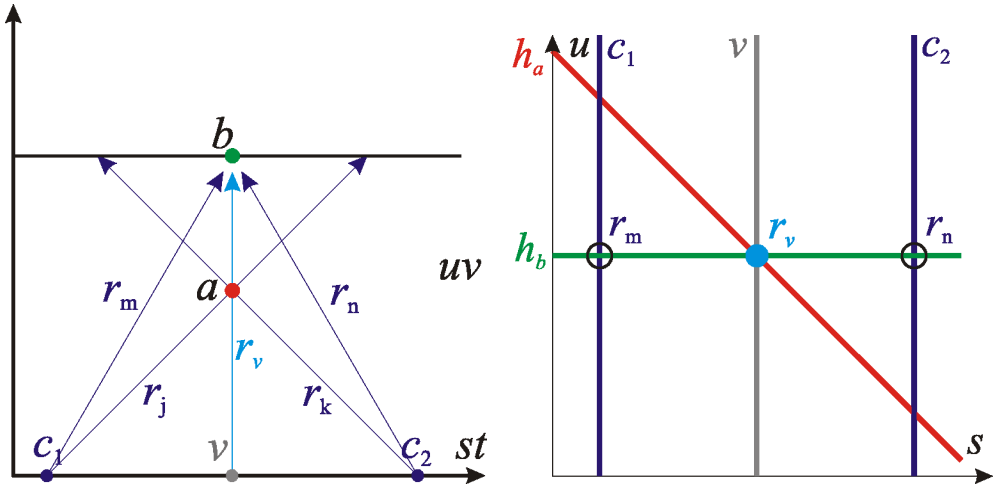
Since the $u-v$ plane is the focus plane, all objects at this depth will appear sharp, and objects at different depths will look blurred (Figure 3.5). Our focal plane is defined by z_{uv}, which cannot be modified once the hyperline coefficients a, b, c and d are calculated. For the purpose of creating a dynamic focal plane, let us analyze the following derivation. Assuming that

$$f = z_{uv} - z_{st} \tag{3.6}$$

and rearranging equation 3.6 for z_{uv} instead



(a) Object space, focus on a (b) Ray space, focus on a



(c) Object space, focus on b (d) Ray space, focus on b

Figure 3.5: How the position of the $u - v$ plane affects rendering. c_1 and c_2 are input cameras, v is the virtual camera. r_m and r_j are light rays from camera c_1. r_k and r_n are light rays from camera c_2. h_a and h_b are the geometry hyperlines from objects a and b, respectively. (a) Focal plane is on object a. (b) The rays used to calculate the color of the virtual ray r_v are r_j and r_k. (c) Focal plane is on object b. (b) The rays used to calculate the color of the virtual ray r_v are r_m and r_n. In (d), object a will appear blurred. Note that the actual geometry hyperlines h_a and h_b are not known, they are only shown for illustration purposes. If h_a and h_b are known, we can represent them as geometry hyperlines and the rendering process works differently, see section 3.5

$$z_{uv} = f - z_s t \tag{3.7}$$

we can then rewrite equation 3.2:

$$\begin{bmatrix} z + z_{st} - f & 0 & z_{st} - z & 0 \\ 0 & z + z_{st} - f & 0 & z_{st} - z \end{bmatrix} \begin{bmatrix} s \\ t \\ u \\ v \end{bmatrix} = \begin{bmatrix} -fx \\ -fy \end{bmatrix} \tag{3.8}$$

which can be represented as

$$\begin{bmatrix} a' - f & 0 & b & 0 \\ 0 & a' - f & 0 & b \end{bmatrix} \begin{bmatrix} s \\ t \\ u \\ v \end{bmatrix} = \begin{bmatrix} -fc' \\ -fd' \end{bmatrix} \tag{3.9}$$

where $a' = z + z_{st}$, $c' = x$ and $d' = y$.

Then, similar to what we did in section 3.3.3, we have to find the closest virtual ray $r = (s_r, t_r, u_r, v_r)$ to this hyperline. The squared distance between r and any ray $r_i = (s_i, t_i, u_i, v_u)$ on hyperline l_i is:

$$d^2 = (s_i - s_r)^2 + (t_i - t_r)^2 + (u_i - u_r)^2 + (v_i - v_r)^2 \tag{3.10}$$

subject to $(a'_i - f)s_i + b_i u_i = -fc'_i$ and $(a'_i - f)t_i + b_i v_i = -fd'_i$. The solution is:

$$r_i = \begin{bmatrix} \frac{b_i^2 s_r - (a'_i - f) b_i u_r - (a'_i - f) f c'_i}{(a'_i - f)^2 + b_i^2} \\ \frac{b_i^2 t_r - (a'_i - f) b_i v_r - (a'_i - f) f d'_i}{(a'_i - f)^2 + b_i^2} \\ \frac{(a'_i - f)^2 u_r - (a'_i - f) b_i s_r - b_i f c'_i}{(a'_i - f)^2 + b_i^2} \\ \frac{(a'_i - f)^2 v_r - (a'_i - f) b_i t_r - b_i f d'_i}{(a'_i - f)^2 + b_i^2} \end{bmatrix}$$

and

$$d_{\min} = \sqrt{\frac{((a'_i - f)s_r + b_i u'_r + f c'_i)^2 + ((a'_i - f)t_r + b_i v'_r + f d'_i)^2}{(a'_i - f)^2 + b_i^2}} \tag{3.11}$$

See below for the values of u'_r and v'_r.

What exactly is the meaning of f? From equation 3.6 we see that f is the difference between the planes z_{uv} and z_{st}. Since we removed our focal plane z_{uv} from our original equation 3.2, we can now specify a *dynamic* focal plane based on f, which is relative to the camera plane z_{st}.

However, although equation 3.11 is correct, its use is not straightforward. z_{st} and z_{uv} are the basis of our two-plane parameterization scheme. Obviously, if we change our focal plane, *the parameterization changes*, and the virtual ray r has to be recalculated every time the focal plane changes.

It turns out that we can include this reparameterization in equation 3.11. The idea is as follows. Every virtual ray r is parameterized according to two fixed 2-D planes. s_r and t_r are still defined by z_{st}, since it does not change. u_r and v_r will be defined by a plane located[1] at $z_{st} + 1$. That is why equation 3.11 has u'_r and v'_r instead of u_r and v_r. The values of u'_r and v'_r are:

$$u'_r = f(u_r - s_r) + s_r$$

$$v'_r = f(v_r - t_r) + t_r$$

Figure 3.6 illustrates the derivation of u'_r.

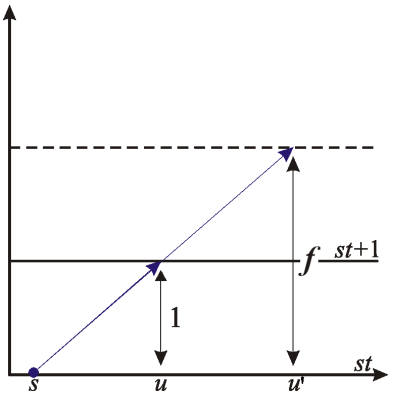


Figure 3.6: Dynamic focal plane reparameterization $\frac{1}{u-s} = \frac{f}{u'-s}$

Another way to implement the dynamic focus plane for camera hyperlines was

[1] Notice that this configuration is equivalent to $f = 1$

presented by Chen et. al [48]. Here, an extra requirement is that the selected source ray has a direction similar to the virtual one. We therefore minimize

$$d^2 = \{(s_i - s_r)^2 + (t_i - t_r)^2 + (u_i - u_r)^2 + (v_i - v_r)^2\} + \beta\{(s_i - s_r - u_i + u_r)^2 + (t_i - t_r - v_i + v_r)^2\} \tag{3.12}$$

subject to $a_i s_i + b_i u_i = c_i$ and $a_i t_i + b_i v_i = d_i$. The first term accounts for the light field distance as before. The second term accounts for orientation difference in world space, which becomes zero when r_i and r are parallel. β is the balancing factor giving the relative weight for the direction disagreement and the length of the common perpendicular between the two rays. Solving 3.12 results in

$$r_i = \begin{bmatrix} \frac{b_i^2((1+\beta)s_r - \beta u_r) - a_i b_i((1+\beta)u_r - \beta s_r) + c_i((1+\beta)a_i + \beta b_i)}{\beta(a_i+b_i)^2 + a_i^2 + b_i^2} \\ \frac{b_i^2((1+\beta)t_r - \beta v_r) - a_i b_i((1+\beta)v_r - \beta t_r) + d_i((1+\beta)a_i + \beta b_i)}{\beta(a_i+b_i)^2 + a_i^2 + b_i^2} \\ \frac{a_i^2((1+\beta)u_r - \beta s_r) - a_i b_i((1+\beta)s_r - \beta u_r) + c_i((1+\beta)b_i + \beta a_i)}{\beta(a_i+b_i)^2 + a_i^2 + b_i^2} \\ \frac{a_i^2((1+\beta)v_r - \beta t_r) - a_i b_i((1+\beta)t_r - \beta v_r) + d_i((1+\beta)b_i + \beta a_i)}{\beta(a_i+b_i)^2 + a_i^2 + b_i^2} \end{bmatrix}$$

and

$$d_{\min}^2 = \frac{(1+2\beta)\left((a_i s_r + b_i u_r - c_i)^2 + (a_i t_r + b_i v_r - d_i)^2\right)}{2\left(\beta(a_i+b_i)^2 + a_i^2 + b_i^2\right)}. \tag{3.13}$$

Unlike equation 3.11 where f is fixed for a specific focal depth, each camera hyperline now has a different β for the same depth focus plane. However, they are linearly related if the selected source rays r_i and r intersect at the same depth. The depth of the intersection is given by:

$$z(\beta) = \frac{a_i z_{uv} - b_i z_{st} - \beta(z_{uv} - z_{st})^2}{a_i - b_i}.$$

Therefore, β can be chosen so that the intersection is at a desired depth. If β=0, the sufficient and necessary condition for $z(0) = z_{uv}$ is b_i=0 . Also, $z(\infty) = \infty$. This is equivalent to saying that the selected source ray is parallel to the virtual one, a result of assigning the orientation term infinite weight.

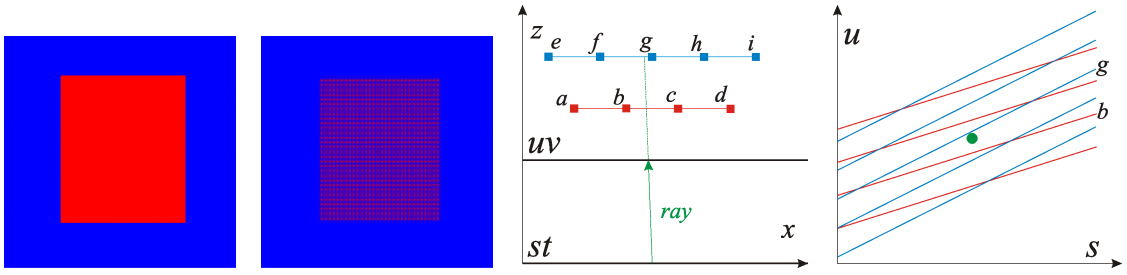
Unfortunately, the use of β is not intuitive. As we said before, β works by scaling a penalty factor given by the direction disagreement of the two rays. Informally, one could say that the dynamic focal plane is actually a byproduct of using β.

3.5 Geometry Hyperline Rendering

In order to render a set of geometry hyperlines derived from a point cloud we have to first choose which geometry hyperlines are close enough to our virtual ray. Second, we have to solve ambiguity between foreground and background rays. This section introduces two thresholds, the 4-D distance threshold and the depth window threshold, that solve these problems. Finally, in order to make the algorithm more generic, the restriction on the virtual camera location can be removed if we use local ray spaces.

3.5.1 Ray Selection

Naively blending the closest set of geometry hyperline rays (based on equation 3.5) does not handle occlusion correctly. For example, in Figure 3.7, the foreground red object may appear transparent. To deal with this, some intrinsic properties of the two-plane parameterization are considered. It turns out that the *slopes* of the geometry hyperlines, which are equivalent to their depth from the $u-v$ plane, indicate how they are ordered. For points behind the $u-v$ plane, the smaller the slope, the closer to the $u-v$ plane they are.



(a) (b) (c) (d)

Figure 3.7: (a) A red square in front of a blue square. (b) Incorrect rendering where the front square appears transparent. (c) Vicinity of a virtual ray representing one pixel being reconstructed. (d) A 2-D slice of the corresponding light field. The virtual ray appears as a small green dot. The hyperline closest to the virtual ray belongs to the background.

Now, let us look at a vicinity of a virtual ray (Figure 3.7c,d), which includes 4 geometry hyperlines from the foreground and 5 geometry hyperlines from the background.

If we choose the closest geometry hyperlines based on the 4-D distance, the red square looks transparent. Instead, we should search the geometry hyperlines front-to-back sorted by *slope*. Therefore, as a pre-processing step, all geometry hyperlines are ordered by slope, which helps us to cluster them by depth.

3.5.2 Depth Clustering

One way of creating clusters is to use a *depth window threshold* V_d that will select only the best hyperlines up to a specific depth. All hyperlines where the current depth is smaller than the depth of the best solution so far plus V_d are used for blending.

For example, Figure 3.8 shows the 4-D distance from the virtual ray in 3.7c to the closest rays of each geometry hyperline. Since hyperlines are clustered by depth, the search stops at hyperline b depth plus V_d (in this case $V_d = 1$). The points from the background do not contribute to this virtual ray.

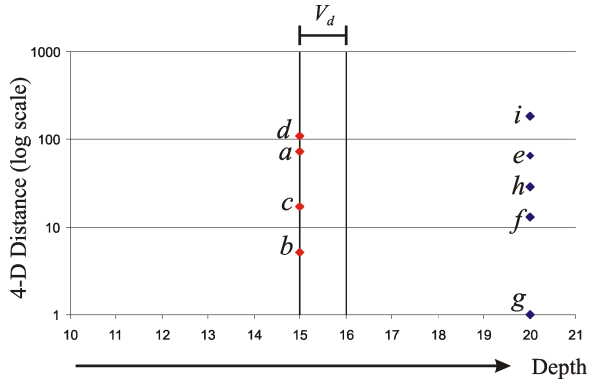


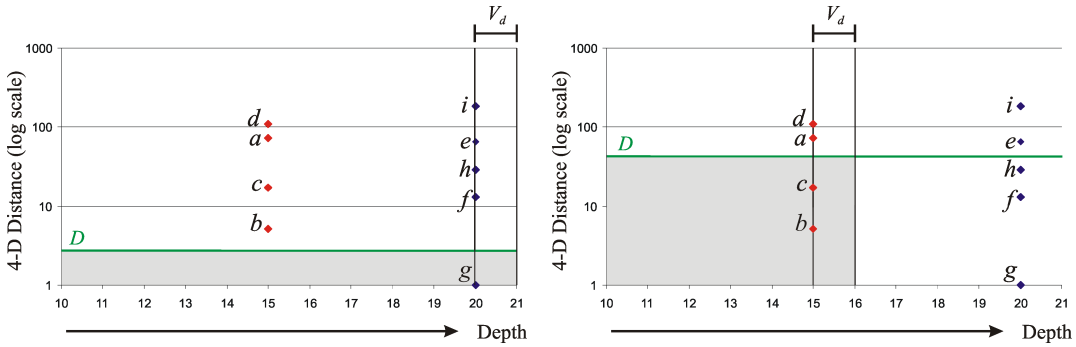
Figure 3.8: Graph of the 4-D distance of all geometry *hyperlines* (Figure 3.7c) from the virtual ray

3.5.3 Hole Filling

Exact solutions – the virtual ray equals to a ray on a hyperline – are rare. If only those are chosen for blending, the image would look like a simple point cloud. Instead, we have to take into account rays that are close, within a given a 4-D distance threshold D of the hyperline. The "best" choice of D depends on the density of the input points. However, D can be also used for hole filling. The larger D is, the larger is the hyperline shape (which

is circular, see section 3.5.4 below) in the final image.

Figure 3.8 shows how both D and V_d are necessary to solve occlusion and to fill holes. If D is too small (3.3a) only hyperlines whose rays are very close to the virtual ray are chosen (hyperline g), causing background leaks to the foreground. In this case V_d does not help because hyperlines a, b, c and d are not even considered for blending. If D is larger (3.3b), the foreground hyperlines c and b are selected for blending. Even though h, f and g all have rays whose distances are below D they are not selected because of V_d.



(a) g is selected (b) c and b are selected

Figure 3.9: Same as Figure 3.8 but now including the 4-D distance threshold D. Even with a fixed V_d different choices of D yield different geometry hyperlines for blending. Only hyperline rays inside the gray area are blended.

3.5.4 Optimizations

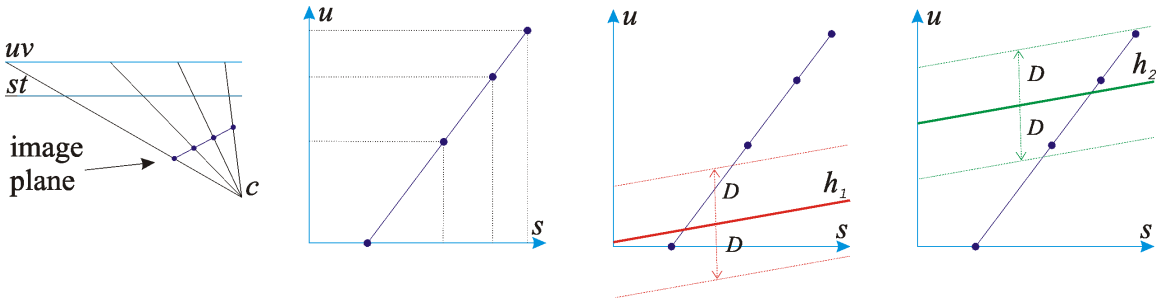
The hyperline renderer is basically a pixel shader. Each pixel defines a virtual ray, and for each virtual ray we have to calculate its distance from all geometry hyperlines. If we have a huge number of geometry hyperlines the rendering will be too slow. Therefore we must restrict our search only to those geometry hyperlines that may contribute to the color of each virtual ray.

This can be done by a *warping* step that defines to which pixel each hyperline will most contribute. However, frequently, a hyperline contributes to not only one pixel, but also to several pixels surrounding it.

How big this coverage area should be? The simplest way of determining this is to overshoot and add a fixed radius for all hyperlines. This is better than calculating the

distance of all hyperlines to each virtual ray, but it is still very slow. This is because each hyperline has a different area of contribution. Even if one hyperline is at the same depth as another, they may have a completely different area of contribution, due to perspective foreshortening (see Figure 3.10).

In fact, the tighter is this search area, the faster is the algorithm. Our solution is the following: for each input pixel, we warp not the pixel but a quadrilateral parallel to the input image plane. This quadrilateral is defined differently for each hyperline. Although the intersection between a geometry hyperline and a camera hyperline is a point, the introduction of the 4-D distance threshold D in the equation creates a circular shape that is parallel to the $u-v$ and $s-t$ planes (Figure 3.11b), and the quadrilateral contains this circle.



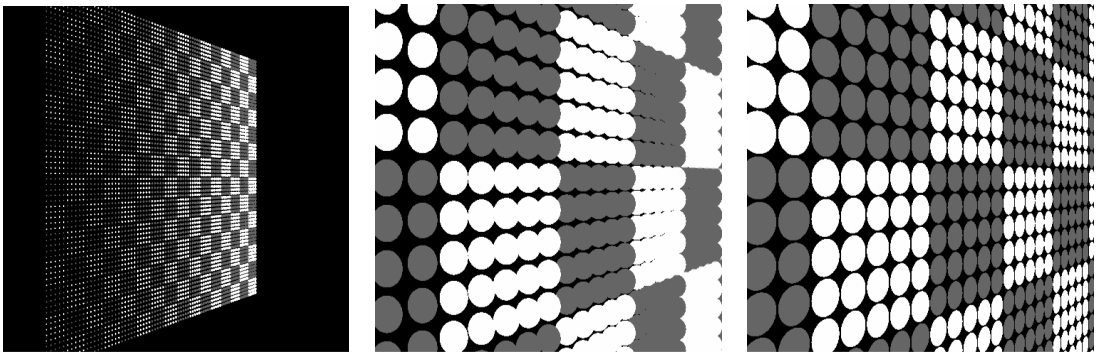
(a) (b) (c) (d)

Figure 3.10: (a) 2-D slice of Euclidian space. c is our virtual camera. (b) 2-D slice of ray space with the camera hyperline. Notice how the distance between virtual rays is not constant. (c) and (d) depict two geometry hyperlines in ray space and their relationship with the camera hyperline. Although D is constant and h_1 and h_2 have the same slope, h_1 only contributes for one virtual ray, while h_2 spans three virtual rays.

3.5.5 Local Ray Space

A global ray space is not enough to render an arbitrary view from the model. The usual solution of the original light field papers is to use many light slabs, usually four, each one having its own two-plane parameterization. However, this increases the computational cost and it produces artifacts at the boundaries of the light slabs [26].

