University of Alberta

Desktop Image-based Rendering

by

Jason Michael Selzer 

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

in

Department of Computing Science

Edmonton, Alberta
Spring 2007

Your file Votre référence
ISBN: 978-0-494-30020-6
Our file Notre référence
ISBN: 978-0-494-30020-6

Abstract

The field of image-based rendering uses photographs rather than geometric primitives as the building blocks of rendering. Past approaches have often required a prohibitive number of cameras and high capture costs with efficient rendering, or few cameras and low capture costs with expensive rendering. However, advances in processor speed and the advent of the graphics processing unit have made inexpensive real-time capture and rendering possible on desktop machines.

This thesis presents an image-based rendering system that makes use of the GPU to achieve interactive performance on a consumer-level PC. The rendering algorithm is a GPU-based backward rendering approach that utilizes depth maps computed using dynamic programming over space-filling curves. Our system is capable of dealing with any number of cameras placed in varying positions, and in the future may open up several applications to home PC users such as gaze correction for teleconferencing.

Acknowledgments

I wish to extend my sincere thanks to my supervisor Herb Yang. He deserves the greatest share of credit for his help and encouragement. Throughout the past two years he has been a constant source of ideas and inspiration, and has always been able to provide some insight into the problem at hand. I am sure I could continue working for several more years and not exhaust all the possible research avenues I have been presented with.

I must also thank my past and present labmates, Nathan Funk, Cheng Lei, Daniel Nielsen, Hai Mao, Xuejie Qin, and Danielle Sauer. Their insight and company has been very valuable. In particular, I have often had the good fortune of investigating topics that overlap with the work of Cheng Lei, and his help has been greatly appreciated. The quality of his results provided a constant goal to strive for.

I also wish to thank my friends, who have made my time in Edmonton enjoyable and have kept me sane. In particular, the companionship of Christina Carter, Doug Demyen, Steven Enns, and Danielle Sauer was invaluable in helping me settle in and get through a stressful first few months away from home, and I thank them for that.

Finally and most importantly, I wish to thank my parents. All through my life they have set an example of hard work, patience, and disposition that I strive to live up to. I could not have accomplished any of this without their example and encouragement.

Contents

1 Introduction 1

2 Background and Related Work 5
- 2.1 Stereo Matching . . . 5
  - 2.1.1 Fast Stereo Matching . . . 9
- 2.2 Image-based Rendering . . . 12
  - 2.2.1 Video-based Rendering . . . 15
- 2.3 Hardware-based Techniques . . . 18
  - 2.3.1 GPU Architecture . . . 19
  - 2.3.2 GPU-Based Stereo Matching Techniques . . . 22
  - 2.3.3 GPU-Based IBR Techniques . . . 23

3 System Overview 33
- 3.1 Space-Filling Curves for Stereo Matching . . . 35
  - 3.1.1 Space-Filling Curves . . . 35
  - 3.1.2 Dynamic Programming on Space-Filling Curves . . . 41
  - 3.1.3 Cost Matching and Aggregation . . . 43
  - 3.1.4 Random Space-Filling Curves . . . 44
  - 3.1.5 Experimental Setup . . . 45
  - 3.1.6 Temporal Filtering . . . 48
- 3.2 Image-based Rendering . . . 49
  - 3.2.1 Wide Baseline View Interpolation with Depth Maps . . . 49
  - 3.2.2 Background Model . . . 54

4 Results and Analysis 58
- 4.1 Stereo Matching Results . . . 58
- 4.2 Image-based Rendering Results . . . 66

5 Conclusions and Future Work 77
- 5.1 Conclusions . . . 77
  - 5.1.1 Stereo Matching . . . 77
  - 5.1.2 Image-based Rendering . . . 79
- 5.2 Limitations and Future Work . . . 79

Bibliography 81

A Camera Calibration 85

B Image Capture 87

C Experimental Parameters 90

List of Figures

2.1 An example of epipolar geometry with two cameras. 7
2.2 An example of epipolar geometry with two rectified cameras. 8
2.3 An illustration of the light field representation. 13
2.4 An example 2D visual hull. C1 and C2 are camera locations. The dark shaded region represents the actual object, and the light shaded region is the area occupied by the visual hull. 17
2.5 A simplified model of the programmable graphics pipeline. 19
2.6 The graphics pipeline, modified for general purpose computation. . . 21
2.7 A 2D example of searching for the proper ray intersection point using a single reference view. 26
2.8 A 2D example of the "rubber sheet" problem. 29
2.9 An example of the rubber sheet effect and fixed output. 29

3.1 An overview of the components used in our image-based rendering system. 33
3.2 The primary steps of the image capture component. 34
3.3 The primary steps of the stereo matching component. 34
3.4 The primary steps of the rendering component. 34
3.5 Hilbert curves at progressively increasing levels of resolution. 35
3.6 Space-filling curves built over the silhoutte image of a baseball player. The curves change colour when crossing object boundaries. 36
3.7 Dafner et al.'s CSFC edge-weighting scheme. 37
3.8 An example of CSFC construction via the minimum spanning tree. 37
3.9 Valid pixels that may be added to a high-resolution CSFC curve segment. The existing curve is coloured with black pixels. 38
3.10 Pixel c is being considered for attachment via pixels a and b. 38
3.11 A space-filling curve is constructed one pixel at a time. 39
3.12 On the left, a space-filling curve under construction with 3 potential ways to grow. On the right, a new node is added to the curve, invalidating the other node connected to the new pixel. 40
3.13 A hole created due to the lack of suitable edges adjacent to the pixel. On the right, neighbouring edges in red are modified to place the missing pixel in the curve. 40

3.14 CSFCs built over the silhoutte image of a baseball player. The curves change colour when crossing object boundaries. (a) Dafner et al.'s original CSFC; and (b) the new high-resolution CSFC. 41
3.15 The camera configuration used in our experiments. 45
3.16 A 2D example of the depth-based backward search algorithm. 50
3.17 A 2D example of the depth-based zero-crossing point location. . . . 51
3.18 A 2D example of the object discontinuity conditions that may produce holes in the output image. 53
3.19 Ground control points for an office scene with toy bricks in the foreground. (a) The input image from a reference camera; (b) the corresponding depth map; and (c) active ground control points based on the depth map background model. 57

4.1 Selected results for random curve-based DP with median filtering. (a,b) The source images for the Tsukuba and Teddy datasets, respectively; (c,d) disparity maps for one random curve; (e,f) disparity maps for 5 random curves; and (g,h) disparity maps for 35 random curves. 60
4.2 An example of the occlusion handling cross-check method. (a) The computed disparity map using a non-random high-resolution CSFC; (b) the same disparity map after cross-checking; and (c) the ground truth disparity. 62
4.3 Resulting disparity maps. First row: disparity map reference images. Second row: ground truths. Third row: Veksler's pixel-tree DP results [45]. Fourth Row: Gong and Yang's reliability DP results [19]. Fifth Row: Results for 35 filtered random SFCs. Sixth Row: Results for non-random high-resolution SFCs. 63
4.4 A zoomed-in section of each testing image, with the non-random curve used and a sample random curve. 64
4.5 Results for the cones dataset, with cross checking enabled. (a,b) The disparity map and erroneous pixels, respectively, for non-random high-resolution curves; and (c,d) the disparity map and erroneous pixels, respectively, for 35 random high-resolution curves. In images (b) and (d), white regions denote pixels without disparity error, black regions denote matchable erroneous pixels (the absolute disparity error is greater than 1.0), and grey regions denote occluded erroneous pixels. 65
4.6 Example IBR results. The first four rows display images coming from each of the four reference cameras, with computed depth maps. The final row is the novel view image synthesized from a camera approximately in the center of the rectangle created by the four reference cameras. 67

4.7 Depth maps computed by curve-based and reliability DP. First row: reference images. Second row: 1 random curve depth maps. Third row: 5 random curve depth maps. Fourth row: 15 random curve depth maps. Fifth Row: 25 random curve depth maps. Sixth Row: non-random hi-res CSFC depth maps. Seventh Row: Reliability DP [19] depth maps. 69
4.8 A comparison of the rendering results (left) and ground truth differences (right) of curve-based and reliability DP. First row: results for 1 random curve. Second row: results for 5 random curves. Third row: results for 15 random curves. Fourth row: results for 25 random curves. Fifth row: results for non-random hi-res CFSCs. Sixth row: results for reliability DP [19]. Seventh row: ground truth image. . . 70
4.9 A comparison of rendering results with and without median filtering. (a,b,c) median filtering enabled; and (d,e,f) median filtering disabled. 71
4.10 An example of image hole filling. 72
4.11 Examples of different camera positions and orientations. 73
4.12 Rendering results for a static scene using different depth resolutions. First row: 128 depth levels. Second row: 64 depth levels. Third row: 32 depth levels. Fourth row: 16 depth levels. Fifth row: 8 depth levels. Sixth row: ground truth image. 75
4.13 Rendering results for a non-static scene using different depth resolutions. 76

B.1 An example of the colour correction technique applied to raw input images. 88
B.2 An example of distortion correction. 88
B.3 An example of median filtering for noise removal. 89

List of Tables

4.1 The percentage of bad pixels for random curve-based DP (RSFCDP) on the Middlebury stereo datasets (error threshold of 1.0). 59
4.2 The percentage of bad pixels for various curve-based DP approaches, compared to other recent DP algorithms (error threshold of 1.0). . . 61
4.3 The percentage of bad pixels for various curve-based DP approaches, compared to other recent DP algorithms (error threshold of 0.5). . . 62
4.4 A comparison of error rates averaged over 5 consecutive frames for curve-based and reliability-based stereo matching approaches in our IBR system (see Figure 4.8 for corresponding images). 71
4.5 Frame rate and error (averaged over five consecutive frames) for the results presented in Figure 4.12. 74

Chapter 1

Introduction

The generation of photo-realistic images using computers is one of the primary pursuits of the field of computer graphics. Traditionally this problem has been addressed using synthetically created geometric primitives such as polygons, which are combined to form larger objects in a scene with some material properties and an illumination scheme that gives some visual approximation of the way light behaves in the real world. The results obtained by this process continue to grow more impressive, and are approaching the point where certain scenes may be simulated on a computer with enough accuracy to fool an unaware observer.

However, this success is not achieved without a fair amount of work. As the complexity of a scene rises the amount of rendering time required grows as well. Due to the repetitive and easily parallelized nature of this work, the problem of computational cost has often been alleviated with the use of dedicated hardware. Most notably, recent years have seen rapid proliferation of powerful and inexpensive Graphics Processing Units (GPUs), consumer level graphics processors designed for the efficient transformation, texturing, lighting and rendering of 3D triangles. Unfortunately, truly high-end photo-realistic graphics still require software-based renderers and anywhere from several minutes to several hours of computational time. In addition to this the time and skill required of graphics artists to produce photo-realistic images is often significant.

In response to these difficulties, an alternative approach to computer graphics known as image-based rendering (IBR) has gained some attention in recent years. IBR algorithms are able to synthesize novel views directly from photographs of

the scene in question. The resulting rendered images are inherently photo-realistic due to the nature of the input data. In addition, the computational cost of IBR algorithms is independent of scene complexity and is often easily within reach of even the most modest consumer-level desktop PCs.

As noted by Shum and Kang [43], image-based rendering techniques can be classified according to their position on a spectrum describing the amount of geometry information required. At the far left end of the spectrum, absolutely no geometry information is needed. As a result a large number of cameras are required to capture enough information to correctly synthesize a novel view. The sheer amount of images required means that camera hardware and raw data storage requirements are expensive and often beyond the reach of the average home user. An advantage of these methods is that rendering is typically very fast, sometimes involving no more than a simple table lookup operation for each pixel.

Towards the right end of the spectrum, more geometry information is used to generate the final rendered result. This allows us to significantly relax the requirements on the number of cameras used at the expense of requiring accurate geometry information to produce an accurate final result. The need to compute (whether implicitly or explicitly) some sort of geometric model for rendering means that the computational costs of these methods are far higher than those at the left end of the spectrum (although capture and data storage are much less expensive). On the other side of the coin, the small number of cameras required for these approaches makes them more practical for the home user. However as noted in [43], obtaining accurate depth information from images is a very difficult problem and many techniques require that depth information be pre-computed "offline" prior to rendering. Fortunately, due to recent increases in PC CPU speed and the ability of certain algorithms to take advantage of the untapped power presented by the GPU, less intensive depth recovery methods can get an approximate depth for each input image pixel in real-time. These depth values are not suitable to produce an accurate geometric model, but are accurate enough to create attractive results in the final rendered IBR image. Fast depth estimation and rendering techniques recently proposed have brought the idea of real-time IBR capture and rendering without dedicated hardware within reach. The ability to perform image-based rendering in

real-time opens up a wide array of practical applications to the home user. For example, teleconferencing over the internet typically requires the user to keep their gaze focused on a chat window on the monitor while the webcam must be placed at a position such that his or her gaze is offset and the front of the face is not recorded properly. Image-based rendering techniques can be used to correct this and produce a real face-to-face conversation.

This thesis presents an image-based rendering system that is suitable for interactive use on a desktop PC. The system is capable of dealing with any number of cameras in general positions. In certain situations as few as 2 calibrated cameras may be sufficient, but we have found in our experimental setup that 4 calibrated cameras provide the best processing time / result quality tradeoff.

The system is able to capture input images from all cameras and render novel views from an arbitrary virtual camera position at interactive frame rates. It is composed of two major components: a dynamic programming-based depth estimation component and a backward-rendering view synthesis component.

Estimating depth from input images is still a very difficult problem, and it has been shown that with current approaches high quality results demand computational resources that prevent real-time execution [38]. However, there are efficient approaches that can be used to create depth maps of high enough quality for IBR at real-time frame rates. One such popular method is scanline dynamic programming (DP) [38], which is one of the oldest (and fastest) approaches to stereo matching. Unfortunately, depth maps constructed using dynamic programming are often plagued by a "streaking effect" error, where the boundaries of foreground objects are dragged to an incorrect position on the scanline, producing streaks in the result. Different approaches have been proposed to combat the streaking effect, including explicitly modeling occlusion in the DP formulation [7] or removing suspected streaks using a reliability criterion and filling in the resulting depth map holes using extra DP passes [18]. Although they may solve the problem to a limited extent, these approaches introduce extra computation in the DP evaluation or require extra DP passes, which can hurt performance when building depth maps for several cameras. To combat the streaking problem, we introduce a novel DP algorithm which optimizes matching costs globally over a space-filling curve in a single pass. Based

on the assumption that adjacent pixels of similar intensity will have the same depth, the space-filling curve can be generated to traverse as much of an image region as possible before moving on to the next one. For real-time performance, we run multiple passes using several random space-filling curves and select the median depth at each pixel. The curve-based DP method can also be easily integrated with existing scanline-based reliability measures if desired. Our results show that curve-based DP is competitive with other recent DP approaches.

For the IBR component of the system, we use a modified version of the GPU-based backward rendering algorithm previously presented in [48]. The algorithm has been modified to work with true depth values instead of disparity, allowing us to remove the constraints placed on the position of the virtual camera in [48]. As such, the virtual camera may be moved to any position in 3D space and will provide good results assuming the input views have adequately covered the parts of the scene we wish to view. To provide better coverage of scene objects and a more accurate result, a dynamic background model is maintained to store visual information from previous frames that may be occluded in the current frame. The background model is also used to label static pixels as "ground control points" to speed up the future dynamic programming passes. We show that our IBR system is capable of producing accurate images suitable for applications in teleconferencing at interactive frame rates, in spite of a relatively large distance between reference cameras.

The remainder of this thesis is organized as follows. Relevant work in the fields of stereo matching, IBR, and general purpose computation on GPUs is discussed in Chapter 2. A system overview and discussion of the stereo matching and image-based rendering algorithms used is given in Chapter 3. Results and analysis are presented in Chapter 4. Finally, Chapter 5 concludes the thesis and discusses possible avenues for future work and improvements.

Chapter 2

Background and Related Work

As previously discussed, image-based rendering techniques are often classified according to the manner of the depth information they use during rendering (if any depth information is used at all). A popular depth representation used by many IBR methods is depth maps (or disparity maps). Because the curve-based depth map estimation algorithm is a significant component of this work, a brief summary of stereo matching algorithms is provided in Section 2.1.

Until very recently, the majority of image-based rendering research has dealt only with static scenes captured in a few photographs. For the relatively unexplored problem of IBR with dynamic scenes, the term "video-based rendering" has recently come into use [51]. The current state of both fields is briefly summarized in Section 2.2. For the sake of readability, any unique stereo matching approaches specific to a certain IBR technique are discussed as necessary in this section.

Finally, the use of the GPU has gained popularity recently as a way to achieve real-time performance for stereo matching and image-based rendering. Because our approach relies heavily on the GPU, an introduction to GPU architecture is provided in Section 2.3. Various stereo and IBR techniques that make explicit use of the GPU are also detailed in this section.

2.1 Stereo Matching

Stereo matching is one of the oldest problems in computer vision, and remains a highly active area of research even today. The goal of stereo matching is to match object surface features over two or more images acquired from different viewpoints.

Based on the observed positions of these features, corresponding depth values can be estimated. The output of stereo matching algorithms can take many forms, including point clouds and/or polygonal meshes. However, most IBR algorithms requiring depth information will use a dense depth map or disparity map, in which a unique disparity or depth value is assigned to each pixel of each input image.

Although depth and disparity are similar, they are not entirely equivalent. The concept of "depth" as it applies to dense depth maps from stereo matching is intuitively obvious: each intensity level assigned to a pixel corresponds to a depth in a real-world coordinate frame. The depth map intensity levels may be transformed to the corresponding depth values through the use of an arbitrarily defined scaling function.

On the other hand, disparity computation is usually performed entirely in image space. The more a scene point's image coordinates change from one view to another, the higher the disparity and the closer the object is.

Suppose we are given two pinhole cameras with corresponding optical centers C_1 and C_2 , as in Figure 2.1. The line connecting C_1 and C_2 is known as the *baseline*. Both cameras observe a point P in world space. The plane defined by P, C_1 and C_2 is known as the *epipolar plane*, and the intersection of this plane with the image plane of C_1 or C_2 is the *epipolar line*. If P projects to pixel p_1 on the image plane of C_1, the corresponding pixel p_2 for camera C_2 is guaranteed to lie on the epipolar line. This provides a handy way for stereo matching algorithms to constrain the correspondence search space and achieve manageable performance.

In the case where both image planes are coplanar and facing the same direction, the epipolar lines are all parallel to the baseline. Stereo matching algorithms often assume that the cameras are arranged such that the epipolar lines are aligned horizontally, constraining the correspondence search to horizontal scanlines. In real-world situations where this is difficult to achieve, images may be warped or *rectified* to satisfy this condition.

A rectified image configuration is pictured in Figure 2.2. Given object point P which projects to pixel locations (x_1, y_1) and (x_2, y_2) in images I_1 and I_2, respectively, the disparity is defined as the horizontal distance $abs(x_1 - x_2)$. This can be shown to be equivalent to

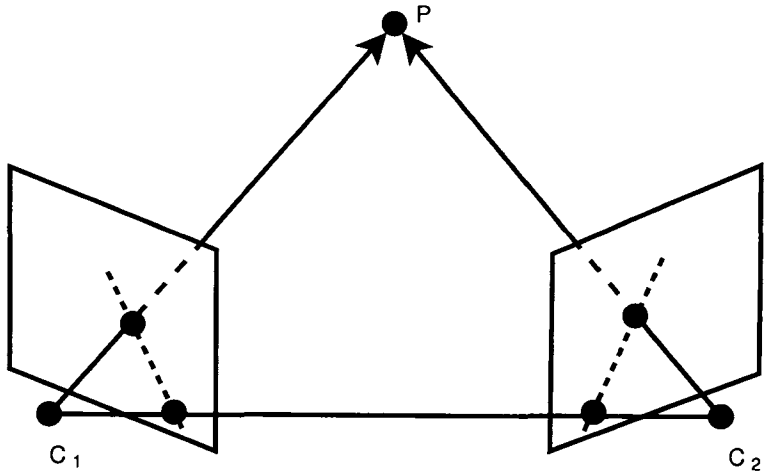


Figure 2.1: An example of epipolar geometry with two cameras.

$$disparity = d/D, \tag{2.1}$$

where d is the distance from P to the image plane and D the distance from P to the plane containing the center of projection [48]. In this sense disparity can be thought of as an "inverse depth" [38].

As noted by Scharstein and Szeliski in their excellent taxonomy of binocular stereo matching algorithms [38], most stereo algorithms perform the following four steps, or a subset thereof, listed here with what may be considered typical (but not universal) approaches:

1. *Matching cost computation* - a search proceeds along the epipolar line in a reference image, comparing pixel intensities at each disparity hypothesis with the intensity of the pixel in the target image that we wish to find a disparity for. Generally, a low intensity difference (or match cost) is likely to signify a matched scene point, assuming a lack of specular reflection.

2. *Cost aggregation* - to minimize possible mismatches due to noise and other issues, the match costs are influenced by other match costs in the neighbourhood. This may be done by applying a simple mean filter or shiftable mean filter to the match cost results.

3. *Disparity computation/optimization* - The match costs are collected and a final

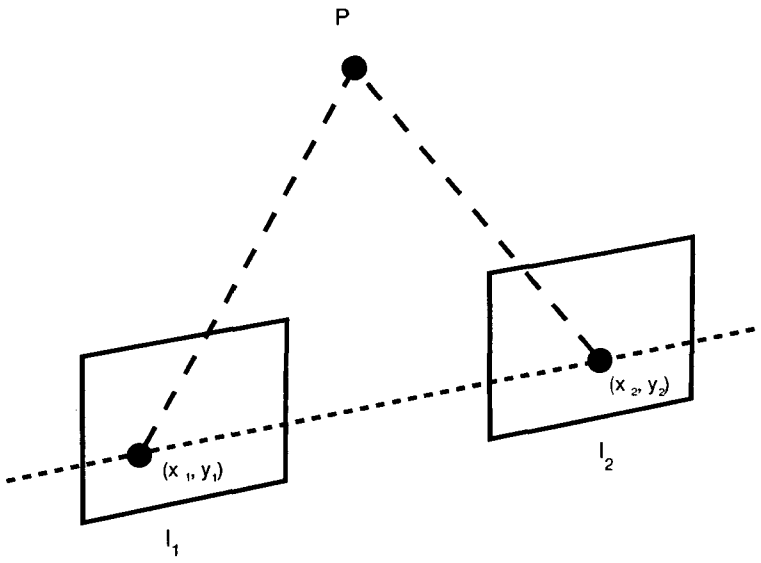


Figure 2.2: An example of epipolar geometry with two rectified cameras.

disparity is selected for each pixel.

4. *Disparity refinement* - Post-processes are applied to refine disparity values. This can include cross-checking disparities with those obtained in a neighbouring reference image [29], localizing and refining object edges [10], and any number of other refinements and enhancements.

Typically less significance is attributed to the cost computation and aggregation steps, and different approaches to these can be used interchangeably with any optimization/refinement algorithm (which is where the bulk of stereo work is focused).

To date, the cost optimization algorithms capable of producing the best quality results are usually variations of 2D optimization methods such as graph cuts [3] or belief propogation [44]. As noted in [45], these approaches seek to optimize an energy function $E(d)$ (or something similar) for a disparity map d over all pixels:

$$E(d) = E_{data}(d) + E_{smooth}(d). \tag{2.2}$$

The data term $E_{data}(d)$ measures the consistency of d with the observed data. Intensity differences obtained from the match cost computation step are one way of filling this term. The smoothness term E_{smooth} encodes a piecewise smoothness assumption, helping to ensure that disparities across object surfaces vary smoothly.

The graph cuts method is an iterative approach to minimizing Equation 2.2. At each iteration of the algorithm, a dynamically updated graph corresponding to the energy function is split using a minimum cut. Depending on the edges intersected by the cut, the pixels corresponding to graph vertices may change or swap their current disparity hypothesis. The exact definition of edge costs varies depending on the implementation. For example, Kolmogorov and Zabih [26] incorporate an occlusion cost into their edge weights for improved results. While graph cuts has been shown to be among the most accurate stereo algorithms available, performance is slow and not suitable for real-time applications.

Belief propagation is a competing 2D optimization method that also produces very good results. First applied to stereo by Sun et al. [44], belief propagation works by passing messages through a graph defined by the four-connected image pixels. Each message passed along the grid encodes the probability that the receiving node should be assigned a certain disparity based on the knowledge presenting to the sending node prior to the current time step. After a certain number of iterations, the minimum-cost belief at each node is selected as the final result. Like graph cuts, this algorithm produces exceptional results at a high performance cost. Recently Felzenszwalb and Huttenlocher have demonstrated ways to dramatically improve the performance of belief propagation [11], although the reported runtime for this approach remains around one second per image, making it unsuitable for real-time applications.

2.1.1 Fast Stereo Matching

A second class of stereo matching algorithms aim for increased performance at the expense of result quality. Generally, these approaches utilize either some variant of local winner-take-all disparity selection or dynamic programming.

The local winner-take-all approach to cost selection is undoubtedly the fastest stereo method available. This approach merely selects the disparity corresponding to the minimum matching cost at a pixel after cost aggregation. Local disparity optimization is highly susceptible to image noise, and so the resulting quality is not very good [38]. To combat this one may use a larger cost aggregation window, although this has the side effect of blurring object boundaries. The lack of inter-

pixel dependence makes local optimization methods well-suited for GPU processing, which is where the bulk of recent work in this field has been focused. A selection of recent GPU-based approaches is presented in Section 2.3.

Dynamic programming is considered a 1D optimization method [45], as it typically optimizes the energy function over a horizontal image scanline. To find the optimal disparity over a scanline, DP algorithms iterate from left to right, accumulating costs as they go. After the entire scanline has been processed and the minimum cost path through disparity space has been found, a trace-back step follows the path backwards assigning disparities. Additional constraints can also be incorporated into the DP formulation to improve results. An example of this is the ordering (monotonicity) constraint, which can be used to model occlusions by requiring that pixels in the reference image be matched in sequential order [14]. However, the ordering constraint is violated in the presence of thin foreground objects.

Because horizontal scanline DP only enforces piecewise smoothness in the horizontal direction, "streaking errors" occur. "Streaking errors" are visually jarring and a major defect of the vanilla dynamic programming approach [38]. Recent work in this field has sought to correct these errors.

For example, Gong and Yang's reliability-based dynamic programming (RDP) [16] reduces streaking errors by incorporating multiple passes, locking in areas of high reliability as "ground control points" for subsequent DP passes. The reliability of a disparity hypothesis d at pixel p is defined as the cost difference between the best path that contains $< p, d >$ and the best path that does not contain $< p, d >$. Under this definition of reliability, a confident match will be part of a clear minimum path when compared to other hypotheses. Gong and Yang run several horizontal scanline DP passes, removing points with a reliability under a certain threshold and keeping the rest as ground control points. Each iteration increases the smoothness penalty until a sufficiently dense disparity map is obtained. This approach was later improved upon in a GPU-based implementation [19] to be discussed later.

Kim et al. present a similar idea [25]. In their work, *generalized ground control points (GGCPs)* are used to guide the DP results and attain better quality. The concept of GGCPs is different from the ground control points employed by Gong and Yang. GGCPs are computed in the match cost computation stage using heuristics

to cull disparity hypotheses that may be invalid due to occlusion, specular reflection and untextured regions. It is important to note that a pixel will have several GGCPs which comprise the sole input to the dynamic programming step. Pixel disparities culled in the GGCP computation are completely removed from consideration. The initial horizontal DP pass adjusts path costs to bias the results of a second vertical DP pass which computes the final result. By incorporating a second vertical pass, Kim et al. are able to reduce the impact of scanline streaking errors, although the fact that they are not applying DP along epipolar lines means they cannot enforce the ordering constraint.

Hirschmüller's Semi-Global Matching (SGM) method presents a unique application of dynamic programming to stereo vision [23]. The SGM method uses a mutual information-based matching cost for robustness to differing illumination conditions. To approximate the appearance of a 2D global solution for pixel p using 1D dynamic programming, Hirschmüller applies DP along several lines intersecting at p (in practice this is usually 8 or 16 paths). The costs of these intersecting paths for each disparity hypothesis d are added at pixel p, and the d with the lowest summed cost is selected as the result. By applying this approach, inter-scanline consistency is enforced (as in [25] the ordering constraint cannot be used). Hirschmüller achieves results comparable with global optimization methods with much better performance (although the performance is still not as fast as some realtime DP implementations).

Recently there has been some work in reformulating DP to apply it to data structures other than a line in the image. This possibility was first investigated by Veksler [45]. Veksler constructs a minimum spanning tree covering a graph defined over the image in which edges connect pixels with their 4-neighbours. Edge weights are defined using intensity differences between neighbouring pixels and a distance transform which calculates how far inside a homogenous intensity image region a pixel is. A tree-based DP algorithm incorporating data matching costs and smoothness penalties is then applied to the constructed minimum spanning tree. Because inter-pixel consistency is enforced over multiple tree branches in several directions, "streaking" error is not a factor in the results. Veksler reports mid-range results, although the time required for tree traversal and construction makes this approach slightly slower than traditional line-based DP.

An approach motivated by Veksler's work was presented by Lei et al. [29]. Lei et al. oversegment the image into similar-intensity regions using mean shift segmentation, and then construct a tree of image segments and apply tree-based DP. The use of regions as a disparity primitive applies stricter smoothness constraints, producing results significantly better than pixel tree dynamic programming. However, the added cost of image segmentation means this approach is slower than competing DP methods and is not yet ready for real-time implementation.

Finally, Deng and Lin [9] propose a similar idea in which the image is quickly segmented into line segments and the line segments are then used as primitives for tree construction. They show this approach to improve on Veksler's result quality, while also reducing tree construction and traversal time.

2.2 Image-based Rendering

As discussed previously in Chapter 1, Shum and Kang classify IBR techniques according to the type of depth information used. Techniques may use explicit geometry information in the form of depth maps, implicit geometry in the form of point correspondences, or no geometry information at all.

Techniques that do not use any geometry information compensate by using a very dense sampling of the scene. Typically rendering is very fast, but data storage costs are high. These techniques work by sampling a subset of the plenoptic function, defined by Adelson and Bergen [1]:

$$P_7 = P(V_x, V_y, V_z, \theta, \phi, \lambda, t). \tag{2.3}$$

The plenoptic function describes the intensity of light rays passing through every possible location (V_x, V_y, V_z), at every possible angle (θ, ϕ), with every possible wavelength λ at every time t.