Instead, we have chosen to use a *local ray space* for each input image. In this



(a) Input Image (b) Global Ray Space (c) Local Ray Space

Figure 3.11: Comparison between using global and local ray space

case, each input image is parameterized according to its own camera parameters. The $u-v$ plane is located at the image plane and the $s-t$ plane is located at the center of projection, parallel to the $u-v$ plane. All virtual rays now have to be converted to each local ray space in order to calculate the 4-D distance. As in global ray space, the projected circular shape is still parallel to the $u-v$ and $s-t$ planes. However, now it is possible to reconstruct the input images (given the right parameters) since the shape is also parallel to the plane of the input image (Figure 3.11c).

The local ray space idea has two problems. First, let us consider all hyperlines generated from one specific input image. As the virtual ray and the image plane become parallel, the hyperline footprint areas approach zero. Although it is desirable to decrease the contribution of an input point as its original viewing vector gets farther away from the virtual ray, simply making it disappear is not visually pleasant. A simple solution to avoid this problem is to have at least two views of each surface point.

Second, depth clustering by sorting hyperline slopes does not work anymore. However, we still need to cluster nearby hyperlines. Now we need a more sophisticated sorting mechanism. We have to keep a global sorted list for *every* axis (X, Y and Z), in ascending order. Then, for each virtual ray, we pick the axis list corresponding to the component of the ray's direction vector with largest absolute value. Depending on whether the original component value is positive or negative, we search the list front-to-back or back-to-front, respectively. We cluster the hyperlines now by coordinate depth. Clustering still works as

in global ray space, though now we have three sorted lists to choose from.

In the end, the added cost of using local ray space is much less than using multiple light slabs. Although we have now to convert each virtual ray for every input image ray space, the added cost of this conversion per input image is much less than rendering each input image four or more times if global ray space was used. Moreover, the use of local ray spaces removes any restriction in the virtual camera position.

3.5.6 Pseudo Code

The basic pseudo code for geometry hyperline rendering using local ray space is below. Here, *BestHyperline*, *ColorSum* and *WeightSum* are local variables of each output pixel. $g(h_{dist})$ is the blending weight.

```
For each axis,
   sort all hyperlines in order of the coordinate values

For each new viewpoint,

(1)    For each virtual ray,
         calculate which axis and sorting direction to chose

       /* Warping */
(2)    For each input image i,
       Initialize ColorSumxy, WeightSumxy to zero
       For each hyperline h   Image(i)
          Warp h to ( hx, hy ) and calculate a mask radius hradius


       /* Calculate closest hyperlines for blending */
(3)    For each axis a (X, Y and Z)
        For each hyperline h //(in ascending or descending order,
                             // depending on the virtual camera position)
           For each virtual ray r(x,y) in mask ( hxhradius, hyhradius)
              whose axis calculated in (1) is equal to a,
                hdist = 4D distance between h and r(x,y)
                if (hdist < distance threshold D)  and
                  (hslope < [ BestHyperlineslope + Vd ] )
                  ColorSumxy = ColorSumxy + hcolor * BlendFunction(hdist)
                  WeightSumxy= WeightSumxy+ g(hdist)
                  if  ( hdist < BestHyperlinedist ) then BestHyperline = h


       /* Finalizing Colors */
(4)    For each output pixel (x,y),
         if WeightSumxy > 0
            Colorxy = ColorSumxy / WeightSumxy
         otherwise Colorxy = 0 // if no hyperlines contributed to this pixel

       Show Image
```

Figure 3.12: Pseudocode for geometry hyperline rendering

We have tried two different weights for our blend function:

$$g(h_{dist}) = 1/(h_{dist})^2 \tag{3.14}$$

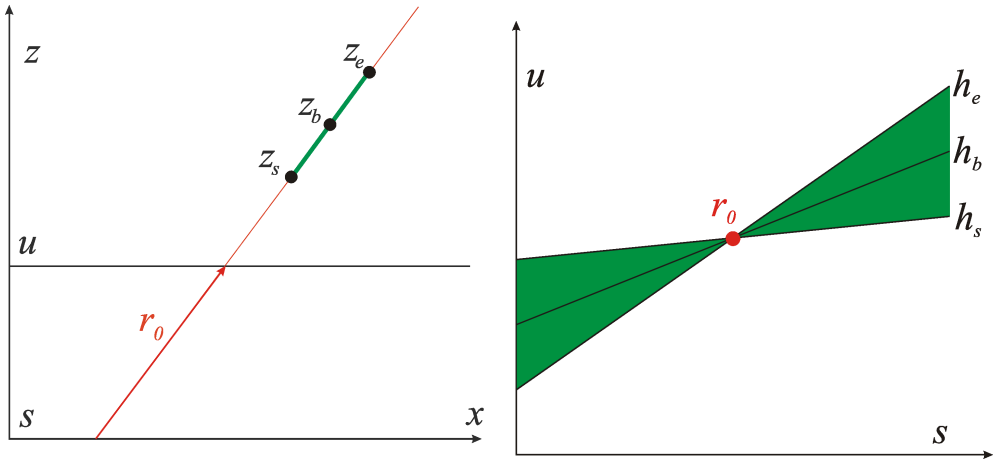
and

$$g(h_{dist}) = ((D - h_{dist})/D) \tag{3.15}$$

where D is the distance threshold. In general, the latter produces a smoother image than the former.

3.6 Modeling Uncertainty in 4-D Rayspace

Although accurate depth is necessary to create a hyperline, it is possible to consider *depth uncertainty* in our framework if we consider it as a set of continuous geometry hyperlines. Each input ray $r_0\,(s_0, t_0, u_0, v_0)$ is represented as a region bounded by z_s (start) and z_e (end), $z_s < z_e$, and also with the best matching depth z_b. Notice that all geometry hyperlines of interest share the same 4-D point corresponding to the input ray r_0 (Figure 3.13). Therefore, we call this entity a *hyperline pencil*, or *hyperpencil*.



(a) Depth region in object space. (b) Depth region in rayspace.

Figure 3.13: Example of a Hyperline Pencil. h_e, h_b and h_s are geometry hyperlines for depths z_b, z_s and z_e respectively.

There are three important considerations that we have to take into account when we render a hyperline pencil.

1. We have to consider the 4-D distance from the virtual ray to each hyperline that belongs to the pencil (equations 3.16 and 3.17).

2. The weight contribution of those hyperlines must be attenuated based on how far its z is from z_b (equations 3.18 and 3.19).

3. Hyperline pencils with low uncertainty should weight more than pencils with high uncertainty (equation 3.20). Low uncertainty means that the intersections between a geometry hyperline with camera hyperlines are well known. High uncertainty indicates that no close intersection is found.

3.6.1 The Weight Function

Since r_0 is known, we have $c = as_0 + bu_0$ and $d = at_0 + bv_0$. Recall that for local ray space the $s-t$ plane is at the input camera's center of projection, which means that $z_{st} = 0$, $s_0 = 0$ and $t_0 = 0$. Considering that $a = z - z_{uv}$ and $b = z_{st} - z$, the 4-D distance from a virtual ray *r(s,t,u,v)* to any geometry hyperline with depth z that share r_0 is (based on equation 3.5):

$$f(z,r) = d_{\min} = \sqrt{\frac{(-uz + u_0 z + s(z - z_{uv})^2) + (-vz + v_0 z + t(z - z_{uv})^2)}{z^2 + (z - z_{uv})^2}} \tag{3.16}$$

Using equation 3.15, the blending function for a specific depth z can be rewritten as:

$$g(z,r) = (D - f(z,r))/D \tag{3.17}$$

Now we need to attenuate $g(z,r)$ based on how far z is from z_b. This is achieved by multiplying $g(z,r)$ by an attenuation function $h(z)$:

$$w(r,z) = g(z,r)h(z) \tag{3.18}$$

with

$$h(z) = \begin{cases} z \leq z_b & e^{-q\left(\frac{z_b - z}{z_b - z_s}\right)} \\ z > z_b & e^{-q\left(\frac{z_b - z}{z_b - z_e}\right)} \end{cases} \tag{3.19}$$

where q is a constant that helps to control how much contribution we want from hyperlines whose depth are far from z_b.

Now, in order to give less weight to hyperline pencils that are more uncertain, we need to assign an *uncertainty value* u_i to each hyperline pencil i, where

$$u_i = \frac{z_e^i - z_s^i}{\max(z_e^j - z_s^j)}, \text{ for any hyperline } j \tag{3.20}$$

and the weight function for hyperline pencil i and a specific z is:

$$w_i(r, z) = (1 - u_i) g_i(z, r) h_i(z) \tag{3.21}$$

Equation 3.21 calculates only the weight of a specific depth z. If we want to consider the weights of all z positions along hyperpencil i, one possible solution is to integrate equation 3.21 for an interval $[z_0, z_1]$ where z_0 and z_1 are the solutions for $g(z, r) = D$ and $z_s \leq z_0 \leq z_1 \leq z_e$. The weight function for a given hyperline pencil i and a virtual ray r is:

$$w_i(r) = (1 - u_i) \int_{z_0}^{z_1} g_i(z, r) h_i(z) dz \tag{3.22}$$

The integral is numerically computed by Gaussian integration. Since equation 3.22 is not continuous in its first derivative (there is a 'breakpoint' in z_b), the integration is done separately for each part (z_0 to z_b and z_b to z_1).

3.6.2 Occlusion

The original geometry hyperline rendering algorithm relies on depth clustering to solve occlusion. However, hyperline pencils do not have accurate depth, so, as was stated before, depth clustering does not work. The solution for this problem is to consider the weight (equation 3.22) to be *opacity* (after clamping equation 3.22 to 1).

Now, in order to accumulate opacity values, the hyperline pencils must also be ordered somehow. Since its depth is uncertain, we use the best information available – the best matching depth z_b - for sorting, and back to front compositing.

The color and opacity accumulation for each virtual ray r, that represents an output pixel, is done as following:

$$C_{Acc} = C_{Acc} + C_i \alpha_i(r)(1 - \alpha_{Acc}) \tag{3.23}$$

$$\alpha_{Acc} = \alpha_{Acc} + \alpha_i(r)(1 - \alpha_{Acc}) \tag{3.24}$$

where $\alpha_i(r) = w_i(r)$, C_{Acc} is the color accumulated and α_{Acc} is the opacity accumulated. After accumulation, the final color of the pixel is C_{Acc}/α_{Acc}.

In some cases – for example in homogenous regions where the uncertainty is very high - depth clustering is still needed. So we still use V_d, though now its value should be much higher than usual. Here, the depth used for depth clustering is z_b.

3.7 Summary

This chapter presented the use of light field hyperlines to render scenes with several different degrees of knowledge of the geometry. Camera hyperlines are 2-D entities in 4-D ray space that represent the bundle of rays that pass through the camera center of projection in the object space and intersect with the $s-t$ and $u-v$ planes. Similarly, geometry hyperlines are 2-D entities in 4-D ray space that represent the bundle of rays that pass through one 3-D point in the object space and intersect with the parameterized planes. Camera and geometry hyperlines intersect where the equivalent 3-D point is seen by that camera.

If no geometry is available, the scene is rendered directly from the camera hyperlines. If geometry is available, the scene is treated as a point cloud where each point is represented and rendered as a geometry hyperline. Finally, if there is geometry available, but it is not accurate, each input ray is treated as a hyperline pencil. A hyperline pencil is a set of geometry hyperlines that share the same input ray. The size of this pencil is defined by the uncertainty calculated during the stereo matching step, presented in chapter 2.

Chapter 4

The 3-D Approach: Ellipsoidal Gaussian Kernels

> On the Gaussian Curve: *"Experimentalists think that it is a mathematical theorem while the mathematicians believe it to be an experimental fact."*
>
> Gabriel Lippman, remarked to Poincaré

Our 3-D implementation came late in our work. Our main research topic was the light field hyperline representation, which evolved some time later to rendering uncertainty information. Once we started to work with depth uncertainty, it was a natural step to implement a 3-D renderer to compare it with our 4-D framework.

There are some advantages of using a 3-D implementation. First of all, 3-D rendering has been the main focus of Computer Graphics research since the 60's. Second, the math involved is easier, since we have one dimension less to worry about. Also, we have hardware that can handle most of our graphics algorithms, with the capability of actually running programs in the graphics chip.

As a result, the 3-D implementation turned out to be faster than the 4-D approach (one second versus six seconds). The process of experimenting with different parameters and methods is much more efficient. Hence the fining-tuning of our 3-D implementation went faster, and we were pleased to find out that in many cases it produces better images.

This chapter presents how we implemented depth uncertainty in a 3-D framework.

4.1 Overview of Approach and Chapter Organization

Our 3-D implementation is straightforward. We represent each depth uncertainty (defined in chapter 2) as an ellipsoid, which is the support of our 3-D Gaussian kernel. Next, based on the Volume Splatting work of Zwicker in [38], we project the ellipsoid on the image plane. The projection is an ellipse, and it is rendered as a textured quadrilateral. Finally, the alpha channel is normalized.

The first half of this chapter is about justifying the use of 3-D Gaussian kernels and its approximation to a 2-D elliptical kernel. We have to show that:

1. Gaussians are good filters. First, we have to talk about the sampling problem (section 4.2), where we briefly review some concepts of digital signal processing. Then, with this background, we explain why we chose Gaussians (section 4.3).

2. The orthonormal projection of a 3-D Gaussian is 2-D Gaussian (section 4.3.2).

3. The perspective projection of an ellipsoid is an ellipse (section 4.3.4).

With this in mind, it is not difficult to see that the perspective projection of a 3-D Gaussian can be approximated by a 2-D elliptical Gaussian (section 4.3.5).

The algorithm is explained in detail in section 4.4, including some hardware optimizations.

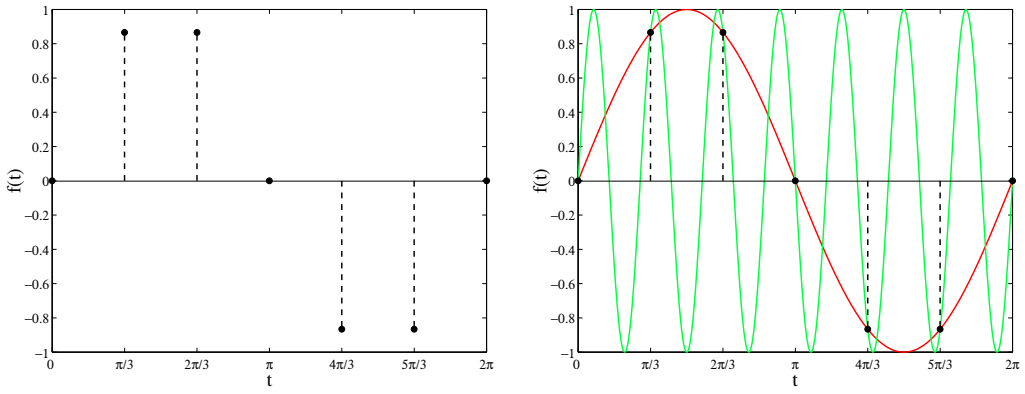
4.2 The Sampling Problem

This section presents a review of some concepts of digital signal processing. We do not intend it to be an extensive review of this subject. This is a rewriting of the work presented in [42], [43], [44] and [45]. Finally, although many examples discussed here are one dimensional, the ideas also apply to 2-D and 3-D.

In Computer Graphics, most of the time we do not work with continuous functions. Images and textures are discrete, and for the purpose of having a good reconstruction it is important to understand the effects of discretizing the data.

Let us take a look at a discrete representation $\hat{g}$ of a function f, Figure 4.1a. We discretized a continuous function in the interval of $t = 0$ to $t = 2\pi$. Assuming that we know the original 1-D function is a sinewave, the first impression we have is that $f(t) = sin(t)$. However, this may not be the case, as Figure 4.1b shows. Those sampled points may also belong to another sinusoid, $f(t) = sin(7t)$.

Now, let us assume that the original function is actually $f(t) = sin(7t)$. The way it was sampled creates ambiguity when we try to reconstruct the original wave. In this case, the sinusoid $f(t) = sin(7t)$ may go by the *alias* of $f(t) = sin(t)$.



(a) Sampled Data $\hat{g}$ (b) $sin(t)$ and $sin(7t)$

Figure 4.1: The Sampling Problem

As a matter of fact, there are an infinite number of sine waves that match these sampled points. So the problem can be defined as which frequencies we may reconstruct and how to reconstruct those frequencies.

4.2.1 The Frequency Domain

The previous example illustrates what we intuitively know: between two samples (or two pixels) there can be an infinite number of intensities, or colors. However, not all colors have the same probability to occur. Think of a picture of the sky. The chances of having a red pixel between two blue pixels exists, but it is very low. It turns out that we can better analyze this problem if we work with the *frequency domain*, instead of the time (or spatial) domain. Using this mathematical formalization, we can create a set of assumptions to help us reconstruct the data the best way we can (this is what filtering is about).

The Fourier Transform

The frequency domain represents data by the rate they occur. The more often a pattern repeats in the data, the higher is the frequency represented. The idea is to represent and/or approximate the data (also called signal) as a sum of sine functions. In order to convert a continuous signal in the spatial domain to the frequency domain we use the Fourier Transform (FT):

$$A(\omega) = \int_{-\infty}^{\infty} f(t) cos\omega t dt - j \int_{-\infty}^{\infty} f(t) sin\omega t dt \tag{4.1}$$

where $j = \sqrt{-1}$ and ω is the angular frequency. $A(\omega)$ is the amplitude and phase of a complex exponential at frequency ω. It is called the *signal spectrum*.

If our data is discrete, we use the Discrete Fourier Transform (DFT) to calculate its frequency domain representation:

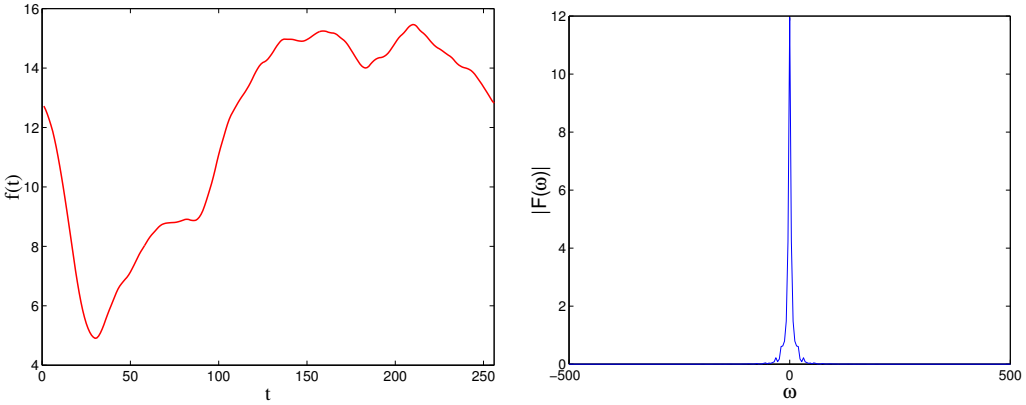
$$A(k) = \sum_{n=0}^{N-1} f(n) \left[cos\left(\frac{k2\pi n}{N}\right) + jsin\left(\frac{k2\pi n}{N}\right) \right] \tag{4.2}$$

where N is the number of samples. It is assumed that the sampled points are equally spaced.

Although a cosine wave can be represented by a sine wave, the use of both simplifies the coefficients needed to represent the signal in the frequency domain. It is convenient to use complex numbers to represent the frequency of a signal. The real part is used to represents the cosine term and the imaginary part the sine term. The sum of two complex numbers is the vector addition of two vectors, and a simple multiplication with a fixed complex number produces rotation and stretching in the complex plane.

Visualizing the Frequency Domain

For the 1-D examples presented in this section we plot the absolute value of the FT or DFT. Figure 4.2 shows the plot of a function in the spatial and frequency domains. Notice that the dominant frequencies are shown as peaks. Since we are plotting the absolute value of the complex frequencies, the frequency domain graph is always symmetric.

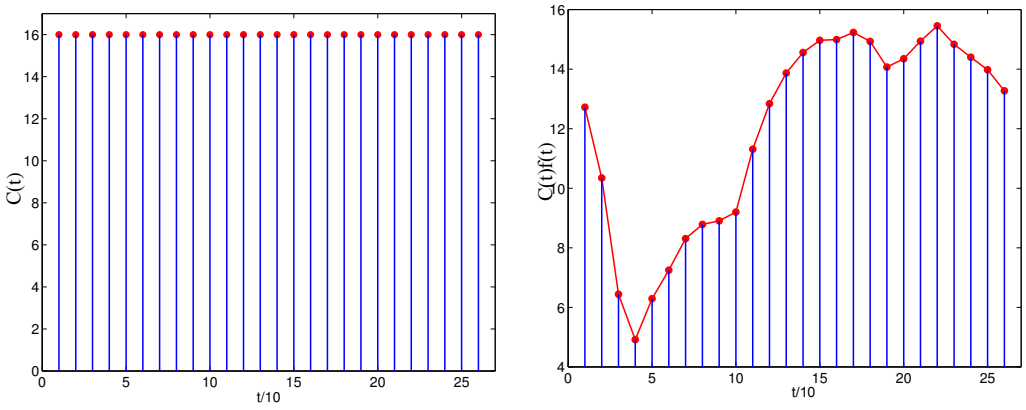


(a) $f(t)$ (b) $|F(\omega)|$

Figure 4.2: Spatial and Frequency domain of function f

4.2.2 The Meaning of Sampling

Now we can understand the meaning of sampling a function by analyzing its frequency domain behavior. Sampling a function f is equivalent of multiplying an input signal to a *comb* function (Figure 4.3a). Figure 4.3b shows the result of multiplying the function $f(t)$ (from Figure 4.2a) to a comb function $C(t)$.

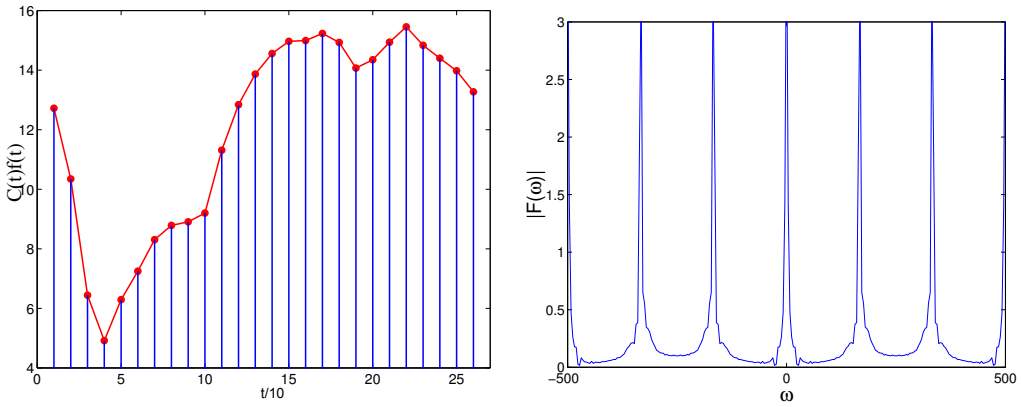


(a) Comb Function $C(t)$ (b) $C(t) \cdot f(t)$

Figure 4.3: Multiplying f by a Comb Function C

The spectrum of $C(t) \cdot f(t)$ is shown in Figure 4.4b. The center of the spectrum looks like the spectrum of $f(t)$ (Figure 4.2b), but now we have infinite copies to both right and left directions. What does that mean?

While in the spatial domain we have performed a multiplication, in the frequency domain what happened is a *convolution*. A convolution is a special interaction between two



(a) $C(t) \cdot f(t)$ (b) FT of $C(t) \cdot f(t)$

Figure 4.4: Sampling causes aliasing in the frequency domain

signals to produce a new one:

$$f(x) * g(x) = \int_{-\infty}^{\infty} f(x-t)g(t)dt \tag{4.3}$$

Convolution can be seen as applying a stationary function (sometimes called filter) to a signal. An interesting analogy to understand convolution is to think of it as the effect of a prism (the stationary filter) on a ray of sunlight (the signal).

The Fourier Transform of the multiplication of two functions in the spatial domain is equal to the convolution of the Fourier Transforms of those functions. The Fourier Transform of a comb function is another (scaled) comb function. If the points in the original comb function are spaced n units apart, the points of its Fourier Transform are spaced $2\pi/n$ units apart.

Therefore, sampling is equivalent to convolving the FT of a comb function and the FT of the original function, which replicates the frequency of the original function throughout the spectrum. Here is where the term *alias* come from. The more the replicas intersect, the more aliased is the resulting curve. The graph in Figure 4.4b shows little to no aliasing, while the Fourier Transform of $f(t)$ (Figure 4.1) has much more aliasing (Figure 4.5). If the copies do not intersect we can reconstruct the original signal exactly.

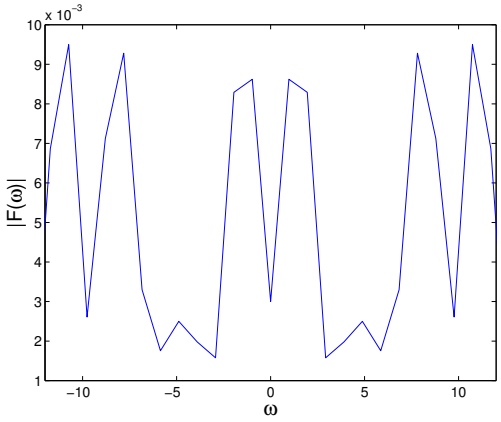


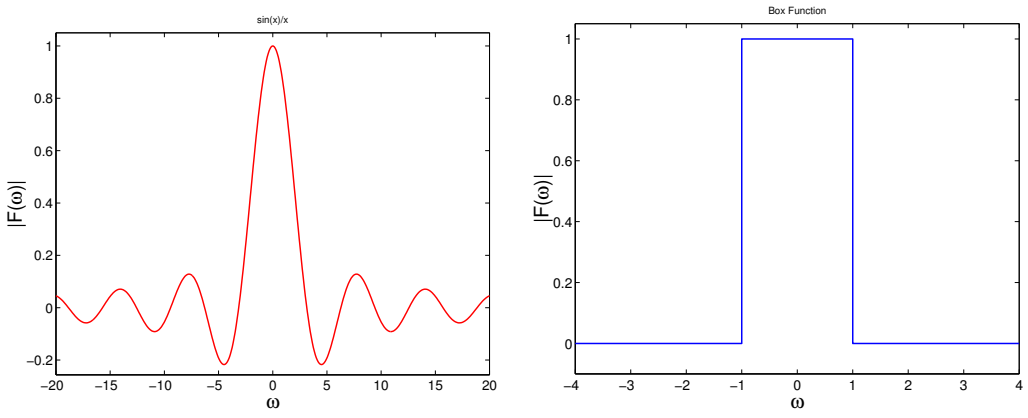
Figure 4.5: Example of Aliasing - the Fourier Transform of $\hat{g}(t)$ (Figure 4.1a)

.

Reconstruction and Filtering

In order to guarantee that the copies will not overlap, the sampling rate must be at least twice the highest frequency of the signal. This is known as the *Nyquist* frequency. In this way, we can discard the wrong frequencies through filtering.

Filtering a signal means to *bandlimit* its frequency. The idea is to discard all the frequencies that we believe do not belong to the original signal or they cannot be reconstructed. The filters that are most used are the *low-pass filters* - they cut off the copies of higher frequencies.



(a) (b)

Figure 4.6: The Ideal Low-Pass Filter

The sinc filter (Figure 4.6) is an ideal low-pass filter. This is because the Fourier

Transform of a sinc function is a *box* function, which can "cut" the function frequencies at the exact frequency we want.

Notice that although the frequency domain is useful for better understanding and modeling of our systems, almost all the time we perform our computations in the spatial domain. Calculating the Fourier Transform and calculating its inverse is not efficient. Thus, the sinc filter is not practical, since it is infinite in the spatial domain.

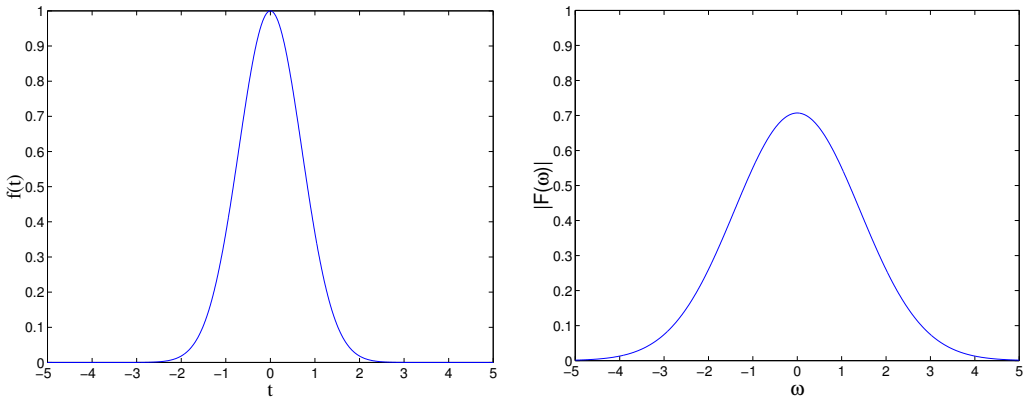
There are many low-pass filters in the literature. Some work by placing a window over an otherwise infinite function, such as the sinc function. Others use higher degree polynomials such as B-spline filters. The filter of interest in this chapter is the Gaussian filter.

4.3 Ellipsoidal Gaussian Kernels

4.3.1 Gaussian as Low-Pass Filters

Now that we have reviewed sampling theory and the frequency domain, we can start building our case for Gaussian filters. There are many advantages of using a Gaussian kernel as our filter:

- Gaussians can be easily truncated. Although they are infinite in the spatial and frequency domain, we can truncate it with a window with almost no loss of quality, since they approach zero rapidly.
- Gaussians are very efficient and easy to implement; a small kernel is usually enough for our reconstruction.
- The convolution of a Gaussian with another Gaussian is also a Gaussian.
- The Fourier Transform of a Gaussian is also a Gaussian (Figure 4.7).
- The orthogonal projection of a 3-D Gaussian is a 2-D Gaussian (see section 4.3.2).
- Gaussian is the natural form of many natural and manmade systems [42].

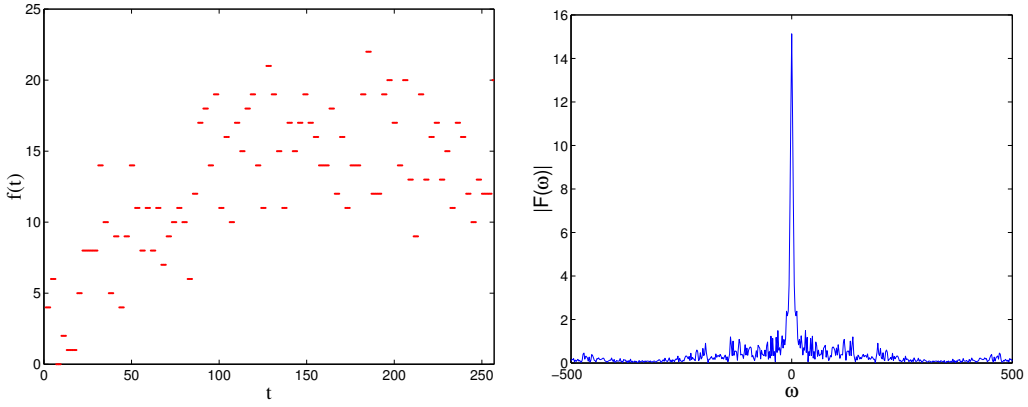


(a) Spatial Domain (b) Frequency Domain

Figure 4.7: Example of a Gaussian Kernel

Of course, there are some disadvantages of using a low-pass filter such as the Gaussian filter. In practice, when rendering from images, it is impossible to guarantee the Nyquist rate. Therefore we may lose some important high frequency information when reconstructing the image, producing a blurred image. We minimize this effect by using smaller kernels when high frequency is detected (i.e., low uncertainty).

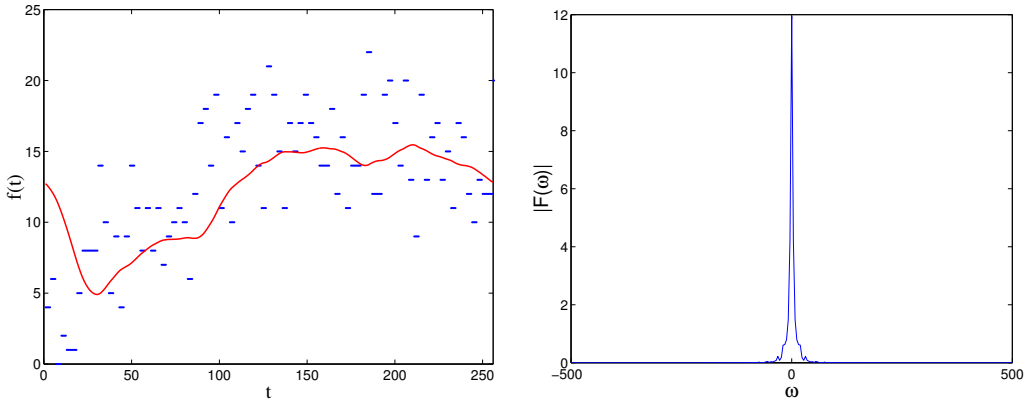
The following images show an example of reconstructing an input signal using a Gaussian filter. Figure 4.9a are the intensities of an horizontal line from a grayscale image acquired from a photograph. Figure 4.9b is the Fourier Transform of this data, which is really irregular, indicating a lot of aliasing.



(a) Spatial Domain (b) Frequency Domain

Figure 4.8: Sampled Data

Figure 4.9a shows the resulting curve of applying the gaussian filter showed in Figure 4.7. Notice that the output signal is smooth, as it is its Fourier Transform (Figure 4.9b). This is the signal we used in section 4.2.1.



(a) (b)

Figure 4.9: Filtering

Applying a Gaussian filter may not reconstruct the original function, but the new function will display the most important aspects of the original function. Figure 4.10 shows the effect of applying a Gaussian kernel to an image.

4.3.2 Orthogonal Projection of a 3-D Gaussian

Here we show that in an orthogonal projection, the integral of a 3-D Gaussian is a 2-D Gaussian. Let us analyze a simple spherical Gaussian in the form of:

$$G_v^{3D}(x,y,z) = \frac{1}{(s\sqrt{\pi/2})^3} e^{-(x^2+y^2+z^2)/(2s^2)}$$

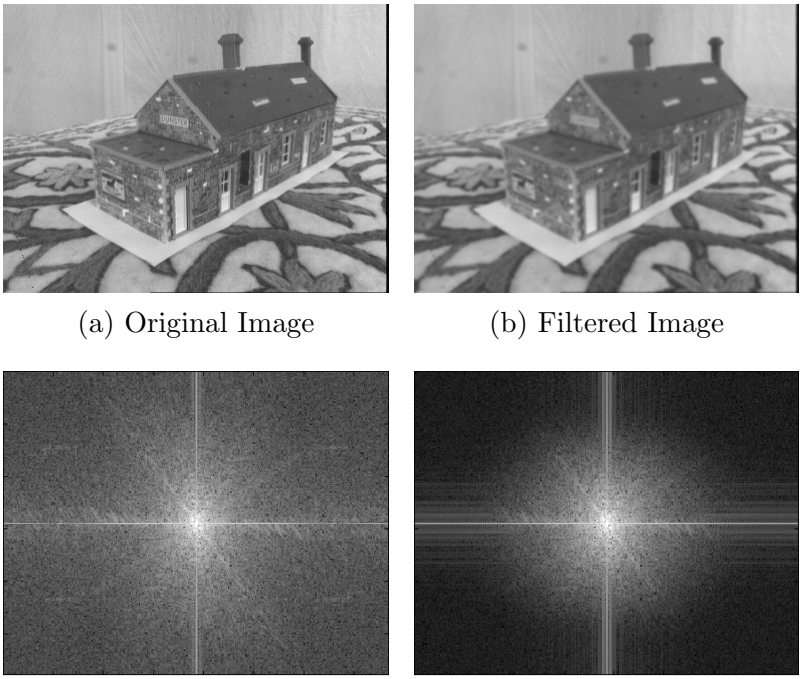
which can also be represented as

$$G_v^{3D}(x,y,z) = \frac{1}{(s\sqrt{\pi/2})^2} e^{-(x^2+y^2)/(2s^2)} \frac{1}{s\sqrt{\pi/2}} e^{-(z^2)/(2s^2)} \tag{4.4}$$

or

$$G_v^{3D}(x,y,z) = G_v^{2D}(x,y) G_v^{1D}(z) \tag{4.5}$$

The meaning of equation 4.5 is that equation 4.4 can also be interpreted as the multiplication of a 2-D Gaussian and a 1-D Gaussian.



(a) Original Image (b) Filtered Image

(c) 2D FT of (a) (d) 2D FT of (b)

Figure 4.10: Applying the Gaussian Kernel to an Image. The frequencies in (c) and (d) are shown as the gray scale intensities, white meaning the highest frequency. The frequencies are plot in log scale.

Now, given that

$$\int_{-\infty}^{\infty} e^{-(z^2)/(2s^2)} dz = s\sqrt{\pi/2}$$

the integration of equation 4.4 along the z axis is exactly a 2D Gaussian:

$$\int_{-\infty}^{\infty} \frac{1}{(s\sqrt{\pi/2})^2} e^{-(x^2+y^2)/(2s^2)} \frac{1}{s\sqrt{\pi/2}} e^{-(z^2)/(2s^2)} dz =$$

$$= \frac{1}{(s\sqrt{\pi/2})^2} e^{-(x^2+y^2)/(2s^2)}$$

$$= G_v^{2D}$$

4.3.3 Implicit Ellipsoid

The ellipsoid is the support of our 3-D Gaussian. We are interested in the projection of this ellipsoid on the image plane. First, let us start with the implicit equation of a generic quadric surface, which is defined as

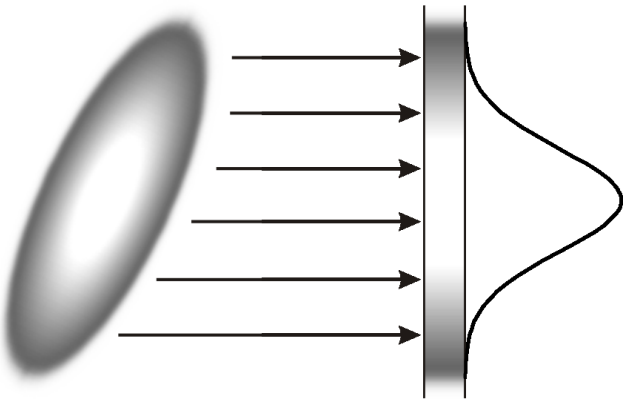


Figure 4.11: The orthogonal projection of an n-D Gaussian is an $(n-1)$-D Gaussian

$$Ax^2 + By^2 + Cz^2 + Dxy + Exz + Fyz + Gx + Hy + Iz + J = 0 \tag{4.6}$$

this equation can be expressed in the matrix form

$$\begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} A & D/2 & E/2 \\ D/2 & B & F/2 \\ E/2 & F/2 & C \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} G & H & I \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + J = 0$$

$$\vec{x}^T Q \vec{x} + \vec{b} \cdot \vec{x} + J = 0 \tag{4.7}$$

If Q is a positive definite matrix, equation 4.7 is an ellipsoid.

4.3.4 Perspective Projection of an Ellipsoid

Here we show (1) how to calculate the perspective projection of an ellipsoid and that (2) this projection is always an ellipse. This section is based on the derivation in [39] and [41]. We assume that the ellipsoid is always in front of the image plane.