One of the earliest and simplest examples of IBR in this category is Chen's Quicktime VR system [6], which simplifies the plenoptic function by fixing the camera to a static position and constructing a panorama around it. Pictures taken by a rotating camera are "stitched" together, and then later warped in the Quicktime VR player to produce the effect of the user pointing the virtual camera at a certain

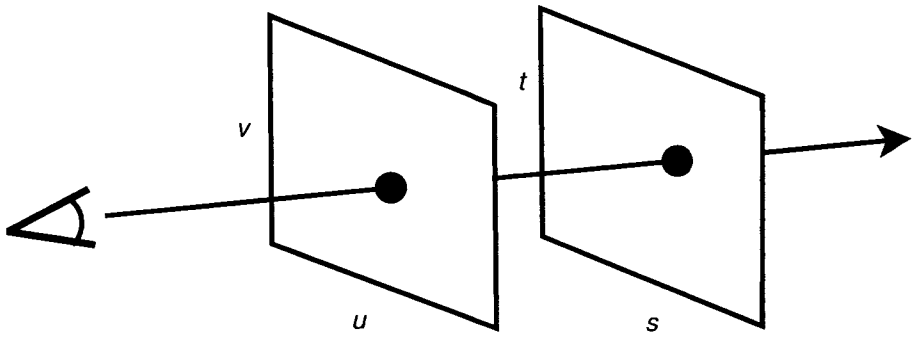


Figure 2.3: An illustration of the light field representation.

direction in the panorama.

Perhaps the most famous techniques in this area are the light field and lumigraph approaches, similar approaches which both use a 4-dimensional parameterization. In this case, the plenoptic function is simplified to

$$P_4 = P(u, v, s, t), \tag{2.4}$$

where (u, v) and (s, t) describe a ray's intersection points in two arbitrarily placed parallel planes that define a bounding box for the object to be sampled as shown in Figure 2.3. Display is handled by projecting rays from the virtual camera through the planes and using a simple table lookup/blending operation to find the intensity. However, these techniques require a very dense sampling, which can capture/storage very impractical. The lumigraph approach can optionally use geometric information to modify which reference rays are sampled so that the sampled rays intersect the object more densely around the desired scene point, improving the quality of results. However, most surveys of image-based rendering consider the lumigraph technique a close sibling of the light field approach, and so we include it here instead of in the geometry-based approaches section.

In 1999, a novel 3-dimensional plenoptic function approach known as concentric mosaics was introduced by Shum and He [42]. A capture rig consisting of a single camera placed on the end of a rotating beam was constructed, and a simplified plenoptic function used three variables to describe the captured rays: rotation angle, radius of camera rotation axis, and vertical elevation. The system allows the user to move around anywhere inside the "capturing circle," sampling rays to produce

the appropriate image.

Finally, a unique 5-dimensional parameterization of the plenoptic function was proposed by McMillan and Bishop [33]. They remove the time and wavelength variables from Equation 2.3 to get the following representation:

$$P_5 = P(V_x, V_y, V_z, \theta, \phi). \tag{2.5}$$

The authors achieve this sampling by using several cylindrical panoramas as reference images, along with disparity maps relating each pair. Novel views are constructed by warping sampled images using the disparity information. It should be noted that this method differs from other stated plenoptic sampling methods in that geometry information is used to render novel views.

Techniques based on implicit geometry information usually rely on point correspondences or epipolar constraints to construct new views. An example of these techniques is the view morphing method introduced by Seitz and Dyer [40]. Their technique is a 3-step process using a pair of input images. In the first step, a projective transformation is computed for each input image using point correspondences such that both images are parallel and horizontally aligned. The two images are interpolated using a shape-preserving morph, and the final image is "postwarped" to the desired position and orientation of the new view. As noted by the authors, this method is very sensitive to changes in visibility, so the reference images must be fairly close to provide good results.

The final category of IBR techniques uses explicit geometry information. This includes depth and disparity maps. Although the problem of acquiring accurate geometry from photos is very difficult and by no means solved, this method of doing IBR has become very popular. The relatively sparse sampling and low storage costs means real-world applications are more practical and affordable than competing methods. Due to this, many real-world IBR implementations are based on this approach.

A common technique when given per-pixel depth information is to apply 3D warping [32]. The basic idea is to project pixels to their estimated 3D locations using depth information and then project these points to the novel view. Depending on the sampling of the scene and the resolution of the images, many holes may be

present in the image. This issue is often solved by "splatting" the reference pixels into the novel view, thus increasing their coverage of the image.

Oliveira et al.'s work on relief textures [36] combines warping with texture mapping to provide a fast IBR solution. In their approach, textures with per-pixel depth information are warped and then applied to polygons using conventional texture mapping. Their method can be used to provide the effect of surface detail and motion parallax. If relief textures are applied to a simple bounding box, entire objects can be rendered accurately. Unfortunately, as a single-layer representation, the relief texture is not equipped to deal with situations where multiple surface layers are required to model an object. In these cases, the authors recommend approaches such as Shade et al.'s layered depth images [41].

The layered depth image approach models a scene as a collection of surfaces at different depths. Each "pixel" in the representation stores a list of intensity values and depths at multiple locations along the corresponding ray. Novel views are rendered by warping pixels in back-to-front order and splatting them to the novel image.

2.2.1 Video-based Rendering

A less thoroughly examined sibling of image-based rendering is the field of video-based rendering. Instead of the static images used by image-based rendering as input, video-based rendering algorithms use video footage typically taken from static camera positions. Aside from this difference, the line between image-based rendering and video-based rendering techniques is not very well defined. It is not yet clear how to fully utilize temporal information to aid in video-based rendering. Many video-based rendering approaches do not consider any temporal information at all, instead treating each video frame as an individual photo upon which standard static-scene IBR algorithms can be applied. In this sense the term video-based rendering may be a bit of a misnomer. Perhaps algorithms such as these can be considered video-based rendering algorithms by virtue of the fact that they run fast enough to render novel view video in real-time.

Most VBR techniques can be considered members of one of two categories: those that require depth information to be computed off-line prior to new view synthesis

(off-line techniques), and those that are able to run on streaming video using fast stereo techniques or no depth information at all (real-time techniques).

Perhaps the first and best known example of an off-line technique is CMU's "Virtualized Reality" project, headed by Takeo Kanade [37]. The originally published system consisted of 51 calibrated cameras mounted on a geodesic dome. A multi-baseline stereo algorithm is used to compute dense depth maps for each reference view, which are projected into 3D space to build a unique triangle mesh or "visible surface model" (VSM) corresponding to the visible scene for each of the reference cameras. Novel views are constructed by transforming the textured VSM of the nearest reference camera to the novel view and then transforming nearby VSMs from other cameras to fill the holes. A second method fuses all VSMs to construct a "complete surface model," which is a complete 3D model of the scene that may be viewed from any angle. Unfortunately, as noted by Zitnick et al. [51], these early results are of low resolution and prone to error at object boundaries. Zitnick et al.'s similar GPU-based system (which segments and deals with object boundaries separately from the main scene) is discussed in Section 2.3.

A similar mesh-based approach is the image-based visual hulls method introduced by Matusik et al. [31]. Figure 2.4 illustrates the basic visual hull concept. Silhouettes of a single object against a static background are extracted from input photos. The positions of cameras and image planes are reconstructed in a virtual 3D scene, and "silhouette cones" originating at camera positions and defined by the edges of the silhouettes are projected into the scene. The intersection of these cones, known as the "visual hull", is guaranteed to contain the original object and can be used to build a 3D mesh approximating the shape of that object. To enhance performance and image quality, Matusik et al. perform all visual hull computations in "image space" by projecting rays from the target camera and computing their intersections with the silhouette cones. No explicit 3D model is produced and computation is performed only for visible pixels. An experimental setup consisting of four calibrated cameras and five computers is used. Four computers perform distortion correction and silhouette extraction, and a fifth quad-processor 550 MHz PC performs visual hull construction. The authors report that image construction runs at about 8 frames per second depending on the number of object pixels in the visual

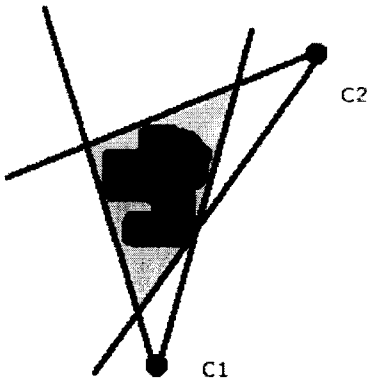


Figure 2.4: An example 2D visual hull. C1 and C2 are camera locations. The dark shaded region represents the actual object, and the light shaded region is the area occupied by the visual hull.

hull.

Schirmacher et al. present a method which uses a generalized Lumigraph structure coupled with per-pixel depth values obtained via stereo matching or real-world depth sensor equipment [39]. Their "Lumi-Shelf" experimental setup involves six firewire cameras arranged on a bookshelf in two rows of three. At each video frame, pixels from the reference cameras are projected to the virtual lumigraph image plane according to the per-pixel depth values and the user's viewing position. In experiments, the lumigraph approach is able to display novel views at 1-2 frames per second using dual-Pentium III 800 MHz PCs. However, reconstruction quality suffers greatly due to errors in the stereo matching used for depth reconstruction.

Criminisi et al. propose a simple system designed for use in one-to-one teleconferencing [7]. Using a pair of horizontally placed cameras, a unique horizontal scanline dynamic programming-based stereo algorithm is used to generate a single disparity map for image rendering. The disparity map corresponds to a "virtual image plane" incorporating both reference views. The DP algorithm used allows pixels to be labeled as matched, occluded in the left image, occluded in the right image, or as slanted surfaces (in which case a unique match in the other image would not be found). Inter-scanline consistency is promoted by applying a large Gaussian filter to match costs prior to the DP step. Novel views are generated simply by projecting this "virtual disparity surface" into the virtual camera. The authors are able to

compute a frame every two seconds for 320x240 images on a 2.8 GHz Pentium IV. The results demonstrated are of good quality, but the baseline used is relatively narrow and the nature of the IBR algorithm constrains the camera position to a location between the reference views (although it can move forward and backward).

Recently some efforts have been made to examine how temporal information can affect the rendering of dynamic scenes. For example, Wang and Yang apply the light field rendering approach to unsynchronized video sequences [46]. They note that typical body motions at sitting distance captured at 30 frames per second can be offset by as many as 10 pixels from the expected location when using unsynchronized cameras. The result is a blurry and inaccurate rendered image. To correct this, they establish a feature point correspondence among all the reference cameras and then connect these points in each image to form feature edges. Image morphing is then used to morph the reference images, match up feature points and edges, and synchronize them in software. Following synchronization, traditional light field techniques are used to render the frame. The system shows promise in correcting the rendering result, but requires a few minutes to correct each individual frame. In addition, the authors note that the temporal correction scheme used only works for motion that is roughly linear in projective space. Rotating objects can be handled in spite of this if the capture rate is fast enough, but if an object is rotating quickly feature matching and temporal correction will fail.

2.3 Hardware-based Techniques

In recent years, the success of the 3D computer game market has driven a startling growth in the performance and availability of GPUs. Beginning with relatively simple fixed-functionality pipelines designed solely for the transforming and lighting of triangles in hardware, GPUs have steadily evolved and added programmability and features with each successive generation. While their prime functionality and utilization is still steeped in the demands and terminology of 3D graphics, it is now possible for well-suited general purpose algorithms to use the GPU as a powerful parallel processor. As such, a large amount of research in the past few years has focused on general-purpose computation using the GPU (often abbreviated as GPGPU).

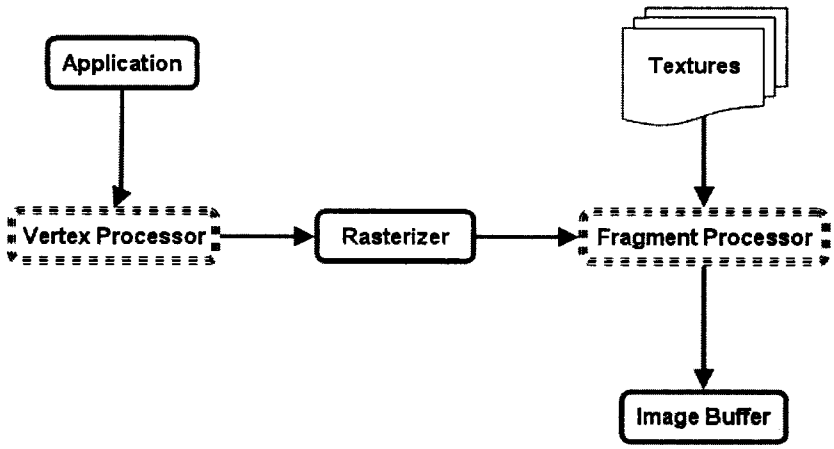


Figure 2.5: A simplified model of the programmable graphics pipeline.

2.3.1 GPU Architecture

To understand how the GPU may be harnessed for purposes other than computer graphics, it is first necessary to have some basic knowledge of how the programmable pipeline works.

The primary function of the GPU is to take a geometrical description of a 3D scene (usually represented as an array of triangle vertices) and transform it to a 2D image composed of an array of coloured pixels. The graphics pipeline responsible for these operations is illustrated in Figure 2.5. A simple explanation of the main components of the rendering pipeline is given below.

- **Application:** The application stage provides high level control from the CPU, sending 3D geometry data in the form of vertex coordinates and specifying what special operations to perform. These operations may either be built into the hardware (for example, alpha-blending or a depth test to reject invisible pixels) or involve the use of a programmer-defined *shader program*, which is also transferred to the GPU.

- **Vertex Processor:** The vertex processor is the first of two programmable processing engines in modern GPUs. The vertex processor is primarily responsible for transforming the 3D triangle vertices to 2D coordinates existing on

an image plane as specified by a given camera model. In addition, the vertex processor may also add lighting/colour information to the vertices and texture coordinate information.

- **Rasterizer:** The rasterizer takes these transformed vertices and "fills in the gaps" between them. *Fragments* are created at each pixel position on the image plane. Fragments may be considered embryonic pixels, containing information such as depth, surface normal direction, and preliminary lighting/colour information. These attributes are interpolated using the vertex attributes provided by the vertex processor. The rasterizer is not programmable.

- **Fragment Shader:** The fragment shader is a programmable unit that takes fragment information from the rasterizer and produces final coloured image pixels. In this stage textures may be applied, as well as more complex per-pixel shading techniques. The final output of this stage is the 2D image buffer, which is usually displayed on screen.

When considering how this architecture may be used for general-purpose computation, it is helpful to regard the GPU as a stream processor [5]. In the stream processing model, an input stream of like data elements is processed by a simple computational kernel and sent into an output stream. Since the computation performed on each data element is identical, the stream elements may be easily processed in parallel.

Figure 2.6 illustrates how the GPU architecture fits into the stream programming model. Ignoring work done in prior stages of the pipeline, we can consider the textures stored on the GPU as input streams. Fragment shading programs are analogous to computational kernels. After fragment shaders manipulate a texture element the result is placed in the image buffer, which represents an output stream. In many GPGPU circumstances, it is desirable to output results directly to textures stored on the video card for fast iteration on data elements. Most current GPUs support this feature, but at the time of writing drivers do not support using this feature in conjunction with other valuable features such as the early z-kill [22].