Deriving the Projection of the Ellipsoid

Let the camera's center of projection be located at the origin. Let $\vec{r}$ be a unit vector, and let $R(\vec{r})$ be the ray through the origin in direction $\vec{r}$. It has the equation

$$\vec{x}(t) = t\vec{r}$$

with $t > 0$. The intersection of all $R(\vec{r})$ with the quadric can be calculated by replacing $\vec{x}$ in 4.7 by $t\vec{r}$

$$\begin{aligned} 0 = &\quad \vec{x}(t)^T Q\vec{x}(t) + \vec{b}\cdot\vec{x}(t) + J \\ = &\quad (t\vec{r})^T Q(t\vec{r}) + \vec{b}\cdot(t\vec{r}) + J \\ = &\quad (\vec{r}^T Q\vec{r})t^2 + (\vec{b}\cdot\vec{r})t + J \end{aligned}$$

$$0 = \quad \alpha t^2 + \beta t + J \tag{4.8}$$

Each ray may (1) intersect the ellipsoid in one place, (2) intersect the ellipsoid in two places, or (3) do not intersect the ellipsoid. If no roots are found the ray does not intersect the quadric. If there are two different roots, it intersects the quadric in two places. We are interested in the rays that are tangent to the quadric, or those that intersect the ellipsoid in exact one place. In this case, there must be a single solution for 4.8:

$$\begin{aligned} 0 = &\quad \beta^2 - 4\alpha J \\ = &\quad (\vec{b}\cdot\vec{r})^2 - 4(\vec{r}^T Q\vec{r})J \\ = &\quad \vec{r}^T\left[\vec{b}\cdot\vec{b}^T - 4QJ\right]\vec{r} \end{aligned}$$

and this is represented as

$$0 = \vec{r}^T M\vec{r} \tag{4.9}$$

Let us now analyze the intersection of all $R(\vec{r})$ with the image plane. The image plane is defined as

$$\vec{n}\cdot\vec{x}_d = d \tag{4.10}$$

where $\vec{x}_d$ are points on the plane and $\vec{n}$ is the normal of the plane. The intersection of $R(\vec{r})$ with this plane is given by $\vec{x}(t_d)$

$$\begin{aligned} d &= \vec{n} \cdot \vec{x}(t_d) \\ &= \vec{n} \cdot (t_d \vec{r}) \\ &= t_d \, (\vec{n} \cdot \vec{r}) \end{aligned}$$

and the solution is

$$t_d = \frac{d}{\vec{n} \cdot \vec{r}}$$

Since the point of intersection of the plane with any ray is $\vec{x}_d = t_d \vec{r}$, rays that intersect the plane can be defined as:

$$\vec{r}_d = \frac{1}{t_d} \vec{x}$$

Next, replacing all $\vec{r}$ in equation 4.9 by $\vec{r}_d$ yields

$$0 = \frac{1}{t_d^2} \vec{x}^T M \vec{x}$$

Notice that $t_d \neq 0$. Thus, the projection of the ellipsoid on the plane is simply given by

$$0 = \vec{x}^T M \vec{x} \tag{4.11}$$

Projected Shape is an Ellipse

Now we have to show that equation 4.11 is, in fact, an ellipse. Without loss of generality let the image plane be parallel to the XY plane. Generic configurations may be obtained by translating and rotating the ellipsoid. The normal of the image plane is:

$$\vec{n} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T$$

The image plane may be located at any z, but for simplicity let us assume that $z = 1$ (Figure 4.12) . Recall that

$$\vec{x} = \begin{bmatrix} x & y & z \end{bmatrix}^T$$

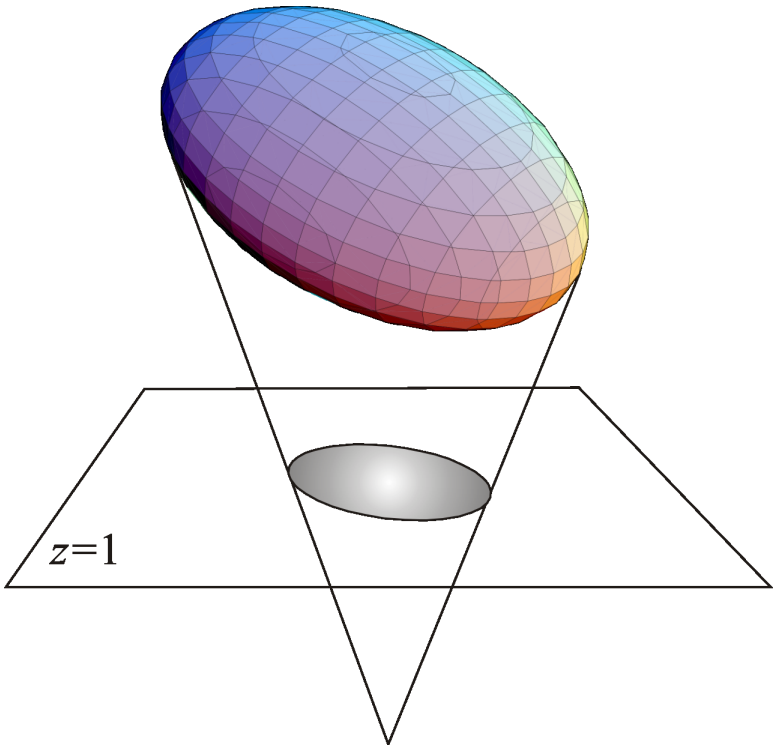


Figure 4.12: The Perspective Projection of an Ellipsoid

Equation 4.11 can be rewritten as:

$$0 = \begin{bmatrix} x & y & 1 \end{bmatrix} M \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The expanded form of this equation is

$$A'x^2 + B'y^2 + C' + D'xy + E'x + F'y + G'x + H'y + I' + J' = 0$$

and rearranging the terms yields

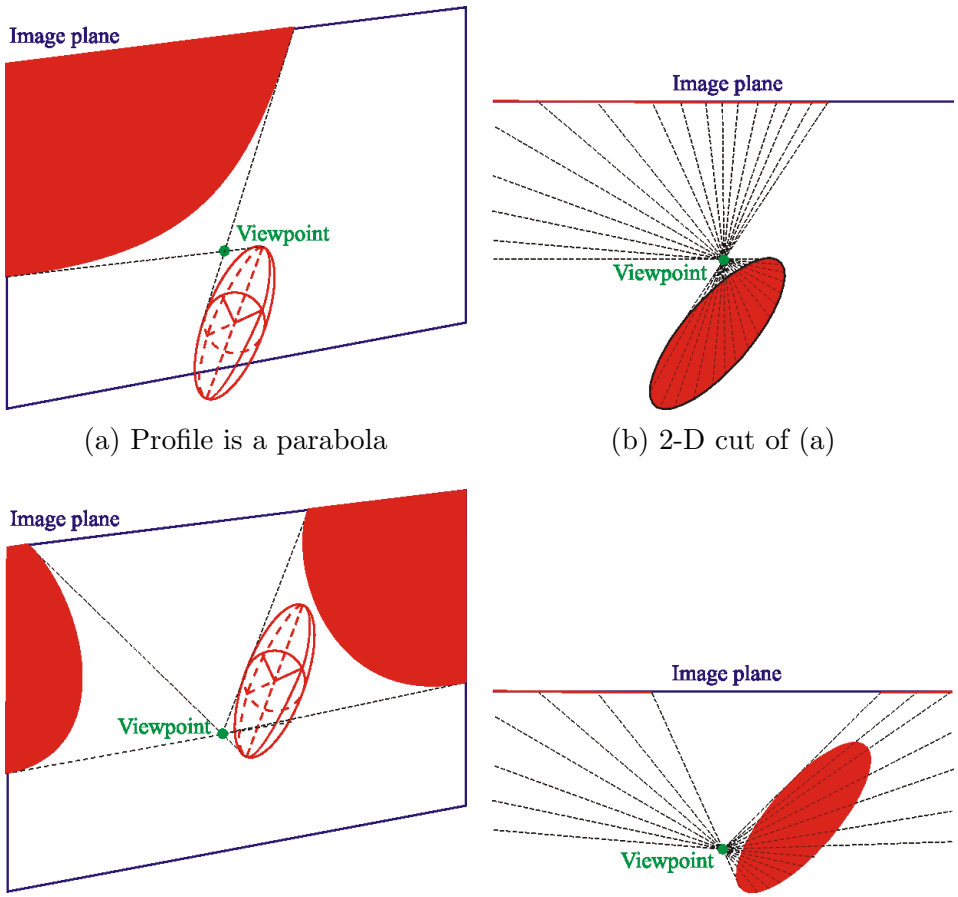
$$A'x^2 + B'y^2 + C' + D'xy + (E' + G')x + (F' + H')y + (J' + I' + C') = 0$$

Finally, let $K' = (E' + G')$, $L' = F' + H'$ and $P' = J' + I' + C'$, which yields

$$A'x^2 + D'xy + B'y^2 + K'x + L'y + P' = 0 \tag{4.12}$$

This is the equation of a conic on the plane $z = 1$. Figure 4.12 shows the elliptical cone formed by all rays from the viewpoint tangent to the ellipsoid, and the curve in which this

cone intersects the image plane. Equation 4.12 shows that this curve is a conic section. If the ellipsoid is entirely on one side of the plane through the viewpoint, the elliptical cone will intersect the image plane in a single, connected curve, which lies in a bounded region of the plane, so it must be an ellipse (or a circle). Figure 4.13 illustrates that if our assumption does not hold, i.e., if, for instance, the ellipsoid extends partly behind the viewer, the projected profile may be a hyperbola or a parabola.



(a) Profile is a parabola (b) 2-D cut of (a)

(c) Profile is a hyperbola (d) 2-D cut of (c)

Figure 4.13: Ellipsoid perspective projection may be a hyperbola or a parabola.

4.3.5 The Perspective Approximation of the 3-D Gaussian

Figure 4.14 depicts our claim. It is possible to approximate a 3-D ellipsoidal Gaussian using a 2-D Gaussian kernel in its elliptical perspective projection.

The idea can be illustrated as follows. First, we perform an orthogonal projection of the ellipsoidal to a plane through the center of the ellipsoid, parallel to the image plane.

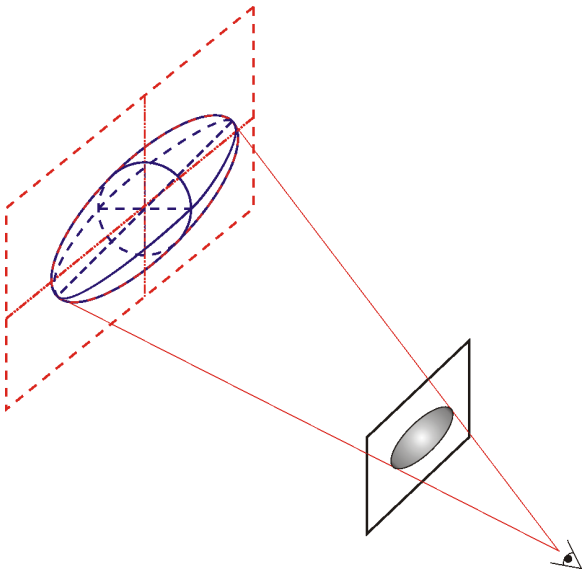


Figure 4.14: Perspective approximation idea

Next, we project the resulting ellipse into the image plane, using a 2-D elliptical Gaussian kernel for blending. If the ellipsoid is relatively far away from the image plane, the resulting approximation is very close to the exact solution. In cases where the ellipsoid is too close to the camera, the difference may be noticed. However, this is not a likely scenario. Most of the time the ellipsoids will be far away from the center of projection.

In practice, the projection to an adjacent plane is not done. Instead, the ellipsoid is projected directly on the image plane.

4.4 Rendering Depth with Uncertainty using 3-D Ellipsoids

Rendering depth with uncertainty using ellipsoidal gaussian kernels is straightforward. The technique can be summarized as follows:

1. Transform all depth uncertainty regions to ellipsoids
2. Build a global sorted list for *every* axis (X, Y and Z)
3. For each new virtual view
   (a) Calculate the elliptical projection of all ellipsoids to the image plane
   (b) For each axis,

i. Calculate the corners of the ellipse

ii. Calculate the weight of each corner

iii. Clip the quadrilaterals, if necessary

iv. Calculate the final color

(c) Normalize the final color

(d) Display Image

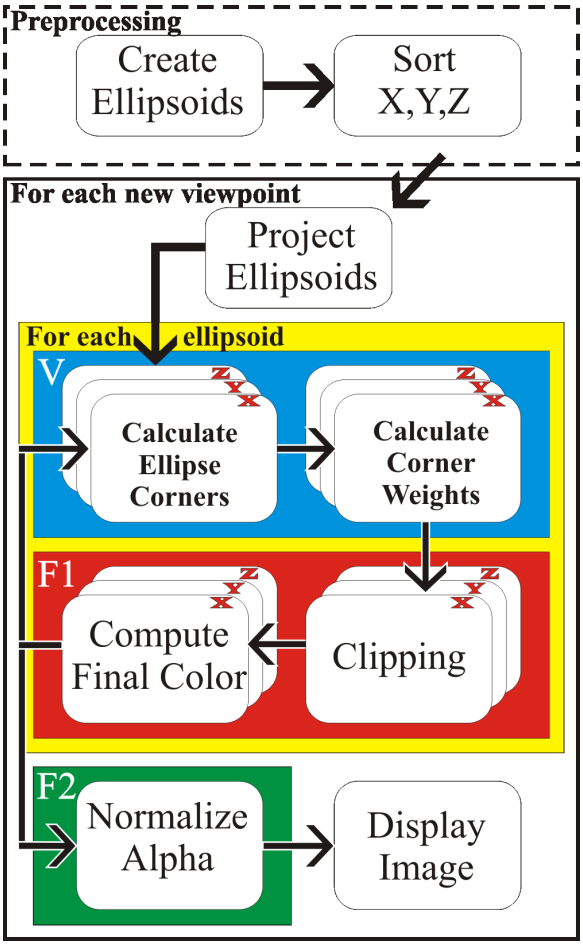


Figure 4.15: The 3-D Algorithm. Modules in F1 and F2 are executed by fragment programs. Modules in V are executed by a vertex program.

Figure 4.15 illustrates our rendering. After the ellipsoid is projected, the rendering is finalized by the OpenGL engine, with the help of some vertex and fragment programs.

Steps 3(b)i and 3(b)ii are implemented as a vertex program. Steps 3(b)iii and 3(b)iv are implemented as a fragment program. The final step of normalizing the alpha of the im-

age (step 3c) is also implemented as a fragment program. The vertex and fragment programs are defined under the OpenGL ARB extensions specifications, version 1.0.

The following sections describe each step of the algorithm in detail.

4.4.1 The Ellipsoid

The ellipsoid is the support of our 3-D Gaussian kernel, and it is defined by a center $\vec{k}$, three orthonormal unit-length vectors $\vec{u}$, $\vec{v}$, $\vec{w}$, and three scalars a, b and c, that are the lengths of the half-axes.

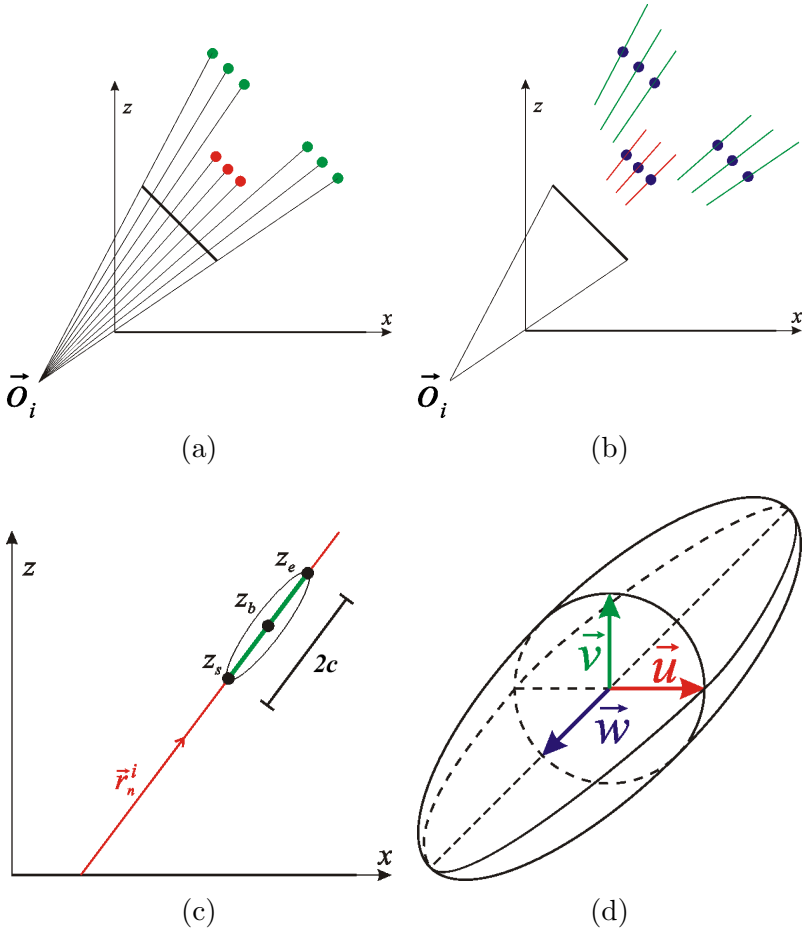


Figure 4.16: Each ray in an input image (a) has a depth uncertainty region (b) assigned to it. This region is modeled as an ellipsoid [(c) and (d)] in our 3D framework.

As seen in section 2.4, each input ray $R(\vec{r}_n^{\,i})$ of all input images i has a depth uncertainty region. This region is defined by z_s (starting depth), z_e (ending depth) and z_b (best estimated depth). Recall that depth in this document means the distance along the

viewing ray. The depth uncertainty region is modeled as an ellipsoid as follows:

- The center $\vec{k}$ of the ellipsoid is given by z_b:

$$\vec{k} = \vec{o}_i + z_b \vec{r}_n^{\,i}$$

  where $\vec{o}_i$ is the center of projection of image i.

- The direction of the major axis $\vec{w}$ is given by $\vec{r}_n^{\,i}$ and its length is (Figure 4.16c):

$$c = \frac{z_e - z_s}{2}$$

- The two minor axes $\vec{u}$ and $\vec{v}$ (Figure 4.16d) are orthogonal to $\vec{w}$. Their directions are computed as follows:

$$\vec{u} = \frac{\vec{w} \times \vec{u''}_i}{|\vec{w} \times \vec{u''}_i|}$$

$$\vec{v} = \frac{\vec{v''}_i \times \vec{w}}{|\vec{v''}_i \times \vec{w}|}$$

  and their lengths are

$$a = |\vec{u'}_i|$$

$$b = |\vec{v'}_i|$$

  where $\vec{u'}_i$ and $\vec{v'}_i$ are the image plane basis vectors (Figure 4.17), $\vec{u''}_i = \vec{u'}_i/|\vec{u'}_i|$ and $\vec{v''}_i = \vec{v'}_i/|\vec{v'}_i|$.

The Ellipsoid Coefficients

Now it is possible to calculate the ellipsoid coefficients. The ellipsoid equation in 4.7 is repeated here

$$\vec{x}^T Q \vec{x} + \vec{b} \cdot \vec{x} + J = 0$$

Any point $\vec{x}$ in the ellipsoid is given by

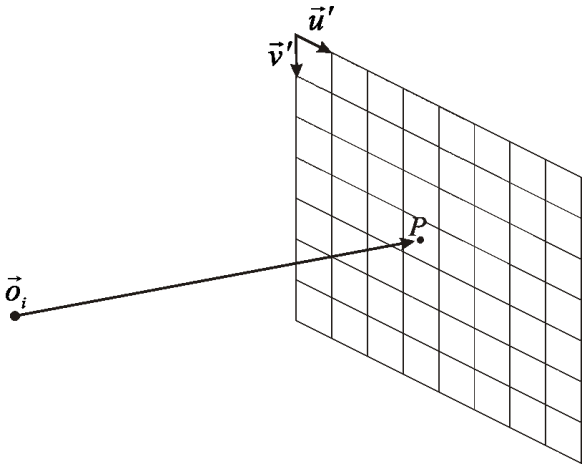


Figure 4.17: Virtual Camera Definition

$$(\vec{x}-\vec{k})^T\left(\frac{\vec{u}\vec{u}^T}{a^2}+\frac{\vec{v}\vec{v}^T}{b^2}+\frac{\vec{w}\vec{w}^T}{c^2}\right)(\vec{x}-\vec{k})=1$$

and, therefore,

$$Q=\frac{\vec{u}\vec{u}^T}{a^2}+\frac{\vec{v}\vec{v}^T}{b^2}+\frac{\vec{w}\vec{w}^T}{c^2}$$

$$\vec{b}=-2A\vec{k}$$

$$J=\vec{k}^TA\vec{k}-1$$

4.4.2 Sorting

We use here an idea similar to the one presented in sections 3.5.5 and 3.6.2. We maintain three lists of all ellipsoids sorted by the X, Y or Z components of their centers $\vec{k}$. Each pixel in the image will receive contribution for only one of those lists.

The list to be chosen depends on the X, Y and Z components of the virtual ray of that pixel. The choice is defined by the component with largest absolute value. The direction of searching the list is defined by the sign of this component, front-to-back if it is negative and back-to-front if it is positive.

A list is only searched if there is at least one virtual ray whose largest component X, Y or Z is the one that identifies the list.

4.4.3 Projection

For the purpose of rendering the image, we first project all ellipsoids to the image plane (step 3a of the algorithm, see section 4.4), using the derivation presented in section 4.3.4. During this step, we identify which lists we have to search (see section 4.4.2) to project each ellipsoid. A few ellipsoids may have to be projected two or three times.

Of course, each pixel cannot receive the contribution of an ellipsoid more than once. The ellipsoid must be clipped accordingly. This is done in hardware, and it is explained in section 4.4.5.

The output of this stage is the representation of the ellipse in the 3-D plane. Specifically, we need to calculate the 2-D ellipse center, the 2-D axes and the axes half-lengths. The following paragraphs are a reproduction of the result of the derivation in [39].

Using the matrix M from equation 4.11, and the plane equation from 4.10, we define six constants k as:

$$\begin{aligned} k_0 &= \vec{u'}_k^T M \vec{u'}_k \\ k_1 &= \vec{u'}_k^T M \vec{v'}_k \\ k_2 &= \vec{v'}_k^T M \vec{v'}_k \\ k_3 &= 2d\vec{u'}_k^T M \vec{n} \\ k_4 &= 2d\vec{v'}_k^T M \vec{n} \\ k_5 &= d^2 \vec{n}^T M \vec{n} \end{aligned}$$

Now, let P be the matrix:

$$P = \begin{bmatrix} k_0 & k_1 \\ k_1 & k_2 \end{bmatrix}$$

Next, consider that the matrix P has a factorization such as that $P = R^T F R$, where R is a rotation matrix (eigenvectors) and F is a diagonal matrix (eigenvalues), and

$$F = \begin{bmatrix} f_1 & 0 \\ 0 & f_2 \end{bmatrix}$$

Also, consider that

$$\beta = \begin{bmatrix} \beta_1 & \beta_2 \end{bmatrix}^T = R \begin{bmatrix} k_3 & k_4 \end{bmatrix}^T$$

Then, the 2-D axes are the eigenvectors of P. The center of the ellipse is given by:

$$e_x = \frac{-\beta_1}{2f_1}$$

$$e_y = \frac{-\beta_2}{2f_2}$$

And, finally, the axes half-lengths are:

$$h_0 = \sqrt{f_2 \phi}$$

$$h_1 = \sqrt{f_1 \phi}$$

where

$$\phi = \frac{\frac{\beta_1^2}{4f_1^2} + \frac{\beta_2^2}{4f_2^2} - k_5}{f_1 f_2}$$

The source code for projecting the ellipsoid is public available in [40].

4.4.4 Corners

The work on the corners is implemented as a vertex program. The input for this stage is:

- The center of the ellipse (e_x, e_y)
- The half-axes length h_0 and h_1
- The ellipse 2-D axes $axis^0$ and $axis^1$
- The image plane 3-D vectors (normalized) $\vec{u''}_i$ and $\vec{v''}_i$ (recall that $\vec{u''}_i = \vec{u'}_i/|\vec{u'}_i|$ and $\vec{v''}_i = \vec{v'}_i/|\vec{v'}_i|$)
- The 3-D center of the image P
- The center of projection $\vec{o}_i$

- The color of the splat and its uncertainty value
- The corner identification EY and EX values. For instance, the upper left corner of the ellipse is $EX = -1$ and $EY = -1$
- The main axis of the ellipsoid k, $\vec{w}_k$

The output is threefold:

a) The Vertex final position

Which is calculated as:

$$\begin{aligned} E = & \quad P + e_x \vec{u''}_i + e_y \vec{v''}_i \\ Vertex = & \quad E + [EXaxis_x^0 + EYaxis_x^1]\vec{u''}_i + [EXaxis_y^0 + EYaxis_y^1]\vec{v''}_i \end{aligned}$$

where E is the 3-D center of the ellipse.

b) The Weight of the Vertex

Each projected ellipse k will have an additional viewing direction weight VW_i^k for each pixel i it projects. The closer in angle the virtual ray $\vec{r_i}$ is to the input ray that defined the ellipsoid k ($\vec{w_k}$), the higher it will be. This weight is calculated as follows:

$$VW_i^k = (1 - u_k) \left| \left(\vec{r_n^i} \cdot \vec{w_k} \right) \right|^2 \tag{4.13}$$

where VW_i^k is the weight of ellipsoid k for pixel i, $\vec{r_n^i}$ is the virtual ray i and $\vec{w_k}$ is the vector that defines the major axis of ellipsoid k. We noticed that for higher exponents (3 or larger) many splats were not considered at all. One minus the uncertainty times this formula results in a small number. In our hardware implementation, we remove small contributions from the pipeline to avoid numerical errors. Therefore the exponent can be 1 or 2, but 2 produces much better results.

The uncertainty is defined by u_k, and it is calculated as follows[1]:

[1] This is basically the same equation as in 3.20.

$$u_k = \frac{z_e^k - z_s^k}{\max(z_e^j - z_s^j)}, \text{ for any ellipsoid } j \tag{4.14}$$

and $\vec{r_n^i}$ is given by:

$$\vec{r_n^i} = \frac{Vertex - \vec{o}_i}{|Vertex - \vec{o}_i|}$$

c) The Absolute Values of the Viewing Ray

That is, $|\vec{r_n^i}|$. This is going to be used by the fragment shader to check if the ellipse must be clipped at this pixel or not, according to the current sorted list being searched.

The code for the vertex program is shown in Figure 4.18.

4.4.5 The Final Color

The final color is computed as the gaussian texture multiplied by the color of the ellipsoid and the weight VW_i^k. The alpha of a fragment is always set to be in the range of 0.1 and 0.8.

There are actually three fragment programs in this stage, one for each sorted list. Before searching the lists, the specific fragment program for each list (X, Y or Z) is activated. The fragment is killed (i.e., it is not used for the final color) if (1) the viewing ray component with the largest value is not the one that the fragment was designed (for instance, the Z component of the viewing ray is the largest, but the current fragment program is specific for the X list) or (2) the final alpha value is less than 0.05. Therefore, (1) performs clipping and (2) rejects pixels that may cause numerical errors.

The fragment program for the list sorted according to the X component is shown in Figure 4.19.

4.4.6 Normalization

Finally, the image must be normalized. If the final accumulated alpha is not equal to one, we have to divide the color by its alpha. Since fragment programs cannot read pixels from the image, the image is read as a texture.

The fragment program that normalize the alpha channel is shown in Figure 4.20.

4.5 Summary

This chapter presented the 3-D approach of rendering uncertainty depth regions. First, we reviewed the sampling problem and the frequency domain. Filtering means to cut the frequencies that either do not belong to our original function or that we cannot reconstruct. A Gaussian filter has many advantages. It can be truncated almost without loss of quality, since it approaches zero rapidly. Its kernel is small and its implementation is efficient. A 3-D ellipsoidal Gaussian kernel can be approximated by a 2-D ellipse, since the orthogonal projections of a 3-D Gaussian is a 2-D Gaussian and the perspective projection of an ellipsoid is an ellipse.

Rendering is straightforward. The uncertainty depth region is represented as an ellipsoid, which is approximated by a 2-D textured quadrilateral. The 3-D implementation is a three-pass algorithm, one for each axis, to deal with occlusion. During each pass a sorted list is searched. The largest component of the virtual ray defines which sorted list (X, Y or Z) to use.

The projection of the ellipsoid is still done in software. The corners of the quadrilateral containing the ellipse, the weighting, the clipping and the alpha channel normalization are all implemented using fragment and vertex programs.

```
!!ARBvp1.0
PARAM  U              = program.env[1];    # U image plane vector
PARAM  V              = program.env[2];    # Y image plane vector
PARAM  P              = program.env[4];    # Center of image
PARAM  COP            = program.env[5];    # Center of Projection
PARAM  mvp[4]         = { state.matrix.mvp };
# Vertex position actually contains encoded position (EX, EY)
# of which corner of the ellipse to calculate for the vertex
# Corners can be: pos.x = -1, pos.y = -1; pos.x = -1, pos.y = 1;
#                 pos.x =  1, pos.y = -1; pos.x =  1, pos.y = 1;
ATTRIB pos            = vertex.position;
ATTRIB color2         = vertex.color.secondary;
ATTRIB intexcoord     = vertex.texcoord;   # Texture
ATTRIB ecenter        = vertex.attrib[10]; # ellipse 2D center
ATTRIB axis0          = vertex.attrib[11]; # ellipse 2D axis 0
ATTRIB axis1          = vertex.attrib[12]; # ellipse 2D axis 1
ATTRIB W              = vertex.attrib[13]; # Ellipsoid main axis vector
# Output Variables
OUTPUT  opos          = result.position;
OUTPUT  outcolor      = result.color;
OUTPUT  outcolor2     = result.color.secondary;
OUTPUT  outtexcoord = result.texcoord;
TEMP    tmp1,tmp2,tmp3,npos;
# Calculate the center of the ellipse npos=P+ecenter.x*U+ecenter.y*V
MAD tmp1,ecenter.x,U,P;
MAD npos,ecenter.y,V,tmp1;
# vertex = npos + [EX*axis0.x + EY*axis1.x]*U +
#                 [EX*axis0.y + EY*axis1.y]*V;
# axis0 is passed as h0*axis0 and axis1 as h1*axis1
# EX and EY are pos.x and pos.y
MUL tmp1,pos.x,axis0;
MUL tmp2,pos.y,axis1;
ADD tmp1,tmp1,tmp2;
MUL tmp2,tmp1.x,U;
MUL tmp1,tmp1.y,V;
ADD npos,npos,tmp1;
# npos receives the final vertex position
ADD npos,npos,tmp2;
# Calculate the viewing ray
SUB tmp1,npos,COP;
# Calculate the weight (tmp2) of this vertex: weight = Dot(ViewingRay,W)^2
DP3 tmp2.w,tmp1, tmp1;          # tmp2.w = x^2+y^2+z^2;
RSQ tmp2.w,tmp2.w;              # tmp2.w = 1/sqrt(x^2+y^2+z^2);
MUL tmp2.xyz, tmp2.w, tmp1;     # tmp2   = tmp1/tmp2.w;
DP3 tmp3,tmp2,W;                # tmp3 = Dot(ViewingRay,W);
MUL tmp3,tmp3,tmp3;             # tmp3 = tmp3^2;
# Weight is stored as the alpha component (w) of tmp2
MOV tmp2.w,tmp3.w;
# tmp2.xyz itself will contain the absolute value of the virtual ray
# components. It will be used by the fragment program for clipping
ABS tmp2,tmp2;
MOV outcolor,  tmp2;
# Project calculated vertex to the screen
DP4   opos.x, mvp[0], npos;
DP4   opos.y, mvp[1], npos;
DP4   opos.z, mvp[2], npos;
DP4   opos.w, mvp[3], npos;
# Secondary color and texture are sent directly
# to the fragment program
MOV outcolor2,color2;
MOV outtexcoord,intexcoord;
END
```

Figure 4.18: The Vertex Program

```
!!ARBfp1.0
# Composite X SORTING
TEMP    tex;
TEMP    ainv;
TEMP    newc;
TEMP    tmp1;
TEMP    tmp2;
TEMP    kilf;
TEMP    chck;
TEMP    aray;
# Color actually contains weight factor
ATTRIB  ray = fragment.color;
# Secondary color contains the color of the current object
ATTRIB  cor = fragment.color.secondary;
TEX tex, fragment.texcoord, texture, 2D;
# Intialize color to object color
MOV newc,cor;
# Initialize alpha value to 1
MOV newc.a,1.0;
# Truncate weight's alpha channel to 1
MIN aray.a,1.0,ray.a;
# Multiply color by weight calculated
# in the vertex program
MUL newc,newc,aray.a;
# Check if component X has the largest value of X, Y and Z
# for 'Y Sorting' change 'r' to 'g' and 'g' to 'r', and
# for 'Z Sorting' change 'r' to 'b' and 'b' to 'r'.
SGE tmp1,-ray.b,-ray.r;
SGE tmp2,-ray.g,-ray.r;
# Zero the color and alpha if X is not the largest value
MUL newc,newc,tmp1;
MUL newc,newc,tmp2;
# Multiply the calculated color by the 2-D gaussian kernel
MUL newc,newc,tex;
# If it does not belong to X Sorting or if the alpha channel
# is too low, then kill Fragment
SLT kilf,newc.a,0.05;
MUL chck,kilf,-1.0;
KIL chck;
# Otherwise clamp alpha to 0.8 and send the new output color
MIN newc.a,0.8,newc.a;
MOV result.color,newc;
END
```

Figure 4.19: Fragment program to calculate final color of the fragment

```
!!ARBfp1.0
# Normalize Alpha
TEMP    tex;
TEMP    ainv;
TEMP    csum;
TEMP    tmp;
TEX tex, fragment.texcoord, texture, 2D;
# Calculate 1/alpha
RCP ainv,tex.a;
# Multiply 1/alpha to the current color of the pixel
MUL csum.r,tex.r,ainv;
MUL csum.g,tex.g,ainv;
MUL csum.b,tex.b,ainv;
MOV csum.a,1;
MOV result.color, csum;
END
```

Figure 4.20: Fragment Program to Normalize Alpha

Chapter 5

Results

"If it is fast and ugly, they will use it and curse you;
if it is slow, they will not use it."

David Cheriton

5.1 Chapter Organization

In this chapter we present the results of our research. We begin with the 4-D approach, the light field hyperlines in section 5.2. Both camera and geometry hyperline images are shown. In case of geometry hyperline rendering (section 5.2.2), we also show how to manipulate the rendering parameters to obtain different results in the final image.

Next, in section 5.3 the results of rendering uncertainty are presented, both with hyperpencils and the ellipsoidal Gaussian kernels. We also compare unconstrained uncertainty and delta uncertainty, using the 3-D approach.

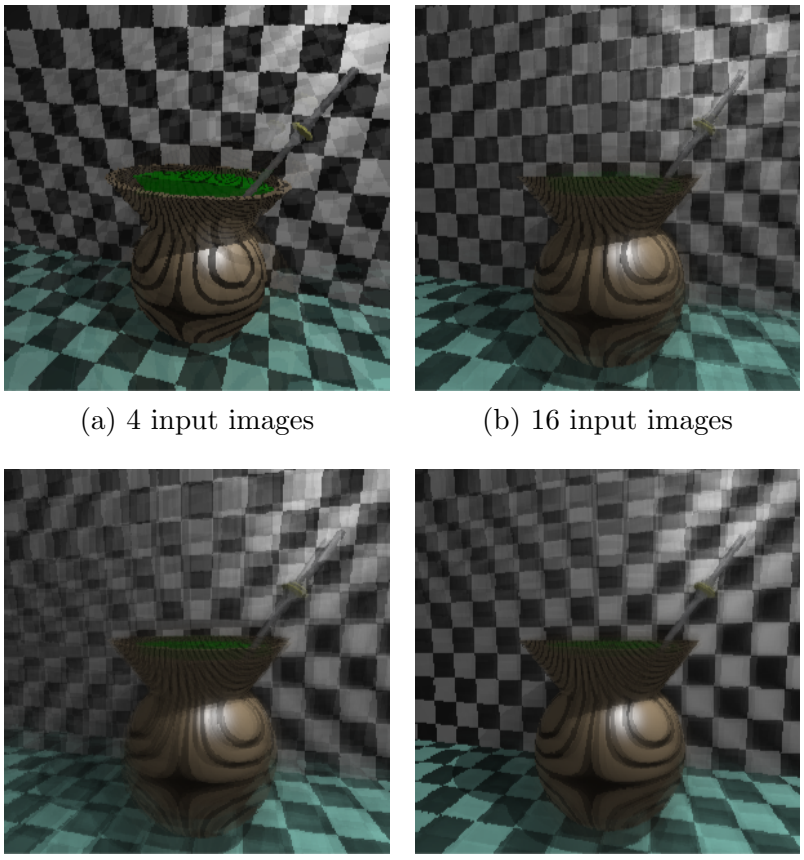
Finally, we present a discussion about the advantages and disadvantages of using the 4-D or the 3-D approach, section 5.4.

See page 104 in the end of this chapter for tables that contain references for the rendering methods, terms and concepts presented here.

5.2 Light Field Hyperline

5.2.1 Rendering Camera Hyperlines

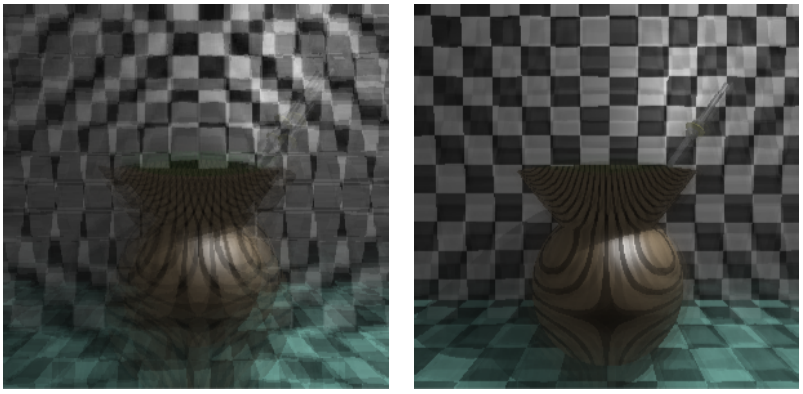
We have used a synthetic model called '*Mate*', with specular reflections, rendered with a ray tracer, to test our camera hyperline rendering. All rendered images in this section maintain the same Z position for the $u-v$ and $s-t$ planes. No geometry proxies were used. Figure 5.1 shows that the more input images we have, the better the quality of the final image. In this case, one ray from every input image is selected for blending one output pixel. Note that in (a), where only 4 images were used, the image appears to be rendered from a different viewpoint. This demonstrates that basic light field rendering (with no geometry information) requires a dense number of input images.



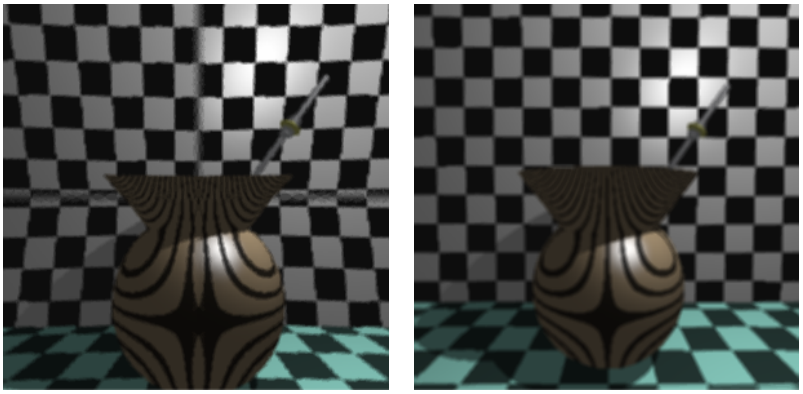
(a) 4 input images (b) 16 input images

(c) 36 input images (d) 64 input images

Figure 5.1: Camera Hyperline Rendering using one ray from every input image, weighted by the inverse of their 4-D distance. All input images have 300x300 pixels.



(a) 16 images, closest ray (b) 64 images, closest ray



(c) 16 images, closest 16 rays (d) 64 images, closest 16 rays

Figure 5.2: Comparison of two different methods of selecting rays for camera hyperline rendering. (a) and (b) use one ray per input image, (c) and (d) choose the overall best 16 rays.

There are many ways of selecting rays using camera hyperlines. Figure 5.2 compares our previous method, i.e., blending one ray from each input image, versus selecting the best overall 16 rays, based on their 4-D distances. Even when we use 64 input images, choosing only one ray still causes some ghosting artifacts. Note that if we use 64 input images, we have much more rays to chose from. Most of the time, the chosen ray is not the same as if we had 16 input images, as Figure 5.2a and 5.2b shown. Similarly, if we chose 16 rays they will probably not be the same 16 rays for a different number of input images (Figure 5.2c and 5.2d).

Some parts of the image in Figure 5.2d, like the plane with the checkboard pattern and the *Mate* straw, still look blurred. This is due to the fact that we don't have any



(a) LFR 8x8 images 256x256



(b) LFR 32x32 images 256x256



(c) CHR 8x8 images 128x128



(d) CHR 8x8 images 256x256

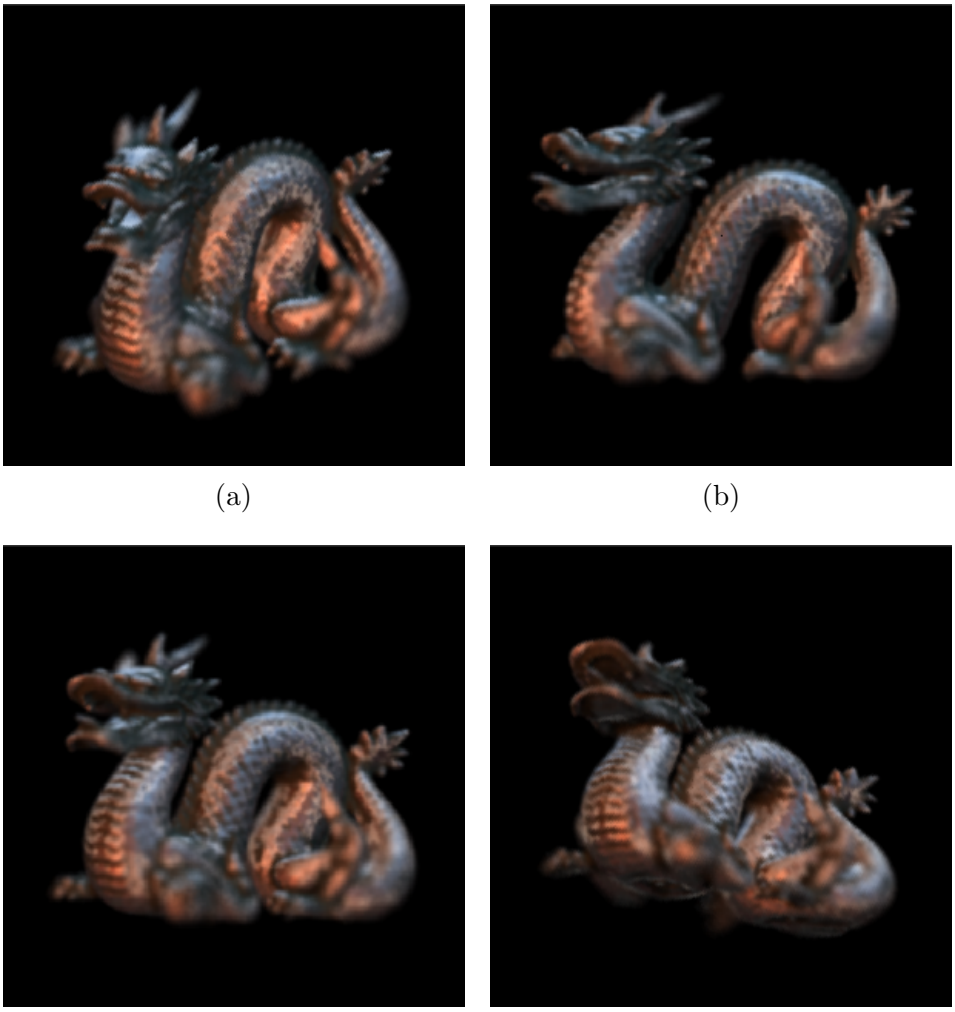
Figure 5.3: Stanford Dragon: comparison of camera hyperline rendering (CHR) and light field rendering (LFR).

geometry proxies. The whole geometry is approximated by the focal plane $u - v$.