A typical GPGPU application working under a graphics API like OpenGL or DirectX uses a few basic steps to simulate this model [22]. First, a screen-sized

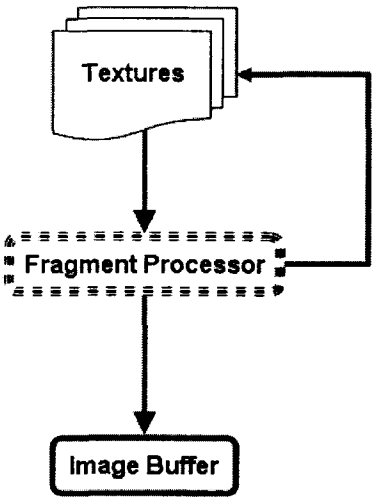


Figure 2.6: The graphics pipeline, modified for general purpose computation.

quadrilateral is drawn using orthographic projection. Input data is placed in a texture with the same resolution as the image buffer. This ensures that there is a 1:1 mapping between texture pixels (*texels*) and image pixels in the fragment shader (it should be noted that it is also now possible to have random access to texture elements inside fragment programs). The fragment shader is then forced to run over each fragment, performing computations and writing results to the image buffer. If the same data is to be used in future rendering passes, it is often best to render directly to textures.

Incorporating the vertex shader into GPGPU solutions is a more challenging problem due to the inescapable operations performed by the rasterizer. As such, vertex shader use varies depending upon the problem being tackled. Although well-suited algorithms that use the fragment processing units exclusively can typically achieve speeds 5X faster than competitive CPU implementations, Larsen has observed that a speed increase of between 10X and 100X is possible when algorithms take advantage the vertex processor, rasterizer, and other graphics-specific GPU components [27]. This effectively divides GPGPU algorithms into 2 categories: those that approach the GPU from the simple perspective of a fragment shader-based stream processor, and those that utilize the other graphics hardware features

for a significant speed increase.

2.3.2 GPU-Based Stereo Matching Techniques

Due to the parallel nature of many existing algorithms and the use of images as a primary data source, computer vision problems are often a natural fit for GPU processing. Several methods of stereo matching on the GPU have been investigated in recent years, the most notable of which are discussed here.

The stereo matching algorithms best suited for GPU processing are the local optimization methods, and recently the GPU has been used to achieve real-time local optimization. Yang et al. propose a local optimization algorithm that uses the GPU exclusively for all computation [49]. First, a matching cost computation is performed by comparing pixels in a fragment shader program. Cost aggregation is then performed in two rendering passes. The first rendering pass uses the GPU's built-in bilinear texture interpolation capability to mimic the effect of summing over a 4x4 support window with only 4 texture fetch operations. The second pass takes the results of the first pass and adds the local support window and two best neighbouring window results to each matching cost. In this way Yang et al. build adaptive window shapes out of three 4-connected square windows to best deal with features such as object edges and corners. The final winner-take-all disparity selection uses the GPU's depth test capability to automatically discard non-optimal disparities.

A similar approach is used by Woetzel and Koch [47]. Their method is very similar to the one employed by Yang et al. The main difference is that they allow for an arbitrary configuration of up to eight cameras. The extra cameras allow them to use only the best aggregated scores for final local optimization, discarding cameras with a high match cost due to occlusion or wide baselines distorting regions underneath the aggregation window.

Gong and Yang [15] later apply an even more sophisticated approach to cost aggregation to achieve improved results at real-time speeds. Prior to cost aggregation, colour discontinuity boundaries are detected on the GPU by applying an edge enhancement filter and then locating the local maximums of intensity gradients in the vertical and horizontal direction. This edge information is used to guide the cost aggregation step. In a trick common among many GPU implementations, a

5x5 box filter is implemented in two steps with a 5x1 horizontal filtering pass and a 1x5 vertical filtering pass. Different weights are applied to corresponding pixels of the filter based on the location of edges. For example, if no edges are detected the filter operates as a basic mean filter. If a colour boundary exists on a certain side of the filter, only matching costs in the center and opposite side are considered. If edges are detected on both sides, the authors assume they are dealing with an over-segmented textured surface and the filter operates as a mean filter. Disparity optimization is performed using either a winner-take-all optimization on the GPU or dynamic programming scanline optimization on the CPU. Both approaches demonstrate results significantly better than [49].

Finally, Gong and Yang [19] introduce a dynamic programming-based algorithm accomplished entirely on the GPU. A 3x3 shiftable mean filter is used on the GPU for cost aggregation. For the DP cost optimization, the reliability threshold previously introduced in [16] is used in a three-pass framework. The first pass performs DP on horizontal scanlines using the reliability threshold. The second pass performs reliability-based DP on vertical scanlines using the results of the first pass as ground control points. Remaining holes are filled with a final horizontal pass with reliability calculation disabled. By considering each scanline in parallel with the fragment shaders, Gong and Yang traverse several scanlines simultaneously with a new rendering pass for each pixel in sequence. However, due to the relatively small amount of work that can be done in a single rendering pass and the fact that the CPU can perform DP while the GPU runs match cost calculation simultaneously, Gong and Yang find that transferring match costs back to the CPU for DP is still 2-3 times faster than the GPU-exclusive approach [19]. They later report that optimizations to the algorithm and increasing GPU performance effectively allow the GPU-exclusive method to surpass the performance of the GPU/CPU hybrid method [20].

2.3.3 GPU-Based IBR Techniques

The power of the GPU makes it particularly attractive to those who wish to attain real-time performance with their image-based rendering algorithms.

Li et al. are able to use GPUs to reduce the problem of computing visual hulls

to a simple automatic image space computation [30], employing a particularly novel solution that goes beyond fragment shaders. Visual hull construction and rendering is performed simultaneously in the GPU. Object silhouettes for each camera are loaded as projective RGBA textures with an alpha value of 1 inside the object and 0 outside. To determine the intersection of silhouette cones, the silhouette cone for a camera Ci is rendered as a polygon mesh. Then the textures for all remaining silhouettes are projected from their respective cameras onto the cone mesh. Register combiners (an ancestor of current fragment shaders) are then used to remove pixels on the mesh that do not intersect any other silhouette cones (where the textured alpha value is 0). The final textured colour of pixels on the visual hull is computed by blending colour information from silhouettes inside the register combiner program. The process of rendering a silhouette cone and finding intersections is repeated for all cameras, resulting in a final textured visual hull. The authors report a significant speed-up compared to CPU-based techniques, achieving 84 frames per second using 4 input silhouette images with a resolution of 320x240. Although capture is performed using 4 networked client PCs, reconstruction and rendering is performed on a single GeForce 3-equipped server.

A simple plane-sweeping hardware-based IBR algorithm [50] is proposed by Yang et al. The space in front of the novel view image plane is discretized into depth planes. The GPU steps through the planes from near to far, performing two rendering passes at each plane iteration. The first pass projects all input textures acquired from calibrated reference cameras to the current depth plane and computes a mean colour and sum-of-squared-difference (SSD) at each pixel. The result is stored in a single texture, with the SSD in the alpha channel. The second pass compares the recently obtained means and SSD scores to the current image result from the last depth iteration. If the SSD for a fragment is less than the previous best, that fragment is inserted into the resulting image. After all depth planes have been considered, the sorted result is used as the output image. Yang et al. achieve real-time performance on a Geforce3, with three client PCs performing capture and distortion correction before passing data to the rendering server.

Li et al. present a hardware-based photo hull approach that effectively combines and extends the previous two algorithms. Photo hulls are visual hulls that use color

information to ensure a more accurate reconstruction. In this case, the visual hull technique discussed in [30] is used as a pre-process to compute a bounding volume for a target image plane-sweeping algorithm similar to [50]. Unlike Yang et al., Li et al. consider object visibility during reconstruction to avoid errors produced by occlusions in one or more reference cameras. Geometric information obtained from the visual hull is used in conjunction with implicit geometry obtained during the rendering photo-consistency checks to update visibility maps for each camera. The visibility maps are then used to exclude fragments from foreground object parts that may erroneously invalidate colour-consistency checks for other depths.

Many recent GPU-based IBR algorithms have also exploited depth information in the form of disparity maps. In most cases, these disparity maps are computed offline ahead of time, and rendering is performed in realtime on the GPU. Goldlücke et al. present a simple IBR method in which disparity maps are converted to triangle meshes with disparity information encoded at each vertex [13]. Meshes for each reference camera are warped to the target camera according to the disparity of each vertex, and the resulting colours are blended together for the final result. The resolution of the mesh may be adjusted to provide the best tradeoff in terms of performance and rendering quality. However as noted by Zitnick et al. these meshes cannot model object discontinuities [51].

A more recent work by Zitnick et al. uses a similar approach to rendering [51]. In this case, a number of high-resolution cameras are placed in a linear configuration with disparity maps computed offline. In addition, the boundaries of objects in the scene are segmented and depth is computed for these prior to rendering. To generate a novel view, the two nearest reference cameras in the data set are selected. The main depth map for each view is converted to a 3D mesh which is warped to the target view. A second pass is run to remove triangles corresponding to depth discontinuities, and the pre-computed boundary layer is used to fill in the resulting gaps. The results from the two reference cameras are blended together in a final fragment shader pass. The authors show that this technique is capable of running in real-time on an ATI 9800 PRO, although real-world performance is slower due to the cost of reading video frames for dynamic scenes from the hard disk. In addition, the authors only demonstrate results for target cameras directly between the two

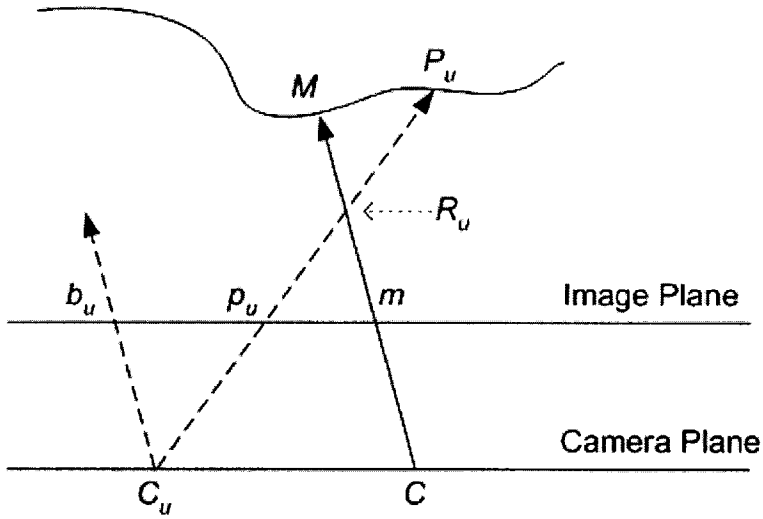


Figure 2.7: A 2D example of searching for the proper ray intersection point using a single reference view.

reference views.

Hardware-Based Backward Rendering

Gong and Yang propose a backward rendering approach to image based rendering [17] known as disparity-matching based view interpolation. Because the IBR algorithm used in this work is based on a hardware variant of this, we discuss the method in detail here along with the GPU implementation and enhancements incorporated by Xu [48].

An array of reference images is acquired using a grid of cameras mounted on a planar surface (a *camera field*). The image plane is identical for all cameras, and parallel to the camera plane. Given these input images, disparity maps are computed offline prior to rendering. For the sake of future discussion, the disparity used here is defined as:

$$\delta(p_u) = \frac{|C_u p_u|}{|C_u P_u|}, \tag{2.6}$$

where p_u is a reference image pixel, C_u is the reference image's center of projection, and P_u is the point of intersection between the ray $C_u p_u$ and the object in the scene. In [48] and henceforth in this thesis, the preceding definition of reference image disparity is referred to as the *estimated disparity.*

Given the disparity information for several reference images, we can synthesize

a novel view. A 2D illustration of the technique is shown in Figure 2.7. In the illustration, C_u is the center of projection of a given reference view, and C is the center of projection of the novel view. The rays Cm and $C_u b_u$ are parallel.

Assume we are trying to find the intensity of pixel m in the novel view. To do this, we must find the physical point M where the ray Cm intersects the closest object in the scene. We can indirectly detect this using rays from the reference views. The reference view ray $C_u p_u$ intersects the target novel view ray at point R_u. Colouring pixel m then reduces to the problem of finding where the target ray intersection point R_u and the reference view ray-object intersection point P_u are the same. In such a case, the proper colour will be in the reference view at pixel p_u.

For a given ray $C_u p_u$ in a reference view, we know the estimated disparity value $\delta(p_u)$ is equal to the ratio $\frac{|C_u p_u|}{|C_u P_u|}$. Based on this, an equation for the length of $C_u P_u$ can be derived [17]:

$$\delta(p_u) = \frac{|C_u p_u|}{|C_u P_u|} \Longrightarrow |C_u P_u| = |C_u p_u| \times \frac{1}{\delta(p_u)} \tag{2.7}$$

In addition, the length of $C_u R_u$ can be defined by Equation 2.8:

$$\frac{|C_u p_u|}{|C_u R_u|} = \frac{|b_u p_u|}{|C_u C|} \Longrightarrow |C_u R_u| = |C_u p_u| \times \frac{|C_u C|}{|b_u p_u|} \tag{2.8}$$

Finding the intensity of the physical point M requires finding the projection of M in a reference image. As described above, this is the reference image pixel p_u where the length of $C_u P_u$ equals the length of $C_u R_u$. In other words, the following equation should evaluate to zero [17]:

$$F(p_u) = \delta(p_u) - \frac{|b_u p_u|}{|C_u C|} \tag{2.9}$$

In [48] and throughout the rest of this thesis, $\delta_{observed} = \frac{|b_u p_u|}{|C_u C|}$ is referred to as the *observed disparity.*

The point where $F(p_u)$ equates to zero is known as the *zero-crossing point* [48]. By searching the reference image from point b_u to m along the epipolar line, the zero crossing point can be found and the novel view pixel coloured correctly. Of course, since the values of $F(p_u)$ along the epipolar line are a discrete sampling and not continuous, we can only detect where the value of $F(p_u)$ changes from $F(p_u) < 0$ to $F(p_u) > 0$ (or in other words, $F(x_i) \times F(x_{i+1}) < 0$ where x_i and x_{i+1}

are adjacent pixels on the search segment). After locating the zero-crossing point between two reference image pixels, the final output pixel colour is calculated by linearly interpolating the reference pixel colours. In the final implemented system, the four nearest reference views are searched in order from nearest to farthest for a zero-crossing point.

The farther away the reference image, the longer the corresponding epipolar line segment $b_u m$ is, resulting in an increase in search time. Xu introduces enhancements to decrease this search time [48].

First, it is noted that points b_u in the reference image and m in the target image have the same pixel coordinates, since Cm and $C_u b_u$ are parallel rays. Also, each pixel's epipolar line is parallel to $C_u C$ and the length of the search line segment mb_u is equal to $|C_u C|$. Because of the similarities in epipolar lines for each image pixel, the epipolar line search segment only needs to be computed once for a given reference view and target view combination. The line segment can be stored as an array of offset vectors instead of explicit coordinates, and then each epipolar line segment in a reference image may be traversed using the same offset array.

Similarly, it is noted the observed disparity value $\frac{|b_u p_u|}{|C_u C|}$ for a position along the epipolar line segment is merely a fraction of the length of that line segment. Since the observed disparity value is constant for a reference image's given offset vectors, they can also be pre-computed and applied to all searches in the corresponding reference image. Because the epipolar line segment and the observed disparity are only computed once per reference image (instead of once for every pixel), there is a significant performance increase.