The Stanford Dragon is used to compare camera hyperline rendering and the original light field rendering, Figure 5.3. The closest 8 rays were used for blending. The blending weight is the inverse of the 4-D distance. Note that (c) and (a) look similar even though (a) has only 1/4 of the information used for (c). (d) and (b) also look similar, but we used only 64 images in comparison with 1024 for the light field rendering.

Figure 5.4 shows the Stanford Dragon as seen from different points of view, using camera hyperline rendering.

Next, we show the effects of using a dynamic focal plane (Figure 5.5). The foreground of the scene is a shiny teapot; the background is a texture-mapped plane. There are 50 source views. Each virtual pixel is selected and equally blended from the 25 closest camera hyperlines. Notice how the background becomes blurry in the middle image, and the teapot almost disappears in the right image.



(a) (b)

(c) (d)

Figure 5.4: Camera Hyperline Rendering: Stanford Dragon. Input: 8x8 images 256x256. Blending weight: inverse of the 4-D distance from closest 8 rays.



Ground truth Focus at foreground Focus at background

Figure 5.5: Dynamic focal plane by adjusting β, as shown in [47].

5.2.2 Rendering Geometry Hyperlines

Figure 5.6 shows the input images of *Mate* used in this section. Notice that, here, it does not have specular reflections. Figure 5.7 compares the global ray space technique with the local ray space technique. Since the global method uses only one pair of parameterized planes, it is obvious that when the viewing ray is close to parallel to those planes we will be able to see through objects in the scene.

Figure 5.8 demonstrates which sorted list is chosen for each virtual ray in order to handle occlusion correctly. The three lists are not searched all the time, though. It depends on the virtual rays. In many views only one list is enough.

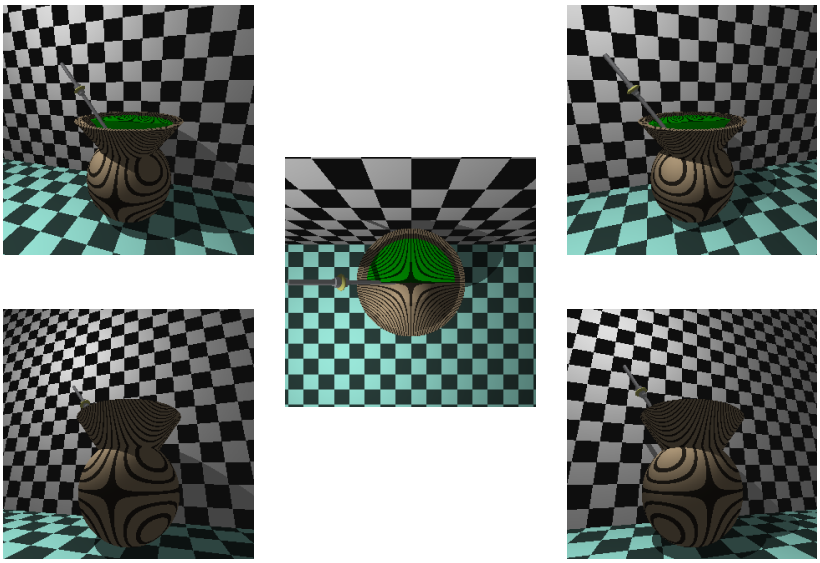
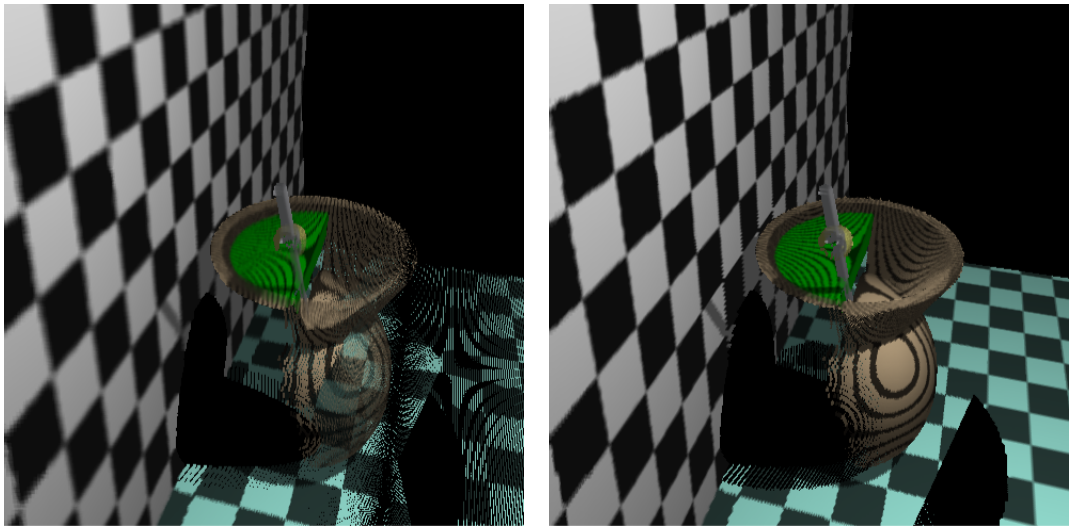


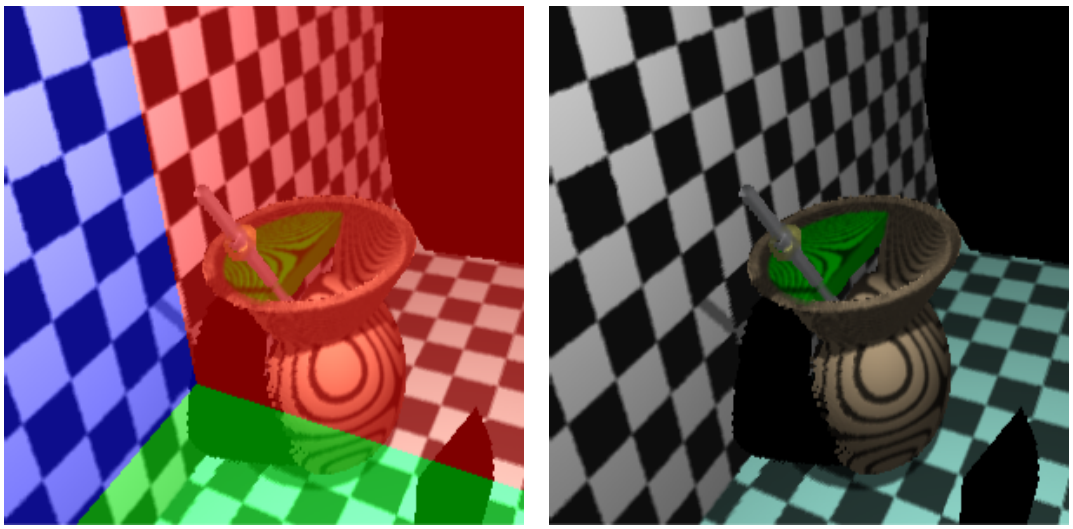
Figure 5.6: *'Mate'* Input Images, 450000 points

Figure 5.9 depicts the use of the depth window threshold. Note that if V_d=0 as in (a) our solution is very aliased. This happens because the algorithm uses only one geometry hyperline ray whose distance is below D. If V_d is large (c) we can see through the objects in the scene. Using a relatively small V_d as in (b) the image is smooth and occlusion is handled. The choice of V_d depends on the density of the 3-D data (how many points we have) and how far are the points from each other. The closer they are, the smaller V_d may be. Figure 5.10 illustrates the same idea using real images. Range data for this model and the *car wreck scene* (shown later) are a courtesy of University of North Carolina at Chapel Hill IBR group.



(a) Global Ray Space (b) Local Ray Space

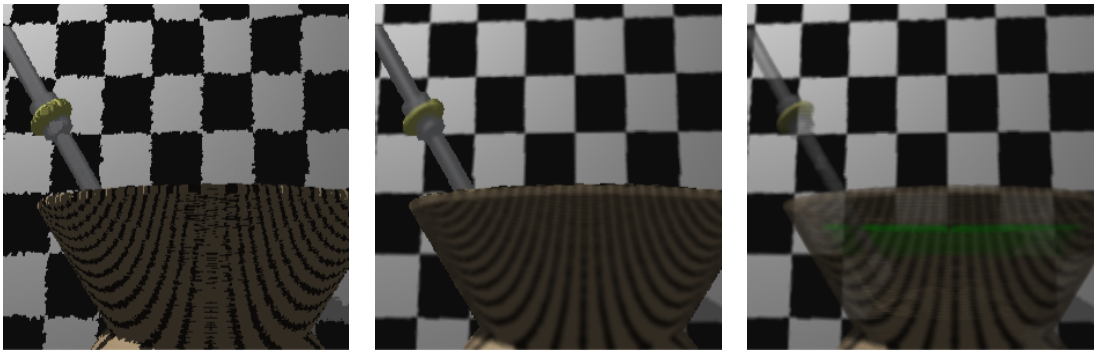
Figure 5.7: Comparison between images produced by the Global Ray Space method and the Local Ray Space method. Large black regions represent surfaces not visible in any of the input images.



(a) Red = X, Blue = Z, Green = Y axis (b) Final Image

Figure 5.8: Local Ray Space sorting list choice is defined based on the virtual ray direction.

Figure 5.11 shows the effect of changing the 4-D distance threshold D for the *Mate* model. If D is small as in (a) and (b), some pixels do not have any solution and holes are generated. Also, it is possible to see through the model at some points. If D is large (d), the image is too blurry and the silhouettes look wrong, particularly in the stirring rod. For $D = 0.5$ we have the best solution for this model.



(a) $V_d = 0$ (b) $V_d = 3$ (c) $V_d = 100$

Figure 5.9: Comparing different choices of the depth window threshold V_d

The choice of D depends on how the planes are parameterized and how many input images we have. In the case of the *Mate* dataset, one unit in u is equal to the pixel width and one unit in v is equal to the pixel height. If only one image is used, D=1.0 is enough for a good reconstruction. The value of D can be smaller if we have more views from the same region. If it were possible to sample the infinite plenoptic function, both D and V_d could be zero. Some holes may still appear if the tangent plane of the sampled surface point is close to perpendicular to the $u-v$ and $s-t$ planes. In this situation, two or more input views from that region are necessary in order to fill those holes.

Figure 5.12 compares rendering results of the "car wreck scene" between a polygonal renderer and our GHL renderer Not all the differences are noticeable at the current resolution. So we only point out a few. On the left car, notice the window, the hubcap and the seam between the front and back doors. On the right car, notice areas around the rear wheel and the vertical ridge on the hood. In all these cases, the right image looks smoother and more visually pleasant.

5.2.3 Two Problems

Most of the time research is guided by ideas. Sometimes, though, it is guided by obstacles. In this section we present two challenging problems we had and how we solved them.

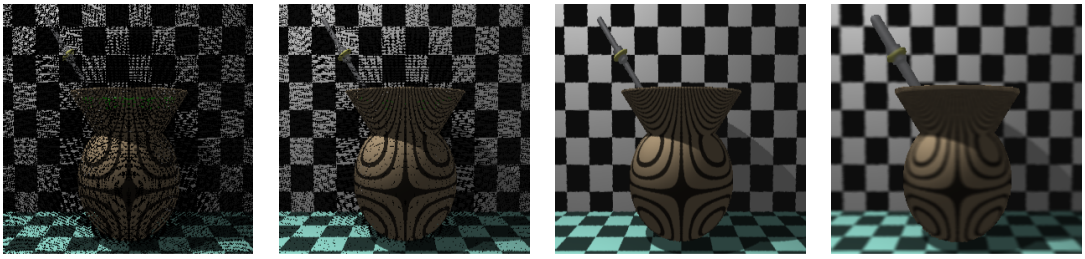


(a) $V_d = 100$ (b) $V_d = 0.2$

Figure 5.10: The Frederick P. Brooks, Jr. Reading Room in Sitterson Hall at UNC-Chapel Hill. This model has 362880 Geometry hyperlines from 7 input views. The gap in the side wall of the left hand chair is caused because that surface was not visible in any input view. The bottom row image (b) is a close-up rendering of the top row image. V_d is also 0.2 in the top row image.

The Image Consistency Problem

Initially in our research we did not have the 4-D distance threshold. We used an image-based threshold instead. For all hyperlines, we defined a fixed pixel radius that should contain its footprint. For blending, we used the inverse of the 4-D squared distance



(a) $D = 0.01$ (b) $D = 0.1$ (c) $D = 0.5$ (c) $D = 10$

Figure 5.11: Comparing different choices of the 4-D distance threshold D



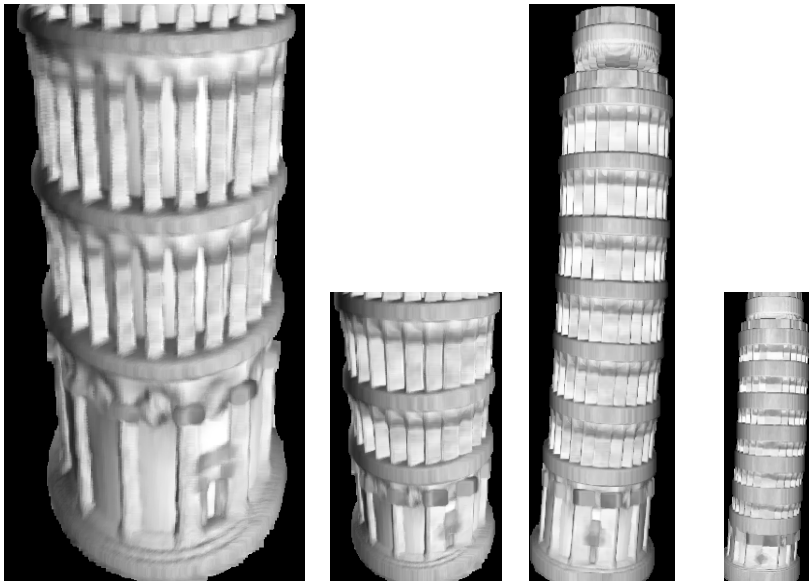
(a) 16 Polygon-based rendering (b) GHL-based rendering

Figure 5.12: The 'Car Wreck Scene', using 9 images (864x576) and their range data

(equation 3.14). The use of an image-based instead of a ray space-based threshold made rendering faster.

Later we found out that although the solution produced good and fast images, it was not *consistent* for different viewpoints and resolutions. This means that for a static image, the rendering was fine. But for a movie, the contribution of each hyperline was not consistent. Figure 5.13 illustrates the problem. In order to make it consistent using this approach, we would have to change the threshold value for each image.

When rendering with camera hyperlines we don't have this problem. This is because (1) the number of input rays is dense and (2) a fixed number of camera hyperlines are chosen for each pixel. In this way the contribution of each input ray is limited. For a sparse input, we also need to limit somehow the contribution of each individual hyperline in ray space. The solution was to create the 4-D distance threshold, so it would be independent of the resolution and viewpoint position. Figure 5.14 shows that the use of the 4-D distance threshold maintains consistency between different resolutions and different viewpoints. Although the solution is simple, the problem of identifying the hyperline footprint



(a) 320x620 (b)160x310 (c) 150x600 (d)75x300

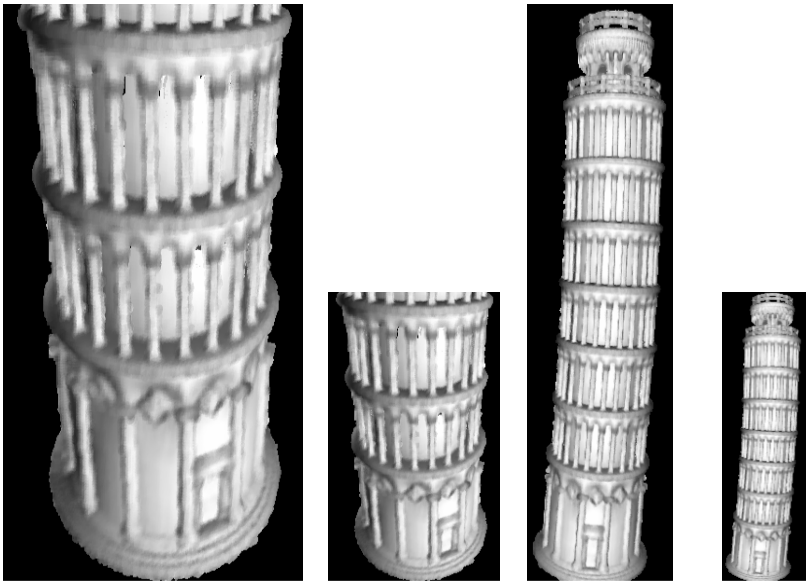
Figure 5.13: The '*Pisa* Tower' model rendered with different resolutions and viewpoints, using an image-based threshold.

area remained. This was later solved by including a warping stage (section 3.5.4). The original input point that defined the hyperline is warped to the novel view. We then center a *quadrilateral* around it. The quadrilateral is always bigger than the hyperline footprint.

The Foreground Expansion Problem

The whole idea of geometry hyperline rendering is to weight based on the distance of two rays; the virtual ray and all rays from each hyperline. The problem is that, using this metric, the silhouettes of the object tend to expand.

Initially, we used a sophisticated mechanism to check if a hyperline pixel contribution for a specific depth was from an edge or not. If the fragment was not completely surrounded by fragments of the same depth cluster, it was discarded. Since we have sub-pixel accuracy, this was done per-pixel; no neighbor pixel was visited. For instance, assuming that we have n geometry hyperlines clustered around a depth d. We want to know if this cluster belongs to an edge or not. The basic idea can be described as follows. First, we calculate where in the pixel the point represented by each geometry hyperline is projected. Now, let l be a line, passing through the center of the pixel, that partition the pixel in two.



(a) 320x620 (b)160x310 (c) 150x600 (d)75x300

Figure 5.14: The '*Pisa* Tower' model rendered with different resolutions and viewpoints, using a 4-D distance threshold D

If there is any configuration where all n projections are contained in one partition, we say this cluster is an edge, and we bypass it. It turns out that we can implement this efficiently in 4-D ray space. More details about this implementation can be found in [47].

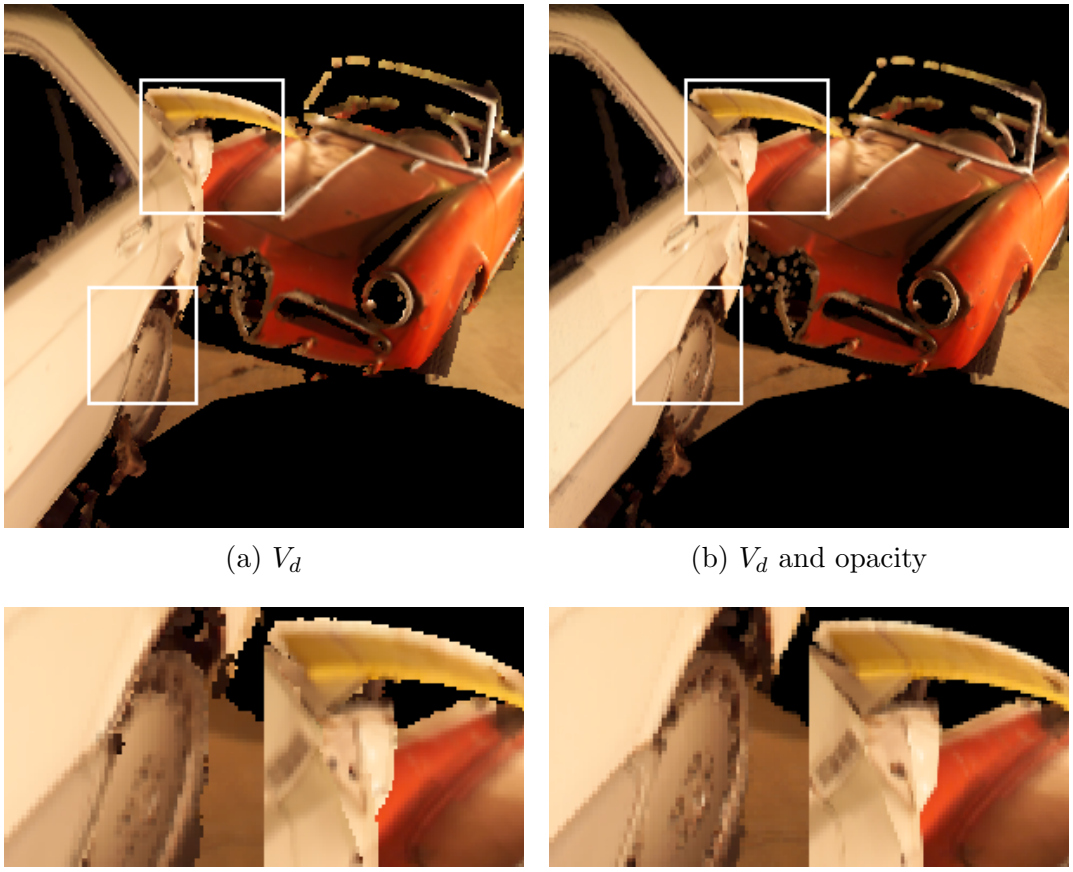
The problem with this approach is that sometimes a few hyperlines would miss the pixel and the algorithm would discard that fragment. Visiting neighbor pixels was too costly. Since the problem is minimized when the data is dense enough, we decided to ignore it.