Interpolating a final target pixel colour from two reference pixels creates a stretching effect along boundaries of foreground objects. This is known as the "rubber sheet" problem [17]. Figure 2.8 demonstrates this problem (a similar figure is shown in [48]). In the figure, x_i and x_{i+1} are two adjacent pixels along the search segment of a reference image with center of projection C_u. The camera ray $C_u x_i$ intersects a foreground object at point X_i, and the ray $C_u x_{i+1}$ intersects a background object at point X_{i+1}. Because $F(x_i) > 0$ and $F(x_{i+1}) < 0$, a zero-crossing point is detected and the target image pixel is filled with an incorrect colour interpolated between two different objects. The resulting effect is shown in Figure 2.9.

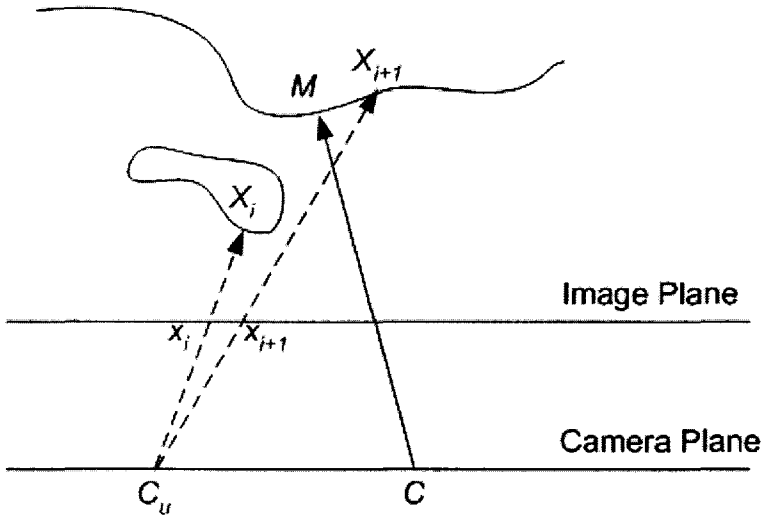
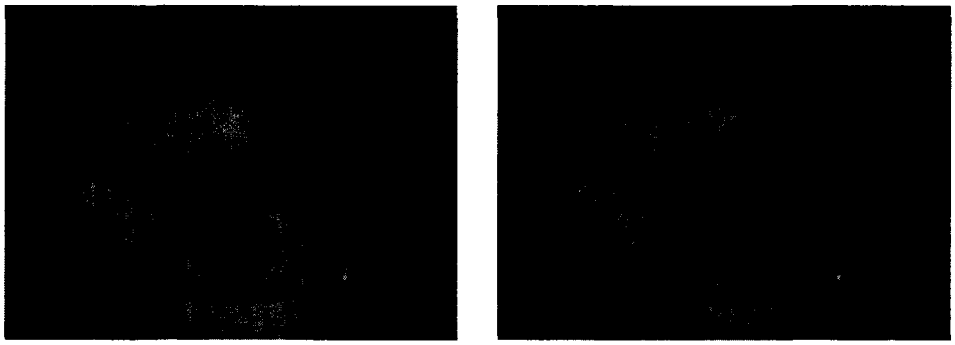


Figure 2.8: A 2D example of the "rubber sheet" problem.



(a) Synthesized image with no zero-crossing threshold

(b) The same image with a zero-crossing threshold applied

Figure 2.9: An example of the rubber sheet effect and fixed output.

To combat this problem, Gong and Yang use a fixed threshold t to discard false zero-crossing points [17]. If $F(x_i) \times F(x_{i+1}) < 0$ then a zero-crossing point is detected. However, it is only used if $|F(x_i) - F(x_{i+1})| < t$. If the difference between $F(x_i)$ and $F(x_{i+1})$ is greater than the threshold, then X_i and X_{i+1} are assumed to be points on two separate objects.

As noted by Xu [48], the fixed-threshold method fails when a reference view is very close to the target view. In that case, a small value for $|C_u C|$ means that $|F(x_i)|$ and $|F(x_{i+1})|$ become very large. As such, the value of $|F(x_i) - F(x_{i+1})|$ may end up being greater than the threshold even if x_i and x_{i+1} are projections of points on the same surface, generating false discards. Xu addresses this problem by

using an adaptive threshold [48]:

$$t_{adaptive} = \frac{t}{|C_u C|} \tag{2.10}$$

In this case, as the views get closer together and the value of $|C_u C|$ decreases, the threshold will grow to compensate.

If a suitable zero-crossing point for a given pixel is not found in any reference view, then that pixel will be left as a hole in the final image. Xu uses a simple heuristic method to fill holes in the target image.

Xu makes the assumption that holes correspond to occluded background regions. When the zero-crossing detection fails because the value of $|F(x_i) - F(x_{i+1})|$ exceeds the adaptive threshold, the color of the pixel with the lower disparity is saved for later hole-filling. If a search of a later reference image should find a proper zero-crossing point and fill the pixel, then there is no need for the hole-filling information. However, if the search fails for all reference images then the saved "background" colour is used as a last resort for filling the hole.

The final algorithm for the method is presented in Algorithm 2.3.3 as pseudo-code.

The GPU-based implementation of this algorithm can be conceptualized as a plane-sweeping algorithm from near to far. Like the software algorithm, the pixel offset and estimated disparity arrays are computed ahead of time. Reference views are stored as RGBA textures with the disparity map in the alpha channel. A rectangle is drawn on the screen using orthogonal projection, and a fragment program compares observed disparities to estimated disparities read from the texture units using coordinate offsets passed as parameters. The zero-crossing search process is split into multiple passes. Each pass considers a different offset/estimated disparity pair, moving sequentially from foreground to background through the pre-computed arrays. If a zero-crossing point is found, the observed colours are blended and written to the framebuffer. Otherwise the fragment is rejected. Each reference image is handled separately in order of distance from the novel view. To prevent previously drawn foreground pixels from being overwritten in subsequent passes, a depth test is used to cancel processing on fragments that have already been drawn at a closer depth.

Algorithm 1 The final algorithm for disparity-matching based view interpolation.

```
for reference image i = 0 to 3 do
  Compute offset array offsetArr[i] and disparity array dispArr[i]
  Set maxIndex[i] to maximum index of offsetArr[i] and dispArr[i]
end for
for pixelIndex = 0 to total number of pixels do
  colourFound = false
  for reference image i = 0 to 3 do
    /* find point where the search begins */
    point = pixelIndex + offsetArr[i][0]
    /* compute first value of 'F' function */
    newF = δ(point)-dispArr[i][0]
    /* traverse epipolar line, searching for zero-crossing point */
    for index = 1 to maxIndex[i] do
      /* save the old search point */
      oldPoint = point
      oldF = newF
      /* move to next point on search segment */
      point = pixelIndex + offsetArr[i][index]
      newF = δ(point) - dispArr[i][index]
      /* if we have found a zero crossing point */
      if oldF × newF ≤ 0 then
        if |newF − oldF| < threshold then
          Write the weighted average of pixel point and oldPoint to pixelIndex
          colourFound = true
          /* continue to next target pixel */
          break
        else
          if δ(point) < δ(oldPoint) then
            /* a hole has been found, save the "background" colour */
            Set holeColour to colour at point
          else
            Set holeColour to colour at oldPoint
          end if
        end if
      end if
    end for
  end for
  /* if there was a hole left in the image */
  if !colourFound then
    Write holeColour to pixelIndex
  end if
end for
```

To fill holes, a second round of rendering passes is run after the initial round is complete. This works in the same manner as the previous zero-crossing search, but the rubber-sheet check is eliminated to prevent zero-crossing points from being culled. When a zero-crossing is detected, the colour of the pixel corresponding to the background is output by the shader. The computational costs of these hole-filling passes is very small, as an early z-kill operation is used to prevent processing where fragments have already been drawn in the initial IBR rendering passes.

Chapter 3

System Overview

As discussed in Chapter 1, the image-based rendering system presented in this thesis consists of two main pieces: a stereo matching component and an image-based rendering component. In this chapter, we first present a brief overview of the system and how its parts work together. Further details of each step in the process are then discussed in the component-specific Sections 3.1 and 3.2.

As shown in Figure 3.1, the data acquired at each frame is handled sequentially by three separate system components. First, image capture is performed and some preprocessing is done to prepare images for stereo matching. After this, the stereo matching component builds a depth map for each input image using a combination of GPU and CPU processing. Finally, these depth maps are used in conjunction with a GPU-based rendering algorithm to display a novel view.

The process followed by the capture stage is illustrated in Figure 3.2. Images from the current time frame are captured for all input cameras. Due to the differences in the recorded colour intensities among the input cameras, a quick and simple colour correction operation is performed so that the colours of objects recorded by different cameras match as they should. Following this, we correct lens distortion in the input images using the GPU and distortion coefficients calculated during camera

Figure 3.1: An overview of the components used in our image-based rendering system.

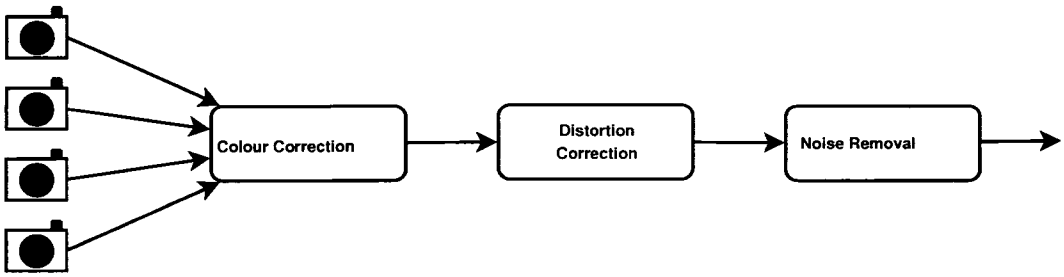


Figure 3.2: The primary steps of the image capture component.

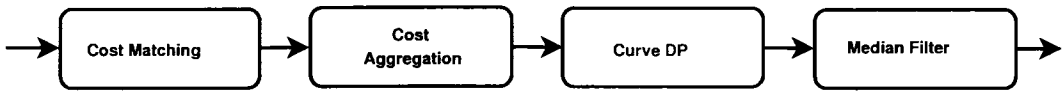


Figure 3.3: The primary steps of the stereo matching component.

calibration. Finally, a 3x3 median filter is applied to the image as a quick and simple way to reduce the effects of noise on subsequent processing, at the cost of some minor image detail. More details for the capture stage can be found in Appendix B.

Following image capture, the corrected images are sent to the stereo matching component for depth map computation (see Figure 3.3). Image pixels are compared on the GPU in the cost matching step and the raw match scores are then processed by a GPU-based shiftable aggregation window. The aggregated match costs are then transferred to the CPU where a novel space-filling curve-based dynamic programming optimization method is applied to arrive at a temporary depth map. For added temporal consistency and fewer errors, an additional post-process filters the depth map results. These steps are described in more detail in Section 3.1.

Finally, the filtered depth maps are combined with their corresponding reference images in the GPU for the rendering step (Figure 3.4). A backward view-synthesis algorithm is applied to search for the correct pixel colour in reference images based on the current position of the target novel view. Any holes in the rendered result are subsequently filled using a GPU-based heuristic, and the final result is displayed on the screen.

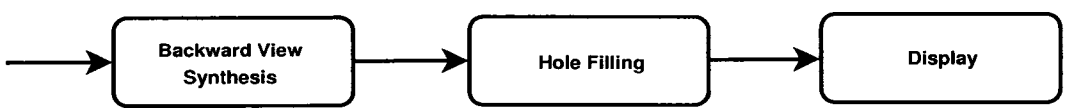


Figure 3.4: The primary steps of the rendering component.

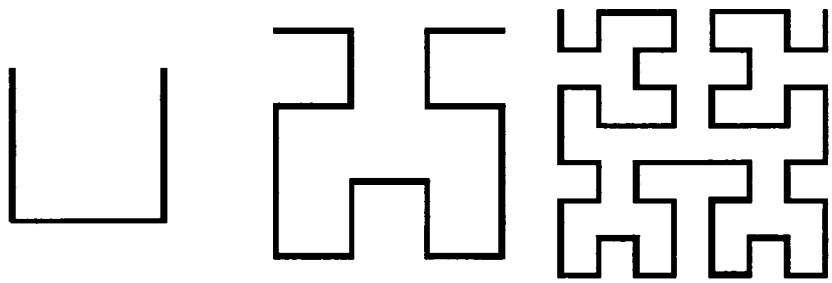


Figure 3.5: Hilbert curves at progressively increasing levels of resolution.

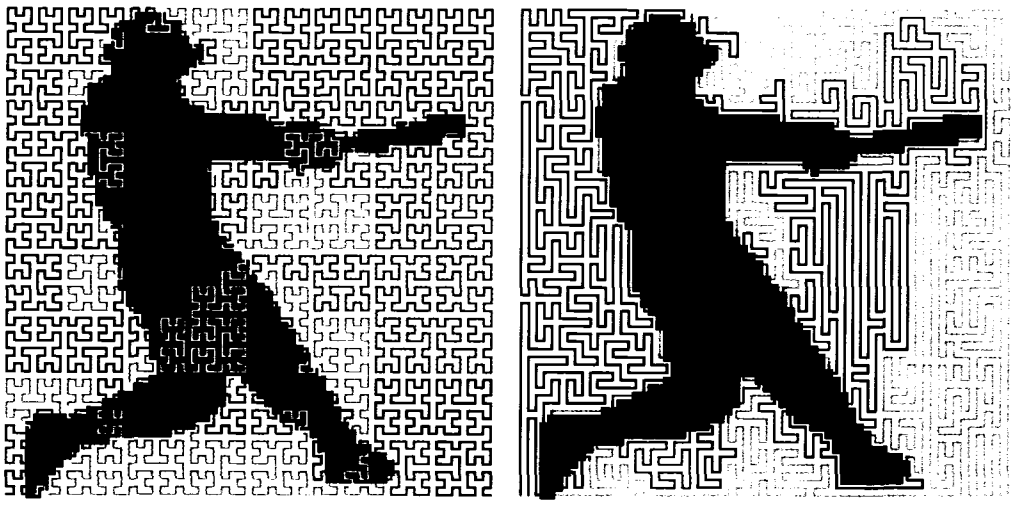
3.1 Space-Filling Curves for Stereo Matching

A principal component of the IBR system presented in this thesis is the stereo-matching depth-estimation module. In this section, we introduce a new space-filling curve-based dynamic programming approach for stereo matching. The space-filling curves may be custom generated offline for static scenes or applied randomly over several frames and filtered in a post-process suitable for real-time applications.

3.1.1 Space-Filling Curves

According to Breinholt and Schierz, space-filling curves were first introduced by Peano in 1890 and subsequently further developed and popularized by Hilbert in 1891 [4]. Space-filling curves are essentially curves that cover every point in a discrete multi-dimensional space. They are commonly used to describe multi-dimensional problems in terms of a single dimension, and have many interesting properties that can be advantageous under certain circumstances. Typical space-filling curves are defined recursively, so construction of the curve is a relatively simple matter.

The most famous (and widely used) example of this is the Hilbert curve (sometimes called the Peano-Hilbert curve), shown in Figure 3.5. Curves like the Hilbert curve have a strong locality property. The Hilbert curve will visit all points in a quadrant before continuing to another quadrant (at any recursive level of curve resolution). This property makes the Hilbert curve attractive to applications that exploit some spatial coherence among data elements. As such, space filling curves have found limited use in a number of varied fields, such as database indexing [28] and image compression [34]. A short list of known space-filling curve applications



(a) Hilbert curve (b) Custom-generated CSFC

Figure 3.6: Space-filling curves built over the silhoutte image of a baseball player. The curves change colour when crossing object boundaries.

may be found in [4].

Context-Based Space-Filling Curves

In an effort to further exploit the spatial coherence in images and other typical space-filling curve problem domains, Dafner et al. introduce context-based space-filling curves (which we will abbreviate as CSFC) [8]. The CSFC is custom-generated to create a curve shape that is well suited to the data it is to be used with. Dafner et al. use the example of run-length image compression to illustrate their point. While the strong locality property of the Hilbert curve will mean that it tends to remain in a certain image region before moving to another, it will cross image boundaries often, potentially ruining any run-length compression performed along the curve path. On the other hand, a CSFC will traverse each image region as much as possible before crossing to the next one, limiting the frequency of the curve "stepping out" of a region. This is illustrated in Figure 3.6.

The generation of a CSFC is relatively straightforward. To begin, a weighted graph is built over the image. The vertices of the graph each define a circuit connecting 4 adjacent pixels in a 2x2 square. Weighted edges connect neighbouring squares, and are defined based on the intensity difference of adjacent vertices. Dafner et al. define the edge weights for an existing curve vertex C_i and potential curve vertex

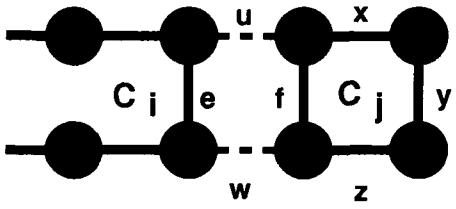


Figure 3.7: Dafner et al.'s CSFC edge-weighting scheme.

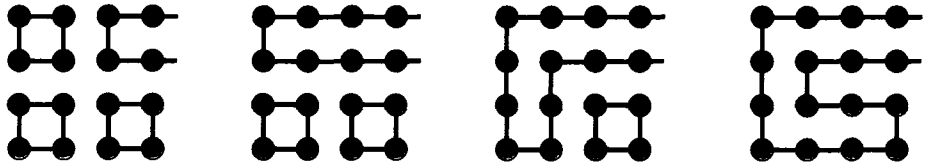


Figure 3.8: An example of CSFC construction via the minimum spanning tree.

C_j as $W(C_i, C_j) = |u| + |w| + |x| + |y| + |z| - |e| - |f|$, where the edges are placed as shown in Figure 3.7. Each term of the equation is simply the sum of intensity differences for each colour channel of the connected pixels. In effect this equation compares the weights of potential curve edges u and w with those of the existing curve edges e and f, which would be removed if the nodes were joined. The $|x|$, $|y|$, and $|z|$ terms measure colour consistency across the 4 pixels of the potential node, ensuring that nodes straddling an object edge are weighted higher to discourage the curve from moving across object boundaries.

To construct a space filling curve, a minimum spanning tree is iteratively built over the graph discussed above. As the minimum spanning tree grows and adds 2x2 "circuit" vertices, it forms a Hamiltonian circuit that eventually covers all pixels in the image. A simple example of this process is illustrated in Figure 3.8. Notice that the curve avoids the red pixels as it is being constructed, staying in the blue region for as long as possible. Because the building blocks of the CSFC are 2x2 blocks, it is possible that a predefined circuit will straddle a boundary edge separating dissimilar data. To accommodate for this possibility, Dafner et al. provide the ability to "split" a 2x2 vertex vertically or horizontally during the minimum spanning tree construction on the condition that its four neighbours are already in the tree.

Dafner et al. demonstrate the strength of CSFCs by examining the autocorrelation of 1-D pixel sequences generated by CSFCs and Hilbert curves on the same

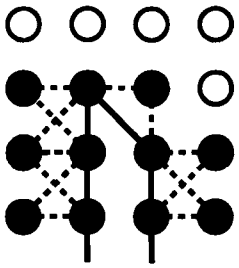


Figure 3.9: Valid pixels that may be added to a high-resolution CSFC curve segment. The existing curve is coloured with black pixels.

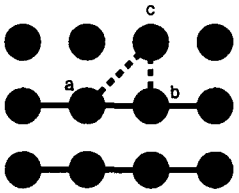


Figure 3.10: Pixel c is being considered for attachment via pixels a and b.

input image. They find that the CSFC outperforms both the Hilbert curve and basic scanline traversal [8], achieving a higher mean autocorrelation over a set of several sample photographs.

High-Resolution CSFCs

The CSFC is able to exploit spatial coherence much more effectively than other space-filling curves, but the relatively low resolution of the 2x2 curve "building blocks" means that the curve still may occasionally be forced to cross data boundaries. Although it is possible to split the 2x2 circuits, Dafner et al. only do this during the curve construction step when a node is surrounded by nodes already added to the tree, a condition that may not always be satisfied. For this reason, we introduce an extension of the CSFC which we call the high-resolution CSFC.

While a CSFC is built out of grid-based primitives by adding 4 pixels at a time, the high-resolution CSFC is built 1 pixel at a time. In a sense, the node primitive of the high-resolution CSFC can be considered a right triangle in which two corners of the triangle are already part of the minimum spanning tree. For example, Figure 3.9 shows an incomplete segment of a high-resolution CSFC and all the possible triangles that may be added to it.

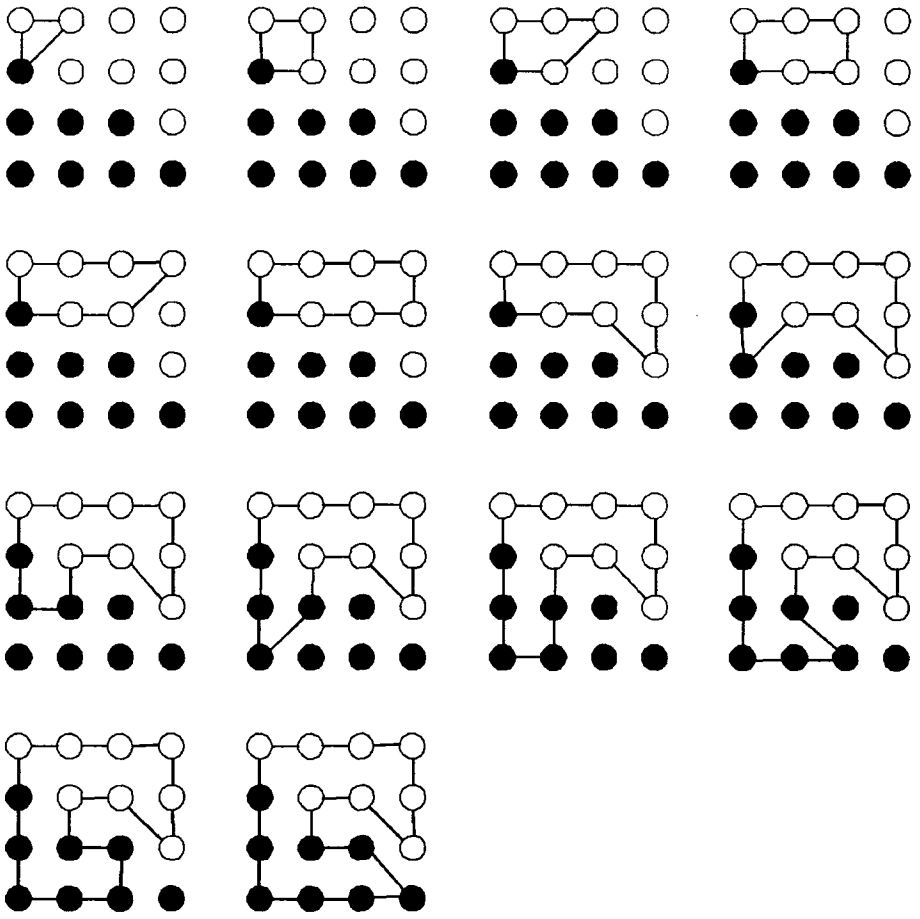


Figure 3.11: A space-filling curve is constructed one pixel at a time.

Because we are only adding one pixel at a time to the curve, a much simpler weighting function may be used. Suppose we are connecting pixel c to the curve via pixels a and b, as shown in Figure 3.10. In this case, the edge weight used in the minimum spanning tree construction is defined by the equation $W(C_{ab}, C_c) = |ac| + |bc|$, where $|ac|$ and $|bc|$ describe the sum-of-squared-difference over each colour channel between pixel c and pixels a and b, respectively.

High-resolution CSFC construction proceeds in much the same way as regular CSFC construction. We begin with a right triangle defined at an image corner and grow the curve one pixel at a time to cover the entire image using Primm's minimum spanning tree algorithm as shown in Figure 3.11.

Due to the use of triangles as curve primitives, certain potential triangle nodes must be disqualified as the curve is constructed to maintain the curve's space-filling

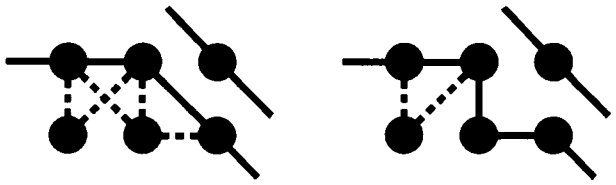


Figure 3.12: On the left, a space-filling curve under construction with 3 potential ways to grow. On the right, a new node is added to the curve, invalidating the other node connected to the new pixel.

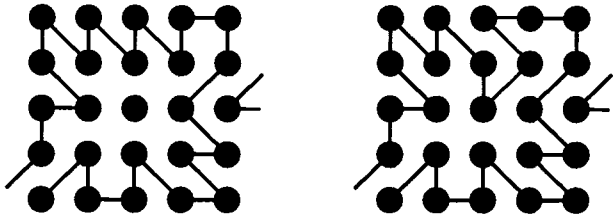
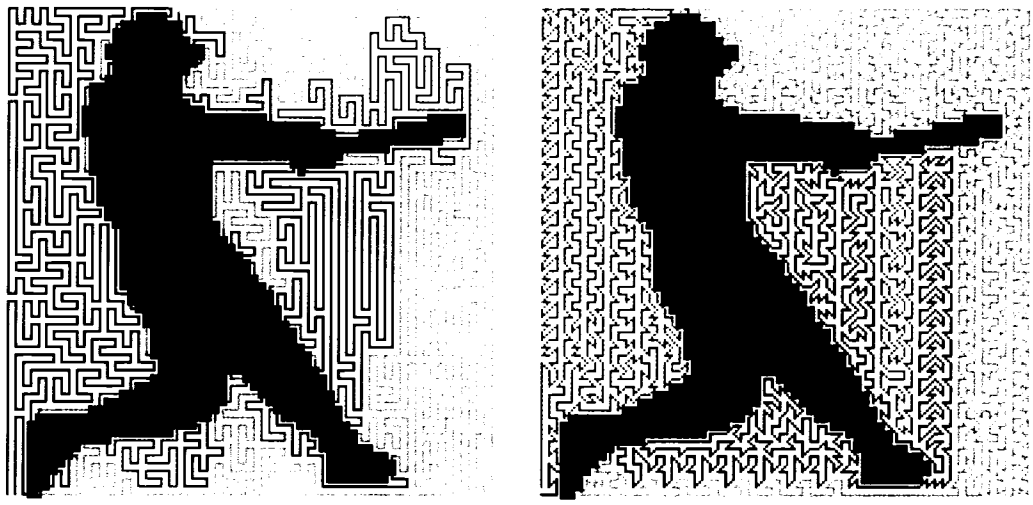


Figure 3.13: A hole created due to the lack of suitable edges adjacent to the pixel. On the right, neighbouring edges in red are modified to place the missing pixel in the curve.

property. Obviously, when a triangle involving a new pixel is added to the curve, that pixel may not be subsequently added to neighbouring curve segment. If all the pixels in a triangle node are added to the curve, we must remove it from consideration as shown in Figure 3.12.

Finally, the lack of a regular grid-based structure means that there is no guarantee a high-resolution CSFC will actually be space-filling (i.e. it is possible for holes to be created in the constructed curve). All of the holes we have encountered in our tests have been characterized by the sawtooth pattern shown in Figure 3.13. To fill in these holes, we simply run a post-process to apply a local curve modification so that the nearby sawtooth pattern encompasses the unassigned pixel as shown in Figure 3.13. If the space-filling property is not absolutely critical to the application, the holes may remain unfilled.

The high-resolution space-filling curve is able to conform to object boundaries much more effectively than Dafner et al.'s CSFC. As shown in Figure 3.14, the curve effectively stays inside boundaries in areas where the grid-based structure of the regular CSFC forces crossings. Unfortunately, the higher resolution and irregular shape of the high-resolution CSFC would imply that data storage costs invalidate the



(a) (b)

Figure 3.14: CSFCs built over the silhoutte image of a baseball player. The curves change colour when crossing object boundaries. (a) Dafner et al.'s original CSFC; and (b) the new high-resolution CSFC.

original image-compression application of CSFC's presented in [8]. In spite of this, the CSFC remains applicable to other problems requiring a strong correspondence between curve shape and image regions (such as dynamic programming in stereo matching).

3.1.2 Dynamic Programming on Space-Filling Curves

Applying dynamic programming to space-filling curves is a simple matter of utilizing the scanline optimization technique previously used to optimize over lines in an image [38]. As discussed in Section 2.1, scanline dynamic programming seeks to assign a disparity value to each pixel so that the cost function $E(d) = E_{data}(d) + E_{smooth}(d)$ is minimized. In this case, $E_{data}(d)$ represents the aggregated match cost for each assigned disparity value, and $E_{smooth}(d)$ is a constant jump penalty incurred when the assigned disparity changes between adjacent pixels.

Since the traversal of a space-filling curve essentially turns a 2D pixel grid into a one-dimensional pixel ordering, applying DP to the curves is simply a matter of using the scanline-based DP algorithm on the pixels traversed by the curve. Because the direction the curve is traveling may change at any time, we do not incorporate selection constraints such as the ordering/monotonicity constraint in our approach.

In our implementation a modified version of the efficient cost optimization algorithm given in [14] is used as is presented in Algorithm 2:

Algorithm 2 Efficent DP cost calculation using space filling curves.

```
Initialize S[0,d], T[0,d], and m[0]
for curve index i = 0 to width × height do
  /* Set p to pixel coordinates at i^th curve index */
  Set p to curveYIndex[i] × width + curveXIndex[i]
  for disparity d = 0 to disparityRange do
    /* IF d is also the preceding disparity on the current best-cost path
    OR the cost of maintaining the same disparity on a previous non-
    best cost path is less than the cost of the current best-cost path and
    a jump penalty λ */
    if m[i − 1] == d OR S[p − 1, d] < S[p − 1, m[p − 1]] + λ then
      /* Maintain same disparity as previous node on path */
      Set S[p, d] to S[p − 1, d] + C[p, d]
      Set T[p, d] to d
    else
      /* Switch to new disparity and incur jump cost penalty */
      Set S[p, d] to S[p − 1, m[p − 1]] + λ + C[p, d]
      Set T[p, d] to m[p − 1]
    end if
    /* IF the cost of this path is less than the cost of the current best
    path at this pixel */
    if S[p, d] < S[p, m[p]] then
      /* Track the current minimum cost path */
      Set m[p] to d
    end if
  end for
end for
Set bestDisparity to m[p]
Set result[p] to bestDisparity
for curve index i = width × height − 2 to 0 do
  /* Trace back the best cost path and save the results */
  Set bestDisparity to T[p, bestDisparity]
  Set result[p] to bestDisparity
end for
```

In Algorithm 2, we move along the curve accumulating and storing potential best cost paths. The array C is the record of local matching costs after cost aggregation has been applied. As we iteratively accumulate costs, S stores the cost of the best path for the current pixel/disparity combination, and T the previous assignment in that path. The variable m stores the current minimum cost path. After the costs

have been accumulated, we use the current state of m and the paths recorded in T to trace the best cost path from back to front and record the final result.