Recently, we decided to use the *opacity* term in our geometry hyperline rendering (equations 3.23 and 3.24, section 3.6.2). This is the hyperpencil solution for occlusion. Here, $\alpha_i(r) = g_i(h_{dist})$. It turns out that using the opacity and the depth threshold produces much better images, especially near the edges, as shown in Figure 5.15.

5.3 Rendering Depth With Uncertainty

Figure 5.16 and Figure 5.17 shows the 'box' and the 'house' input images respectively, and the picture of their uncertainty maps, using the *unconstrained* uncertainty[1],

[1]See section 2.4 for more details.



(a) V_d (b) V_d and opacity

(c) details of (a) (d) details of (b)

Figure 5.15: Comparison of using V_d and opacity in the original Geometry Hyperline Rendering and using only V_d

obtained by the original range-space approach. As expected, the more textured a region is the more certain it is. Figure 5.18 shows the result of hyperline pencil rendering (with q=4, $V_d = \infty$) compared with several other methods using only the best matching depth z_b.

Figure 5.19 illustrates how the use of depth clustering helps to fine tune occlusion. Since it has homogeneous color, the roof of the house has high uncertainty, which means it has low opacity. In this case, solving occlusion by opacity is not enough.

Finally, Figure 5.20 presents a comparison between our 3-D and 4-D frameworks. Notice how the final images are similar, even though the 3-D approach rendering time was about two seconds per image, while the 4-D approach rendering time was about six seconds per image.



Figure 5.16: Picture of (b) uncertainty map (u_i values, in green) for the box input images (a). In (b), brighter means data with higher uncertainty.

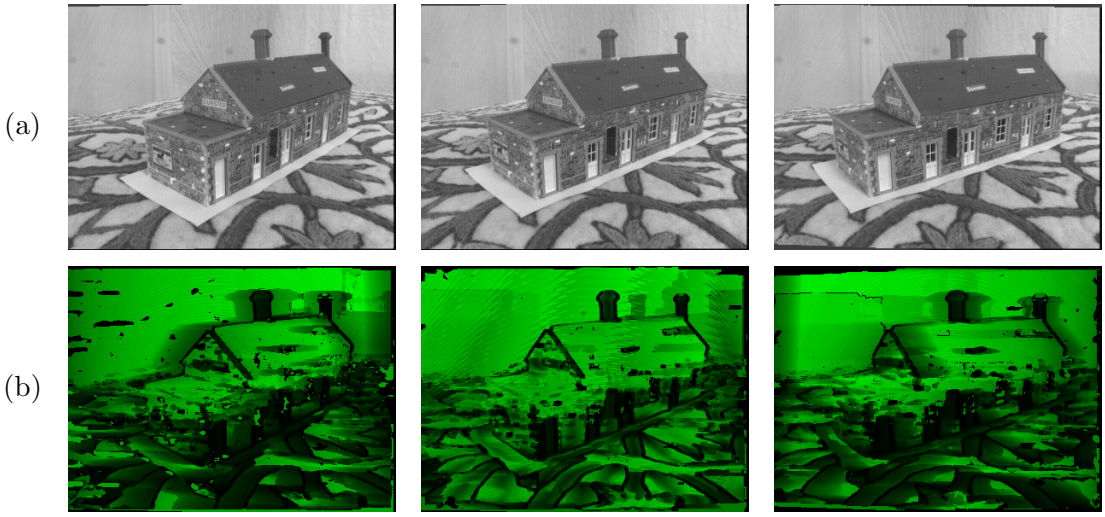


Figure 5.17: Picture of (b) uncertainty map (u_i values, in green) for the house input images (a). In (b), brighter means data with higher uncertainty.

5.3.1 Delta vs. Unconstrained Uncertainty

For this section, we use only the 3-D approach for rendering.

As seen in section 2.4, we have two techniques to obtain our uncertainty maps. The *Unconstrained* uncertainty is obtained by the original range-space approach. The *Delta* uncertainty is obtained by starting our search from the disparity δ of the best depth, and restricting the search for a region defined by $\Delta = \delta \pm \epsilon$. Recall that disparity here means

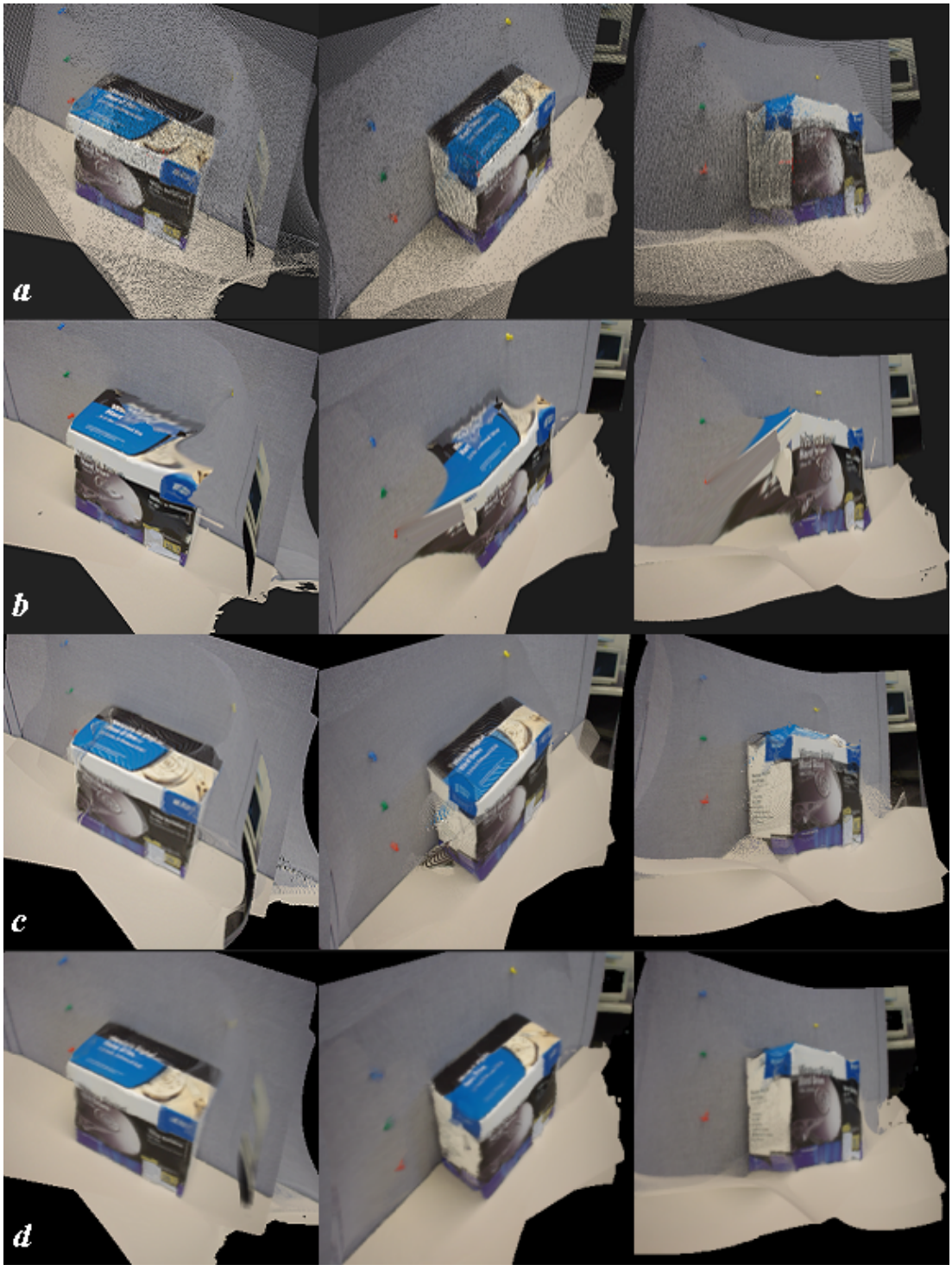
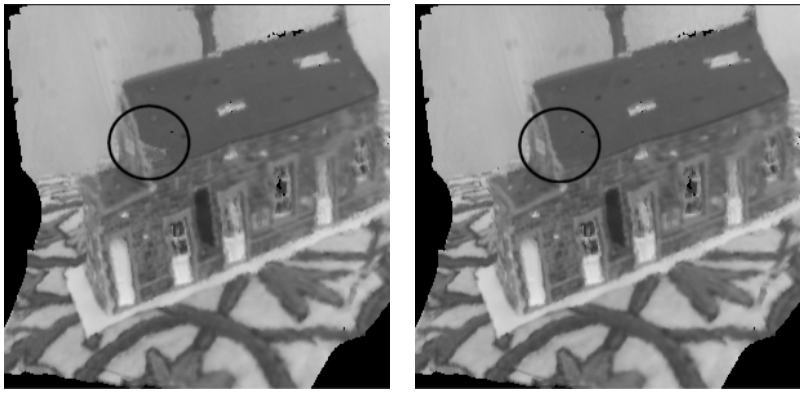


Figure 5.18: Comparison of (a) input points, (b) polygon meshes, (c) geometry hyperline rendering and (d) hyperline pencil rendering



(a) $V_d = \infty$ (b) $V_d = 30$

Figure 5.19: Depth clustering may still be used to help solve occlusion

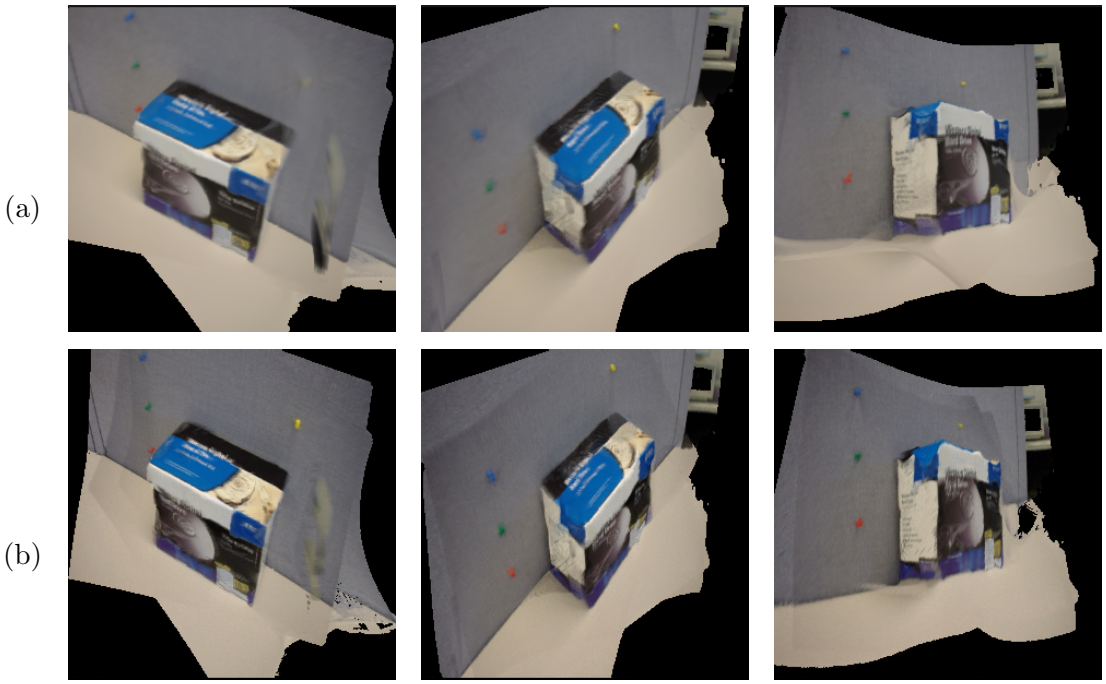


Figure 5.20: Comparison of (a) Hyperpencil rendering and (b) volume splatting.

the *generalized* disparity (equation 2.2).

Figure 5.21 compare the values of u_i (equation 4.14) for both types of uncertainty. Figure 5.22 shows the contour graph of the depth size $(z_e - z_s)$ for one of the box images. Finally, figure 5.23 plots the depth size of one line from one of the box images. There are two important differences: first, the unconstrained uncertainty is more uncertain. Second, the delta uncertainty is smoother. The reason of why unconstrained uncertainty is less certain is obvious: since it is unconstrained, the uncertainty will be much higher for homogeneous

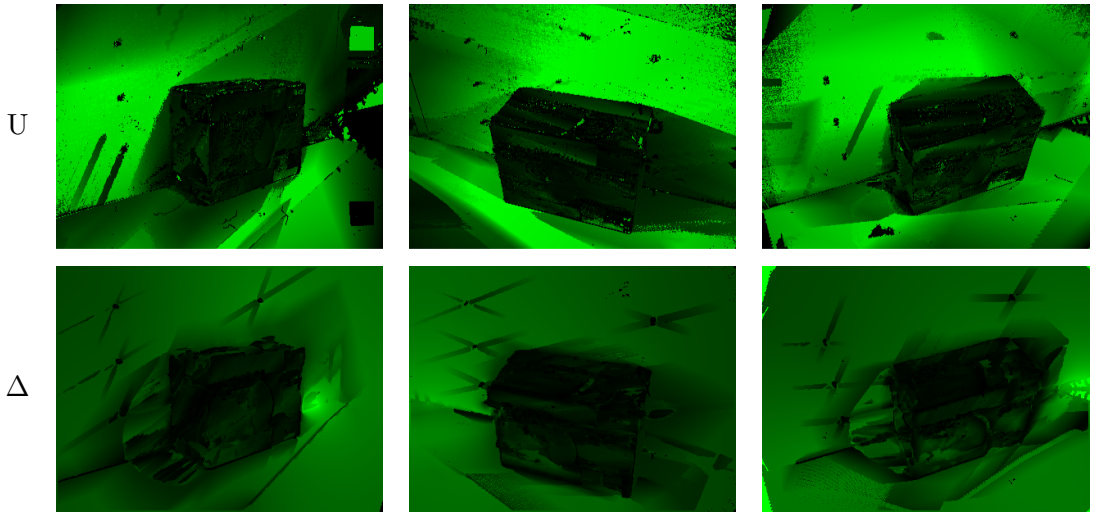
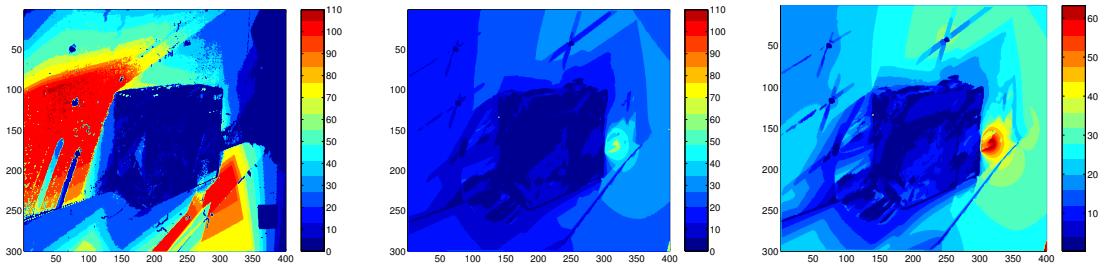


Figure 5.21: Comparison of the uncertainty values u_i of Delta Uncertainty (Δ) and the Unconstrained Uncertainty (U). Brighter means data with higher uncertainty.

regions. The delta uncertainty has the additional assumption that the estimated depth cannot be too far away from the actual depth. Therefore, even if the matching curve is nearly flat (which indicates a homogeneous area) the uncertainty region will be limited by the disparity threshold ϵ.



(a) Unconstrained, max=110 (b) Delta, max=110 (c) Delta, max=63

Figure 5.22: Contour graphs of the depth size $z_e - z_s$ for the unconstrained uncertainty and the delta uncertainty. Notice that (b) and (c) are both delta uncertainty with different scales.

The reason why the delta uncertainty is smoother is more complicated. The delta uncertainty is obtained from a *disparity* region. This region is limited by ϵ. Since $\epsilon = 0.5$ in this example, it turns out that almost all homogeneous regions will reach the limit defined by $\delta \pm \epsilon$. This means that the size of the uncertainty region, if defined as disparity, is equal to 1. Figure 5.24 shows the uncertainty disparity region size (delta uncertainty) as a gray

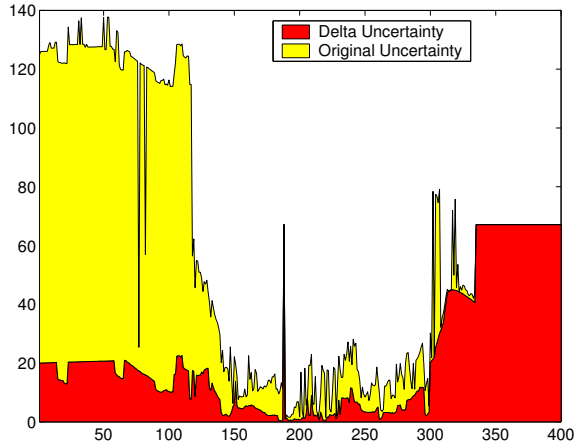


Figure 5.23: Depth size plot (for Y=150,X=0 to 399)

scale image. The whiter it is, the larger is the region size. As we can see, most pixels are completely white, which means that the disparity size equals to one.

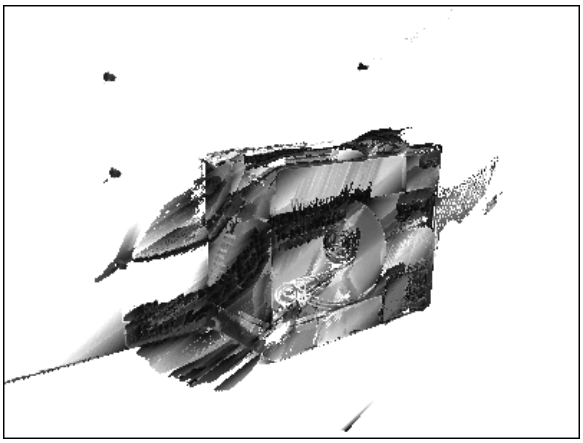


Figure 5.24: Uncertainty disparity region size for the Delta Uncertainty, $\epsilon = 0.5$

On the other hand, the uncertainty values u_i used for rendering are based on *depths*, not disparity. Even though most of the homogeneous areas have the same uncertainty disparity region size 1 (i.e., its size is equal to 2δ), as shown in figure 5.24, the *actual* uncertainty value u_i depends on the depth size. The depth uncertainty size for two adjacent pixels with disparity size 2δ is not the same, but it varies smoothly. Let p_i be the 3-D location of an input pixel i in the image plane. Let $2d_i$ be the depth uncertainty region size given by $s_i - e_i$, where $e_i = b_i + d_i$, $s_i = b_i - d_i$, and where b_i is the best estimated depth, s_i is the starting depth and e_i is the ending depth. Based on equation 2.2, the generalized

disparity of s_i and e_i is given by:

$$\delta_{s_i} = \frac{b_i - d_i}{|p_i - C|}$$

$$\delta_{e_i} = \frac{b_i + d_i}{|p_i - C|}$$

where C is the input camera center of projection. The disparity size is then given by

$$|\Delta_i| = \delta_{e_i} - \delta_{s_i}$$

Now, let p_j be an adjacent pixel of p_i where $|\Delta_i| = |\Delta_j|$. We then have

$$\frac{b_i - d_i}{|p_i - C|} - \frac{b_i + d_i}{|p_i - C|} = \frac{b_j - d_j}{|p_j - C|} - \frac{b_j + d_j}{|p_j - C|}$$

$$\frac{2d_i}{|p_i - C|} = \frac{2d_j}{|p_j - C|}$$

Since C is not on the image plane, $|p_i - C| \neq 0$ and $|p_j - C| \neq 0$. In this way, we avoid the derivative discontinuity of the square root at the zero input value, so the square root is a smooth function. Therefore, having most of the image with a disparity region size equal to one produces a smooth uncertainty map.

Now we have to show that it is advantageous to have most of the image with disparity size 2δ. In terms of depth uncertainty, the meaning of this can be explained as follows. A disparity region of size 2δ results in different values of u_i depending on where in the input ray it is located. For a fixed disparity region size 2δ, the closer the best (estimated) depth is to the viewpoint, the *smaller* is u_i. Therefore, if the object is really far from the observer, even if its uncertainty disparity region is small, the resulting depth region will be large.

To illustrate this, notice the red and yellow areas that are to the right of the box, Figure 5.22c. They have the largest u values of this image. Nevertheless, Figure 5.24 shows that several other areas also have the maximum uncertainty disparity size (2δ). Now, look at Figure 5.25, which contains the 3-D points based on the best depths. We can easily tell that the region marked with a red circle has the worst reconstructed depths for that image.