Occlusion Handling

Because they are generated according to the intensity information in an image, context-based SFCs can be particularly good at computing disparity along object boundaries and avoiding the streaking problem typically associated with scanline DP. However, they are especially susceptible to error caused by occlusion. The strong locality property of the SFC will cause it to linger in occluded areas, taking on erroneous disparity values and "dragging" them beyond the occluded region in an effect similar to streaking. To combat this we introduce a simple cross-check post-process to clean up error. Similar techniques have been used in several recent papers, including [23] and [29].

For this work, we use the technique previously published in [29]. The disparity maps for both left and right images are computed. Each disparity map may then be compared against its partner using the weak consistency constraint [16]. The weak consistency constraint states that for a pixel p with disparity d_1 in the original image, the corresponding pixel q with disparity d_2 in the reference image should have a disparity that is greater than or equal to d_1. If this condition is violated, we assume d_1 is an erroneous disparity due to occlusion and replace it with d_2.

3.1.3 Cost Matching and Aggregation

Cost Matching

Although the space-filling curves can be used interchangeably with a number of matching and aggregation strategies, we perform matching and aggregation on the GPU in a manner similar to that used in [19]. Based on the current disparity hypothesis d selected, we select a pixel from the reference image offset along the horizontal scanline (which coincides with the epipolar line) by d pixels. The intensities in each colour channel of the reference images are then compared to the original pixel in the matching image, and the results for each colour channel of a given pair are then added to obtain the final matching cost.

This is implemented efficiently on the GPU using a single shader program. For

each rendering pass, a fragment shader compares the given matching image with a single reference image at four contiguous disparity hypotheses and packs the results in a four-channel RGBA texture. Gong uses this approach in [20] to allow for easier GPU-based horizontal and vertical scanline DP. Although we are not applying DP on the GPU, we find this method of packing still gives a slight performance increase compared to the approach of packing 4 horizontally adjacent pixels at a single disparity hypothesis used in [19].

Cost Aggregation

After matching, we apply a shiftable box filter to reduce the effects of image noise while preserving object boundaries. This is done on the GPU in a two rendering pass approach described in [19]. The first rendering pass applies a simple 3x3 mean filter to the match cost texture. The second pass takes this result and selects the minimum averaged match cost in a 3x3 window centered on each fragment. This achieves the effect of a mean filter with a shiftable center. Of course, each shader program is coded to take advantage of the four-channel packing scheme, allowing us to aggregate costs for four depth levels in a single pass. After cost aggregation, a shader program is used to copy the costs for the current four disparity hypotheses to a single large match cost texture which is later transferred to CPU memory for dynamic programming optimization.

3.1.4 Random Space-Filling Curves

Custom generating space-filling curves according to intensity and/or region information gives acceptable results, but curve generation can be slow (taking between 20 seconds and a few minutes depending on the image resolution) and is currently unsuitable for real-time performance. For this reason, we also investigate the use of pre-generated random space filling curves. These curves are generated in the same manner as above, but instead of defining minimum spanning tree edge weights using image intensity information, we use a random number. The result is a random space-filling curve that may be saved to disk and used in the future without any generation time.

Although random space-filling curves maintain the advantage of enforcing con-

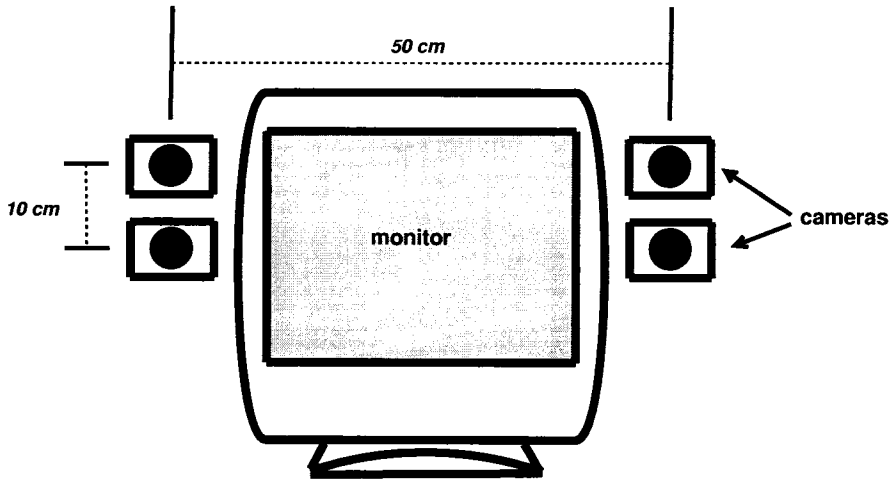


Figure 3.15: The camera configuration used in our experiments.

sistency in several directions over the course of the image and thus avoiding visually jarring "streaking" errors, these errors are replaced by small "blobs" of erroneous labels along foreground object boundaries where the curve leaves the object region and moves around the local area. Our opinion is that these blob errors are less visually offensive than streaking errors (especially in IBR scenes rendered with the disparity maps). However these errors still have a negative impact on result quality.

Fortunately, the random nature of the curves also means that the location of these errors is usually inconsistent between different curves, allowing us to detect and remove most of them. To do this, we can apply DP over several random curves for the same scene. The final disparity results for all curves are compiled together, and the median disparity at each pixel is selected as the final result. By doing this, we are able to remove many labeling errors in which curve shape is a contributing factor (assuming the disparity consensus among the majority of curves is correct).

3.1.5 Experimental Setup

For the IBR experiments presented in this thesis, we use a configuration consisting of four colour Point Grey Firefly firewire cameras, each with a native resolution of 640x480, arranged in a rectangular fashion. A diagram of our experimental configuration is presented in Figure 3.15. Although the number of cameras used and their position is generally up to the user, we have found this particular setup performed

reasonably well in experiments. Please note that the horizontal baseline in this case is wide enough to accommodate the insertion of a 19-inch CRT monitor between the cameras so that the approach may be used for position and eye gaze correction during teleconferencing.

We have found that limiting the configuration to 2 cameras produces large amounts of occlusion errors, which are amplified due to the extra-wide horizontal baseline involved. In fact, for the horizontal pairs in our stereo rig a large foreground object such as a person will often occlude completely independent sections of the background, making an entire half of each reference image impossible to match. For this reason, we use two additional cameras to create a vertical pairing with a much shorter baseline. This aids in the accuracy of matching and resolves most of the occlusion difficulties encountered with two cameras.

Because the placement of cameras around a monitor demands an extra-wide baseline, the disparity range in such a situation becomes impractical for interactive-rate implementations (in our experimental setup, the disparity for foreground objects at a depth of 60 centimeters may be as high as 200 pixels). In addition, rectifying more than three cameras in an arbitrary configuration is not a well-investigated problem. To circumvent this and allow the general placement of several cameras, we have modified the original method to deal with depth maps from calibrated cameras instead of disparity.

The cameras are set up to capture 320x240 resolution images at a rate of 30 Hz. The images are captured in a YUV 4:2:2 colour format (Y is sampled at every pixel and U and V are sampled at every second pixel on a horizontal line). These images are transferred to the CPU, where we convert them from the YUV 4:2:2 colour space to the RGB space, which is a more natural pixel format for GPU processing. In addition, we perform colour and distortion correction to prepare images for stereo matching (please see Appendix B for more information).

For the experimental IBR implementation, cost matching and aggregation is handled entirely on the GPU using methods nearly identical to those described above. However since we are working with depth values, matches are found by projecting pixels in to the 3D space and then reprojecting them to the reference cameras rather than simply shifting across a horizontal scanline using disparities.

We start with the images captured at the current time frame for all input cameras. For each of these images we wish to compute a grey-level depth map describing the depth of each pixel. We define depth using the same definition presented in [12]. That is, the depth labels in our depth maps correspond to a depth function $D : \mathbb{R}^3 \rightarrow \mathbb{R}$ such that for all scene points $P, Q \epsilon \mathbb{R}^3$ and all cameras k, if P occludes Q in k then $D(P) < D(Q)$. As noted in [12], this constraint is automatically satisfied when the reference cameras are all sitting on one side of an imaginary plane looking at the other side. This condition is satisfied in all of our experiments.

The depth maps look very similar to disparity maps, but the grey-scale intensity d of each pixel actually references a real-world depth value z in the range $[z_{min}, z_{max}]$ as computed using this equation from [24]:

$$z = \frac{1.0}{\frac{d}{255.0} * (\frac{1.0}{z_{min}} - \frac{1.0}{z_{max}}) + \frac{1.0}{z_{max}}}. \tag{3.1}$$

For each pixel $P(u, v)$ of a matching image I with a 4×4 camera matrix M with elements M_{ij} from the ith row and jth column, we project the pixel to a location in world coordinates (x, y, z) at a given depth z with respect to the world origin using the projection equations 3.2 given in [24]. Segments of the equations that are constant across all pixels are computed ahead of time on the CPU and cached for the GPU fragment shader programs to improve performance.

$$\begin{aligned}
c_0 &= z * M_{02} + M_{03}, \\
c_1 &= z * M_{12} + M_{13}, \\
c_2 &= z * M_{22} + M_{23}, \\
y &= \frac{u \times (c_1 \times M_{20} - M_{10} \times c_2) + v \times (c_2 \times M_{00} - M_{20} \times c_0) + (M_{10} \times c_0 - c_1 \times M_{00})}{v \times (M_{2,0} \times M_{01} - M_{21} \times M_{00}) + u \times (M_{10} \times M_{21} - M_{11} \times M_{20}) + (M_{00} \times M_{11} - M_{10} \times M_{01})}, \\
x &= \frac{y \times (M_{01} - M_{21} \times u) + c_0 - c_2 \times u}{M_{20} \times u - M_{00}}.
\end{aligned} \tag{3.2}$$

Remember that the world coordinates are defined with respect to the reference frame of a camera selected during the calibration step. After this projection, we re-project the corresponding point in world coordinates to image coordinates in each remaining reference image simply by multiplying the world coordinates by the camera matrix for each camera j:

$$P_j(u,v) = M_j * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}. \tag{3.3}$$

This operation is performed in the match cost fragment shader and the resulting texture coordinates are used to look up the matching reference pixel for comparison. At the end of this process, we have matching cost images for each possible pairing of the matching image I and each reference image I_j.

We combine the matching costs obtained from all reference images into a single matching cost image, which is then added to the large matching cost texture that is used for the DP calculation. To do this, we run a fragment shader that selects the minimum matching cost from each reference image at each fragment for the final computation. This approach avoids the inclusion of high match costs caused by occlusions in some of the reference images. Unfortunately simply taking the minimum matching cost is susceptible to image noise (other more effective methods are discussed in [35]). We use it in spite of this because competing methods such as the sort-summation (see [35]) work best under conditions in which image features are visible in more than one of the reference image cameras, a condition that our experimental setup cannot guarantee.

3.1.6 Temporal Filtering

Because our IBR implementation uses the random curves discussed above, a median filter is employed in the temporal direction to clean up incoming results. In this case, rather than running several DP passes on each frame using different curves, we increase performance by running a single DP pass on the current frame and comparing the results to those obtained from previous frames. A record of the four previous (unfiltered) depth maps for each camera is stored in a single RGBA texture. After the DP optimization, the unfiltered depth maps for the current time frame are transferred back to the GPU. A fragment program runs for each reference camera and selects the median depth of the four previous frames and the current frame at each pixel. The result is used as the final depth map for rendering. In addition, the fragment program updates the record of previous frames to include the most recent frame.

Of course, the median filter only works correctly if the objects in the scene were relatively stationary over the majority of the previous four frames. This means that when an object moves, two or three frames are required for the depth map and rendered result to "catch-up." To prevent this, our fragment program performs a pixel-by-pixel Euclidean distance comparison of the incoming RGB frame for the current camera with the same camera's previous RGB frame, using a built-in Cg library function. If the distance is above a given threshold (see Appendix C), then each colour channel of the corresponding pixel in the RGBA texture of previous depth maps is replaced with the currently computed depth. The median filter will then be forced to use the current depth estimate instead of the result from previous frames.

3.2 Image-based Rendering

The random SFC-based depth maps discussed above are fast enough for real-time frame rates on a modern desktop PC, or interactive frame rates when several images are considered simultaneously. In this section we discuss the interactive GPU-based IBR technique that uses these depth maps to perform IBR on a desktop PC. The results are applicable to a wide range of applications, including gaze correction for teleconferencing.

3.2.1 Wide Baseline View Interpolation with Depth Maps

For rendering novel views, we use a modified version of the previously introduced GPU-based backwards rendering algorithm [48]. The original algorithm used disparity maps generated for rectified images, here we are dealing with depth maps for calibrated cameras. In spite of these differences, the algorithm proceeds precisely as before. However, instead of using the disparity-based formulation given in [17] and [48], we reformulate the underlying mathematics to deal directly with depth values.

As mentioned in Section 3.1, the depth maps created by the stereo matching component are represented by grey-level integers which are transformed into real-world depth values using Equation 3.1. After applying the equation, we have a real-world floating-point depth value within a user-specified range for every pixel of each input image. Because we assume the cameras have been calibrated, we are able

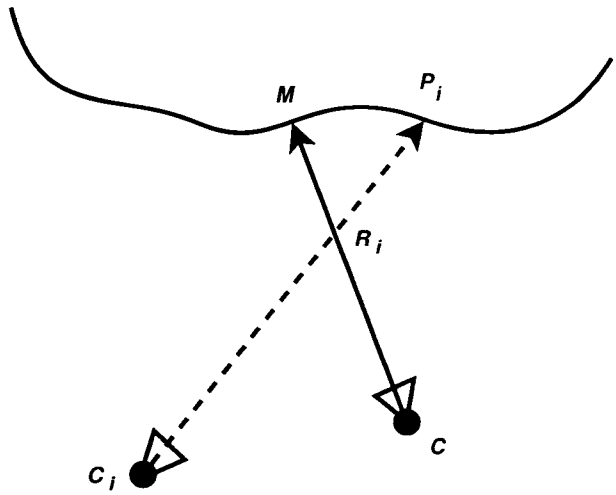


Figure 3.16: A 2D example of the depth-based backward search algorithm.

to remove the parallel and coplanar image plane assumption that was made in [17] and [48] and instead simply assume that the cameras are facing generally the same direction. In addition, the depth-based version of the algorithm allows the novel view to be translated and rotated in any direction without any ad-hoc modifications to the algorithm such as selective ray sampling or post-processing.

As shown in Figure 3.16, we are once again searching for the ray in the reference image C_i where the depth M of the ray's intersection with the object corresponds to the depth of the novel view C ray's intersection with the ray at R_i. It is important to note that in this case "depth" is defined with respect to the world coordinate system and does not refer to any local coordinate system or the length of the camera rays.

To determine the proper colour for a novel view pixel, we merely search for this intersection point from near to far, testing each depth hypothesis. Naturally, in our experiments it is very rare for the search to find an exact correspondence between the intersection of the two rays and the intersection of the reference ray with the object. For this reason we look for intersections that bound this point (dubbed the "zero-crossing point"), and approximate the correct colour based on those intersections as in [17].