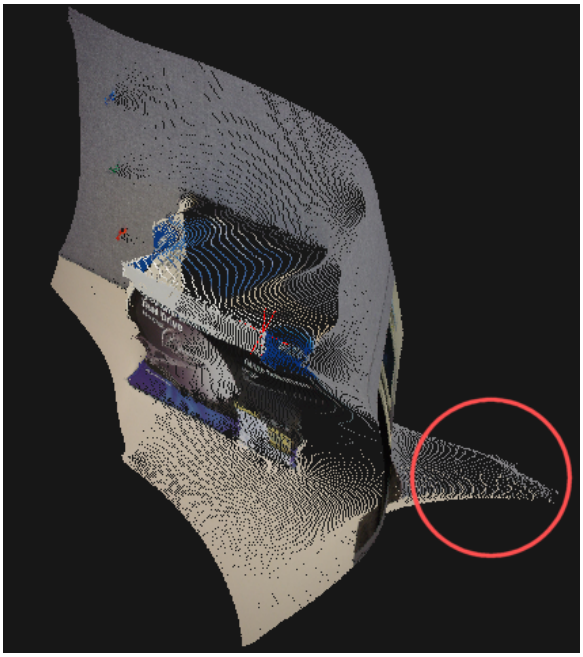


Figure 5.25: Virtual view of the reconstructed 3-D points for one of the box images

This is the same region where some of the largest u values of the image are located. Since our final uncertainty is based on depth and not disparity, our technique detects this region because (1) it is located in a homogeneous area and (2) the estimated depths are too far from the viewer.

Finally, Figure 5.26 compares the rendering of the box model using both types of uncertainty. As expected, images that used the delta uncertainty look sharper.

5.4 The Choice of the Best Approach

The best approach for rendering uncertainty depth, at this moment, is the 3-D approach (3DA). This does not mean that the 4-D approach (4DA) may not be the best in the future, though.

The 3DA rendering is much faster, but it produces a few more artifacts due to the limited numerical capabilities of the hardware. This can be seen in the right part of the first picture in Figure 5.20(a). The fragments that would contribute to that region were discarded; if they were included, the numerical errors would make the whole region saturate to white (this happens because sometimes the final alpha value may be so small that when

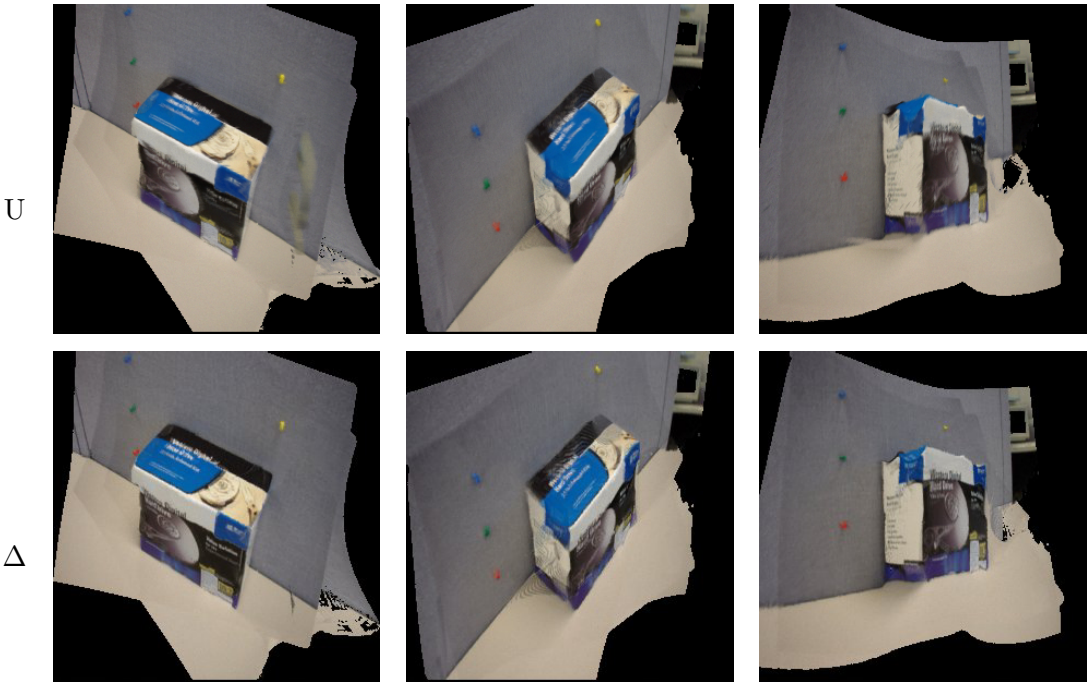


Figure 5.26: Comparison of rendering using Delta Uncertainty and Unconstrained Uncertainty.

we try to normalize it - section 4.4.6 - the result is a very large number). The images are a little bit sharper, though. This means that it is easier to see the details. On the other hand, the 4DA solution does a better job at blending. It is more difficult to tell the input images apart in the first row (4DA) than it is in the second (3DA).

The problem of the 4DA being slow is twofold. First, we have a costly integral to solve for each hyperline (equation 3.22). Second, its implementation is all done in software. We will look into those issues in the future, which means implementing some parts of the 4DA in hardware and maybe to find a faster alternative to the integral. After that, the comparison will be more fair.

5.5 Summary

In this chapter we presented the results of our research, including camera hyperline rendering and geometry hyperline rendering. We presented two techniques for rendering uncertainty depth, hyperpencil rendering and ellipsoidal Gaussian kernels. We also compared

two approaches to obtain the uncertainty, the unconstrained and the delta uncertainty. The latter results in sharper images than the former.

We have shown two major obstacles we had in our research - (1) the image consistency problem and (2) the foreground expansion problem - and how we solved them by (1) using a 4-D distance threshold and (2) using an opacity-based accumulation.

Finally, we compared the two approaches for rendering uncertainty depth, the 3-D approach (3DA) and the 4-D approach (4DA). The best solution so far is the 3DA because it is much faster, but we believe we will have a better comparison in the future when the 4DA is optimized.

Rendering Methods	Acronym	Approach	Input
Light Field	LFR	4-D	only images
Camera Hyperline	CHL/CHR	4-D	only images
Geometry Hyperline	GHL/GHR	4-D	Images with depth
Hyperpencil	HPR	4-D	Images with depth and uncertainty region
3-D Gaussian Volume Splatting	3DG	3-D	Images with detph and uncertainty region

Table 5.1: Summary of rendering methods

Rendering Methods	Section	Figures
Light Field	1.2.1, 3.3.1	5.3, 5.4
Camera Hyperline	3.4	5.1, 5.2, 5.3, 5.4, 5.5
Geometry Hyperline	3.5	5.7, 5.8, 5.9, 5.10, 5.11, 5.12, 5.13, 5.14, 5.15, 5.18
Hyperpencil	3.6	5.18, 5.19
3-D Gaussian Volume Splatting	4.4	5.20, 5.26

Table 5.2: Rendering methods: references and figures

Concepts	Context	Section	Figures
Hyperlines	CHR/GHR	3.3.2	See CHR/GHR
Dynamic Focal Plane	CHR	3.4.1	5.5
Global/Local Ray Space	GHR	3.5.5	5.7
Multiple Sorting Lists	GHR/3DG	3.5.5, 4.4.2	5.8
4-D Distance Threshold D	GHR	3.5.3	5.11
Depth Window Threshold V_d	GHR	3.5.2	5.9, 5.10
Opacity	HPR	3.6.2	5.20, 5.15
Depth Uncertainty Region	HPR/3DG	2.4	5.16, 5.17
Unconstrained Uncertainty	HPR/3DG	2.4	5.21, 5.22, 5.23, 5.26
Delta Uncertainty	HPR/3DG	2.4	5.21, 5.22, 5.23, 5.24, 5.26

Table 5.3: Summary of concepts

Chapter 6

Conclusions

"I always avoid prophesying beforehand, because it is a much better policy to prophesy after the event has already taken place."

Winston Churchill

Simple texture mapping is not enough. Since Computer Graphics researchers decided to look further into new ways of mapping images to geometry, rendering algorithms in general became more realistic. At some point even geometry wasn't necessary, we could render images directly from light rays. However, a simple light field is not enough either. Both geometry and light rays are necessary for obtaining more realistic images. Image-Based Rendering techniques are usually divided between the use of geometry with fancy texture mapping mechanisms and the use of light fields with geometry proxies (Kang et al. in [50] describe them as physically-based and appearance-based representations). In this context, our approach of rendering taking into account uncertainty uses both representations.

6.1 Summary

We have presented here a new light field rendering framework that combines the benefits of many existing approaches. It is based on a dual representation of the world space such that 3-D points and lines become 4-D hyperlines and hyper points respectively. We have shown that using the 4-D distance for weighting the rays produces better images than the original light field rendering.

If per-pixel depth is given, it is possible to have a point cloud rendering in the context of light field ray space. In our framework, 3-D points are modeled as geometry hyperlines in ray space. For each virtual ray, we find and blend the closest rays of each geometry hyperline that are within a given 4-D distance threshold D.

In addition, our approach is capable of rendering data whose depth is not accurate. We take advantage of the knowledge obtained during stereovision reconstruction. Thus, not only the best match, but also an uncertainty depth region is used for rendering. Here, our uncertainty region is based on the error regions obtained by range-space search, presented in [22]. The range-space approach is a multi-pass method. First, we calculate the matching curve of each pixel with its possible correspondence in all other images. This may produce several error regions per input pixel. Second, we select one error region from each matching curve. This is our depth uncertainty region. We also proposed a new, tighter, uncertainty region estimation, called Delta uncertainty. Our approach is similar to the previous one, but we start from the best estimated depth and we constrain our search within a specific generalized disparity region around it. Since the uncertainty region is smaller, images rendered with delta uncertainty look sharper.

In 4-D ray space, this uncertainty region aggregate many geometry hyperlines, which forms a hyperline pencil. Ray selection takes into account the contribution of all geometry hyperlines in this pencil, based on some basic rules (like proximity to the best matching depth).

Finally, we have implemented a 3-D version to render our uncertainty depth region based on volume splatting. Each uncertainty region is modeled as an ellipsoid. The ellipsoid projection is an ellipse, and we approximate the 3-D Gaussian kernel by a 2-D elliptical kernel for blending.

In the end, the results of rendering with both the 3-D and the 4-D approach were similar. However, the 3-D approach is much faster. The reasons are twofold. First, it does not have to solve a costly integral for each uncertainty region. Second, most of the algorithm is implemented in hardware. Nevertheless, the 3-D approach still produces some artifacts due to the limitations of the hardware implementation.

6.2 Future Work

We have three major areas of future research in the topics presented in this dissertation. They are related to the three major chapters in this dissertation.

6.2.1 Uncertainty Region Estimation

First, we believe that it is possible to have a better (tighter) estimation of the uncertainty region. Second, since the computer hardware is getting faster every year, it may be possible to estimate the uncertainty region and render it in *real time*. As of now, we start from a good seed and we calculate the uncertainty region around it. We believe that if we also take into account depth discontinuity we can achieve much better results. Also, it is possible to have an adaptive window for searching. However, the bigger is the window used for this matching, the slower the rendering will be. Another example is the error threshold. The bigger it is, the farther we have to search in the epipolar line. Therefore the next step in obtaining the uncertainty region estimation is going to be a trade-off between speed and accuracy.

6.2.2 The 4-D Approach

As we said before, we have two important improvements for the 4-D approach. First, we have to optimize our rendering. The best way to do this will be to approximate the integral in equation 3.22 with another, faster, weighting mechanism. Second, it is possible to implement the later stages of the hyperpencil rendering in hardware, similar to how the 3-D approach is implemented.

6.2.3 The 3-D Approach

There is at least one stage left to be optimized in the 3-D approach. The projection of the ellipsoid to the image plane is the costliest computation now, and it is done in software. We believe that this stage can also be done in hardware.

Bibliography

"He who cannot draw on three thousand years is living hand to mouth."

Goethe

[1] Levoy, M. and Hanrahan, P., *Light field rendering*, Proc. of SIGGRAPH 96 (1996), pp. 31–42.

[2] Gortler, S. J., Grzeszczuk, R., Szeliski, R. and Cohen, M. F., *The Lumigraph*, Proc. of SIGGRAPH 96 (1996), pp. 43–54.

[3] Chai, J.-X., Tong, X., Chan, S.-C. and Shum, H.-Y., *Plenoptic Sampling*, Proc. of SIGGRAPH 2000 (2000), pp. 307–318.

[4] Schirmacher, H., Ming, L.and Seidel, H.-P., *On-the-Fly Processing of Generalized Lumigraphs*, Proc. of Eurographics 2001 (2001), pp. C165–C173.

[5] Buehler, C., Bosse, M., Gortler, S., Cohen, M. and McMillan, L., *Unstructured Lumigraph Rendering*, Proc. of SIGGRAPH 2001 (2001), pp. 425–432.

[6] Debevec, P. E., Taylor, C. J. and Malik. J., *Modeling and Rendering Architecture from Photographs: A Hybrid Geometry- and Image-Based Approach*, Proc. of SIGGRAPH 96 (1996), pp.11–20.

[7] Chen , Q. and Medioni, G., *Image Synthesis from A Sparse Set of Views*, Proc. of IEEE Visualization (1997),pp. 269–275.

[8] Heigl, B., Koch, R., Pollefeys, M., Denzler, J. and Van Gool, L., *Plenoptic modeling and rendering from image sequences taken by a hand-held camera*, Proc. of the 7th International Conference on Computer Vision (1999), IEEE Computer Society Press, Korfu, Griechenland, pp. 585–591,

[9] McMillan, L., *An Image-Based Approach to Three-Dimensional Computer Graphics*, Ph.D. Dissertation, University of North Carolina at Chapel Hill (1997).

[10] Rusinkiewicz, S. and Levoy, M. *QSplat: A Multiresolution Point Rendering System for Large Meshes*, Proc. of SIGGRAPH 2000 (2000), pp. 343–352.

[11] Levoy, M. and Whitted, T., *The Use of Points as a Display Primitive*, Technical Report TR 85-022, Univ. of North Carolina at Chapel Hill (1985).

[12] Pfister, H., Zwicker, M., Baar, J. v. and Gross, M., *Surfels: Surface Elements as Rendering Primitives*, Proc. of SIGGRAPH 2000 (2000), pp. 335–342.

[13] Westover, L., *Interactive volume rendering*, Proc. of the Chapel Hill workshop on Volume visualization (1989), pp. 9–16.

[14] Laur, D., Hanrahan, P., *Hierarchical splatting: a progressive refinement algorithm for volume rendering*, Proc. of SIGGRAPH 91 (1991), pp. 285–288.

[15] Zwicker, M., Pfister, H., van Baar, J. and Gross, M., *Surface Splatting*, Proc. of SIGGRAPH 2001 (2001), pp. 371–378.

[16] Grossman, J. P. and Dally, W. J., *Point Sample Rendering*, Proc. of 9th Eurographics Workshop on Rendering (1998), pp. 181–192.

[17] Shade, J. W., Gortler, S. J. and Szeliski, R., *Layered Depth Images*, Proc. of SIGGRAPH 98 (1998), pp. 231–242.

[18] Gortler, S. J. and He, L-w, *Rendering Layered Depth Images*, Technical Report MSTR-TR-97-09, Microsoft Research (1997).

[19] Max, N., *Hierarchical Rendering of Trees from Precomputed multi-layer Z-buffers*, Rendering Techniques '96 (1996), Vienna: Springer-Verlag, pp. 165–174.

[20] McMillan, L. and Bishop, G., *Plenoptic Modeling*, Proc. of SIGGRAPH 95 (1995), pp. 39–46.

[21] Chen, G. Q., *Sparse Light Field*, Technical Report, Graphics Competence Center, Human-Computer Interaction Laboratory, ST Microelectronics (2001).

[22] Ng, K. C., Trivedi, M. and Ishiguro, H., *Generalized Multiple Baseline Stereo and Direct Virtual View Synthesis Using Range-Space Search, Match, and Render*, International Journal of Computer Vision 47 (2002), pp. 310–148.

[23] Ng, K. C., *Simultaneous and Consistent 3D Reconstruction of Multiple Views with Altering Search Path and Occlusion Modeling*, Proc. of the 7th International Workshop on Vision, Modeling, and Visualization (2002).

[24] Gu, X., Gortler, S. J., Cohen, M. F., *Polyhedral Geometry and the Two-Plane Parameterization*, Proc. of the 7th Eurographics Workshop on Rendering (1997), pp. 181–192, 1997.

[25] Isaksen, A., McMillan, L., Gortler, S. J., *Dynamically Reparameterized Light Field*, Proc. of SIGGRAPH 2000 (2000), pp. 297–306.

[26] Camahort, E., Lerios, A., Fussell, D., *Uniformly Sampled Light Fields*, Proc. of Eurographics Rendering Workshop (1998), pp. 117–130.

[27] Gonzales, R. C., Wintz, P., *Digital Image Processing*, Second Edition, Addison-Wesley Publishing Co., Massachusets (1987), pp 334-340.

[28] Matthews, J., *An Introduction to Edge Detection: The Sobel Edge Detector*, http://www.generation5.org/content/2002/im01.asp (2002).

[29] Ruzon, M. A., Tomasi, C., *Color Edge Detection with the Compass Operator*, IEEE Conference of Computer Vision and Pattern Recognition '99 (1999), pp 160-166.

[30] Scharstein, D., Szeliski, R., Zabih, R., *A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms*, Proc. of the IEEE Workshop on STereo and Multi-Baseline Vision (2001).

[31] Okutomi, M., Kanade, T., *A Multiple-Baseline Stereo*, IEEE Transactions on Pattern Analysis and Machine Intelligence (1993), 15(4), pp. 353-363.

[32] Schechner, Y. Y., Kiryati, N., *Depth from Defocus vs. Stereo: How Different Really are They?*, Proc. of the International Conference on Pattern Recognition (1998), pp. 1784-1786.

[33] Chang, C., Chatterjee, S., Kube, P. R., *On an Analysis of Static Occlusion in Stereo Vision*, Proc. of Computer Vision and Pattern Recognition (1991), pp. 722-723.

[34] Brown, M. Z., Burschka, D., Hager, G. D., *Advances in Computational Stereo*, IEEE Transactions on Pattern Analysis and Machine Intelligence (2003), 25(8), pp. 993-1008.

[35] Kanade, T., Okutomi, M., *A Stero Matching Algorithm with an Adaptive Window: Theory and Experiment*, IEEE Transactions on Pattern Analysis and Machine Intelligence (1994), 16(9), pp. 920-932.

[36] Kang, S. B., Szeliski, R., Chai, J., *Handling Occlusions in Dense Multi-View Stereo*, Proc. of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (2001), pp. 103-110.

[37] Seitz, S. M., Dyer, C. R., *Photorealistic Scene Reconstruction by Voxel Coloring*, Proc. of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (1997), pp. 1067-1073.

[38] Zwicker, M., Pfister, H., van Baar, J., Gross, M., *EWA Volume Splatting*, Proc. of IEEE Visualization (2001), pp. 29-36.

[39] Elbery, D. *Perspective Projection of an Ellipsoid*, Technical Report (1999), http://www.magic-software.com/Documentation/PerspectiveProjectionEllipsoid.pdf

[40] Elbery, D. *Projection*, Magic Software (1999), http://www.magic-software.com/ProjectionBody.html

[41] Grey, S., *Projecting Ellipsoids: Do You Get Ellipses?*, Math Forum (1995), http://mathforum.org/epigone/geometry-puzzles/78

[42] Smith, S. W., *The Scientist and Engineer's Guide to Digital Signal Processing*, California Technical Publishing (1997), http://www.dspguide.com/pdfbook.htm.

[43] Hackbert, P. S., *Fundamentals of Texture Mapping and Image Warping*, Master's Thesis, University of California, Berkeley (1989).

[44] Räsänen, J. *Surface Splatting: Theory, Extensions and Implementation*, Master's Thesis, Helsinki University of Technology (2002), pp. 19-44.

[45] Lyons, R. G., *Understanding Digital Signal Processing*, Prentice Hall (2001), pp. 23-128.

[46] Chang, C. F., Bishop, G. and Lastra, A., *LDI Tree: A Hierarchical Representation for Image-Based Rendering*, Proc. of SIGGRAPH 99 (1999), pp. 291–298.

[47] Chen, G. Q., Hofsetz, C., Max, N., Hong, L., Liu, Y. and McGuinness, P., *Light Field Duality: Concept and Applications*, Proc. of the ACM Virtual Reality Software and Technology 2002 (2002), pp. 9–16.

[48] Chen, G. Q., Hofsetz, C., Max, N., Hong, L., Liu, Y. and McGuinness, P., *The Use of Hyperlines in LightFields*, ST Journal of System Research (2003), vol. 0, pp. 74–83.

[49] Wood, D. N., Azuma, D. I., Aldinger, K., Curless, B., Duchamp, T., Salesin, D. H. and Stuetzle, W., *Surface Light Fields for 3-D Photograpy*, Proc. of SIGGRAPH 2000 (2000), pp. 287–296.

[50] Kang, S. B., Szeliski, R., Anandan, P., *The Geometry-Image Representations Tradeoff for Rendering*, International Conference on Image Processing (2000), Vancouver, Canada.