The algorithm works iteratively as illustrated in Figure 3.17. At each iteration we project the novel view ray for each novel view pixel to two depths: foreground depth D_j and an adjacent background depth D_{j+1}. We call these depths the *observed*

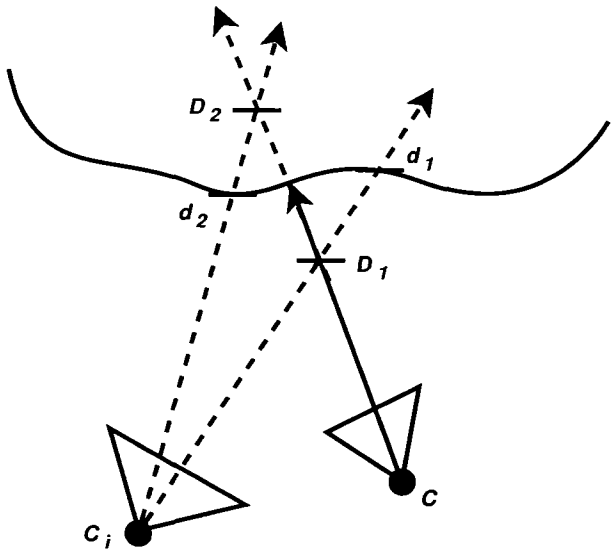


Figure 3.17: A 2D example of the depth-based zero-crossing point location.

depths. The points in world space at D_j and D_{j+1} are then projected into the reference image C_i, and the depths d_j and d_{j+1} as computed by the stereo matching component are retrieved from their respective pixels. We call these depths the *estimated depths.* We then compare the observed depths to the reference image estimated depths to determine if a zero-crossing point exists using the following equation:

$$F(p_u) = (d_j - Dj) \times (d_{j+1} - D_{j+1}) \tag{3.4}$$

The value of $(d_k - D_k)$ for some k in our depth range describes where the rays of the novel view and of the reference view intersect. If $(d_k - D_k) < 0$, then the rays intersect behind the object surface point defined by d_k. If $(d_k - D_k) > 0$, the rays intersect in front of the object surface point. And if $(d_k - D_k) = 0$, then the intersection corresponds directly to a zero-crossing point. Therefore any time the function $F(p_u)$ is less than or equal to zero we know we have a zero crossing point and the novel view ray intersects the object somewhere between D_j and D_{j+1}.

This algorithm is implemented in a shader program that handles a single depth pair per rendering pass. The two observed depth values are passed in as parameters, and the current fragment is projected into world coordinates at the specified

depths using Equation 3.2. These points are then projected into each of the reference cameras using Equation 3.3 with parameters passed in for the novel view projection matrix and each of the reference camera projection matrices. Each reference camera image is passed in as an RGBA texture with the alpha channel storing the depth values computed during stereo matching. After computing the re-projected reference camera pixel coordinates, we fetch the depth values using the GPU's built-in texture filtering capabilities to interpolate for floating-point pixel coordinates. The estimated depth values for each reference camera are then computed from the integer depth map values using Equation 3.1, and these values are compared to the observed depth parameters using Equation 3.4. If a zero-crossing point is found in one of the reference images, the resulting colour is calculated and output. Otherwise, we output a black pixel with an alpha value of 0.0 and the space is left open for future rendering. Because all four cameras are processed in the same shader program, it is possible to remove some suspected error in the rendered image by restricting the program to only output results when zero-crossing points are found simultaneously in two or more cameras. A second pass can then be used to fill in remaining areas with less confident single-camera zero-crossing points.

Similar to [48], we use a "rubber sheet" threshold to cull false zero crossing points detected between two separate foreground and background objects. In this work, the rubber sheet threshold is defined as $(D_i - D_{i-1}) \times \gamma$, where D_i and D_{i-1} are the two depths being considered for the current IBR rendering pass, and γ is a user-defined constant (we set it to 1.01). This removes any zero-crossings where the depth difference is larger than the range we are currently searching.

As in [48], each shader pass is run by drawing a quadrilateral at an increasing depth from the viewer using orthographic projection and the GPU's depth test. This has two helpful benefits. First of all, the depth test prevents the rendering of pixels that have already been drawn at a nearer depth, so foreground details are not overwritten with background information and we do not have to use a back-to-front "painter's algorithm." Secondly, modern GPU's implement an early z-kill feature that prevents such pixels at occluded depths from being processed by the fragment shader. This results in a substantial performance increase.

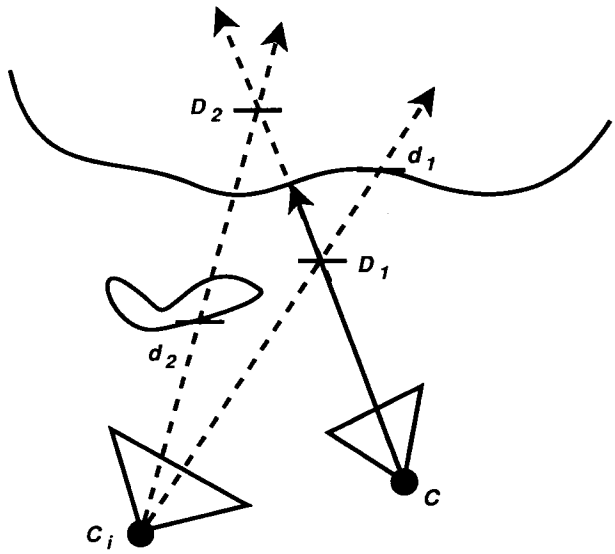


Figure 3.18: A 2D example of the object discontinuity conditions that may produce holes in the output image.

Hole Filling

After the initial rendering passes have been completed, there are often holes in the final image that should be filled. To do this we use the same heuristic-based hole-filling method introduced in [48], modified slightly to account for the change from disparity to explicit depth maps. Xu's hole-filling approach notes that holes in the IBR results occur at object discontinuities that fail the rubber sheet test. Based on the assumption that visible background continues behind the foreground as in Figure 3.18, holes are filled heuristically using neighbouring background information obtained during the IBR process. When the rubber sheet threshold detects a depth discontinuity and causes the zero-crossing test to fail, the pixel with the lower reference image disparity/greater depth d_1 as shown in Figure 3.18 is recorded as a potential hole-filler in the future. In the software-based implementation, the hole-filling solution can be written directly to the resulting image and potentially overwritten in the future if a proper zero-crossing is found.

Due to the limitations of GPU architecture, this temporary storage of hole-filling values is impossible with a GPU implementation. Instead Xu uses a second round of rendering passes after the initial IBR process for hole filling. The hole-filling shader is nearly identical to the IBR shader, except it disregards the rubber sheet

threshold and outputs the intensity of the reference image's background pixel as the final result. Because of the depth test and early z-kill, the hole-filling portion only runs over the remaining image holes and is very fast.

We fill holes using one camera at a time, processing reprojected pixels in the order from the background to the foreground before proceeding to the next camera. The cameras are each processed in the order from the nearest to the target novel view to the farthest, as the nearest camera is most likely to have the most accurate information for hole filling.

3.2.2 Background Model

Depending on the current position of the virtual camera and the configuration of the reference cameras, we may need to render segments of the background that simply are not visible in any reference cameras at the current frame. Assuming we are dealing with a foreground object such as a person and the background remains static over the course of session, the user may choose to activate a background model to help fill in holes.

The background model is maintained in a manner similar to that used by Criminisi et al. [7]. In their work, a histogram is created along each scanline for the incoming disparity map, and the valley in the histogram is selected as the disparity threshold segmenting background and foreground regions. In our case, to allow for an efficient GPU implementation, the depth threshold between background and foreground is a user defined-parameter in the interface (histogram computation is particularly inefficient on a GPU). To prevent inaccurately labeled foreground edges from "leaking" into the segmented background, we temporarily apply a 5x5 dilation filter to the depth maps used in background model updates. The filter is run over two GPU passes. The first pass runs a 5x1 vertical filter, and the second pass uses those results to run 1x5 horizontal filters for a final 5x5 window size. Each pass selects the maximum depth label (corresponding to foreground objects) under the 5x5 window, therefore enlarging the area covered by foreground objects and avoiding adding erroneously labeled foreground boundaries to the background model.

As in [7], the background model is updated at each time frame using the following equation:

$$I_B^t(p) = \tau I_B^{t-1}(p) + (1 - \tau) I^t(p), \tag{3.5}$$

where $I_B^t(p)$ is the RGB intensity and depth value for pixel p in the background model at time t. For this equation to apply, the depth corresponding to the reference image $I^t(p)$ must be above the defined background depth threshold. As in [7], the constant τ is a decay factor (the authors recommend using $\tau = 0.9$) used to promote temporal consistency and negate the effects of noise. Criminisi et al. compute three separate background models: one for their single disparity map and each of the two reference images. We maintain a separate background model combining intensity and depth for each reference camera, storing depth in the alpha channel of the GPU texture. Equation 3.5 is implemented in a shader program to update the model in a single rendering pass. For areas of background that have not yet been observed, we simply copy the current depth and intensity into the background and refine with future observations.

To render using the background model, we simply stop rendering with the reference images at the background depth threshold and switch to the background model images. In addition to filling holes that may otherwise exist, the background model has the additional benefit of reducing temporal artifacts and promoting temporal consistency in the background of the rendered result.

Ground Control Points

In our experimental system the reference cameras are arranged in such a way that occlusion of background points is minimized. However there are additional uses for the background model. In addition to being used in the rendered result, the background model can be used to establish "ground control points" (GCPs) that can increase the efficiency of the curve-based dynamic programming. These ground control points are used in the DP algorithm in a manner similar to that employed in [19], although they are established differently.

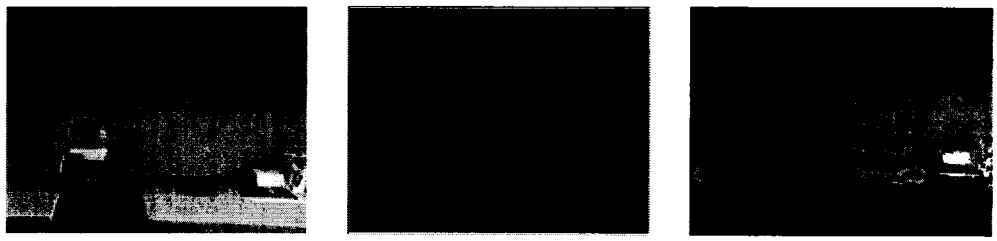
In our case, the term "ground control points" refers to intensity and depth observations in a static background that have been repeatedly confirmed over a majority of the previous frames. Along with the background model intensity and depth texture, we maintain a second texture that stores a flag for each pixel indicating

whether or not it is currently active as a GCP, and a counter indicating the "age" of the pixel prior to GCP activation. As background model observations are made, the counter for each pixel increments until it passes a user-defined threshold. At that moment the GCP is activated and locked in for future DP passes. While the GCP is active the counter decrements at each frame until reaching 0, at which point the GCP resets, new observations are accumulated, and the process repeats.

When we update the background model after computing the depth of each pixel, a check is performed on the depth of the incoming pixels. As mentioned above, pixels with a depth within the user-defined background threshold are added to the background model. Those outside of the range are rejected as foreground pixels, and the GCP counter is decremented to discourage future adoption of the pixels as background GCPs. If a pixel passes the test and contributes to the background model, the shader program checks if a GCP is currently active for the pixel. If it is not, the GCP counter is incremented and if it is then no action is taken.

At the next frame, immediately prior to the curve-based DP on the CPU, a separate shader program compares the current frame for each camera to the recorded background model via image differencing. If the GCP is currently active and the intensity difference at the current fragment is below a certain threshold, then the GCP remains active and the GCP counter is decremented. When the counter reaches 0, the fragment program deactivates the corresponding GCP. Should the observation give an intensity difference above the threshold, we assume we have detected a foreground pixel and the GCP and GCP counter are immediately reset. A final check is performed on the GCP counters to add any new ground control points. Any new counters that are large (or "mature") enough to justify a GCP have their corresponding GCP activated, and will be decremented in the following frames as discussed above. The behaviour of GCPs over time for an input scene and BG model is shown in Figure 3.19.

After the fragment program has been executed, the GCP flags are transferred to the CPU for the SFC dynamic programming passes. The GCPs lock in the depth at corresponding pixels using the approach discussed in [19], providing a substantial speed increase in the DP step. Specifically, if a pixel p with disparity d is labeled as a ground-control point, then the dynamic programming component will not look at



(a) (b) (c)

Figure 3.19: Ground control points for an office scene with toy bricks in the foreground. (a) The input image from a reference camera; (b) the corresponding depth map; and (c) active ground control points based on the depth map background model.

other potential disparities for that pixel, reducing the search space.

Chapter 4

Results and Analysis

All tests were performed on a system with an AMD Athlon 64 3800+ processor running at 2.41 GHz with 2 GB of RAM and a Geforce 6800 graphics card. The final implementation is programmed in C++ using FLTK for a GUI interface and OpenGL and nVidia's Cg shader language to utilize the GPU.

4.1 Stereo Matching Results

We evaluate the effectiveness of the curve-based approach using the popular Middlebury stereo datasets and rankings [38]. The Middlebury rankings evaluate stereo algorithms using four datasets ("Tsukuba," "Venus," "Teddy," and "Cones"). The Tsukuba and Venus datasets have been in use for several years and are now handled very effectively by the top stereo algorithms. They have a disparity range of 0-15 and 0-19, respectively. The Teddy and Cones datasets have only recently found widespread use. These datasets present more complex scenes, and both have a disparity range of 0-59.

The Middlebury rankings evaluate an algorithm's performance in three areas: regions of the disparity image that are not occluded in the reference image, regions of depth discontinuity, and overall accuracy evaluation performed over the entire disparity map.

First, we examine the effectiveness of median filtering in improving results with random high-resolution space-filling curves. The results computed by the Middlebury rankings with an error threshold of 1.0 are shown in Table 4.1. The numbers in the table represent the percentage of "bad" pixels with a disparity error higher

	Tsukuba			Venus			Teddy			Cones		
	nonocc	all	disc	nonocc	all	disc	nonocc	all	disc	nonocc	all	disc
RSFCDP (1 curve)	4.60	6.59	20.1	3.45	4.68	26.8	17.4	24.2	33.4	14.6	22.1	25.8
RSFCDP (5 curves)	3.49	5.59	17.4	2.39	3.57	23.5	13.9	21.0	29.5	12.0	19.7	21.9
RSFCDP (10 curves)	3.95	6.09	18.2	2.62	3.70	26.0	13.6	20.6	29.6	11.3	19.2	21.4
RSFCDP (15 curves)	3.42	5.56	16.5	2.44	3.58	25.3	13.5	20.6	28.5	11.1	18.9	20.5
RSFCDP (20 curves)	3.50	5.64	17.0	2.47	3.56	26.0	13.4	20.5	28.5	10.9	18.8	20.5
RSFCDP (25 curves)	3.42	5.54	16.5	2.31	3.42	24.8	13.6	20.7	28.4	10.8	18.7	20.2
RSFCDP (30 curves)	3.50	5.64	16.7	2.34	3.45	25.1	13.4	20.6	28.5	10.8	18.7	20.2
RSFCDP (35 curves)	3.42	5.53	16.4	2.23	3.35	24.3	13.5	20.7	28.3	10.7	18.7	20.1
RSFCDP (40 curves)	3.58	5.70	16.9	2.29	3.42	24.8	13.4	20.6	28.3	10.8	18.7	20.2
RSFCDP (45 curves)	3.46	5.57	16.5	2.26	3.40	24.1	13.4	20.6	28.2	10.8	18.7	20.2
RSFCDP (50 curves)	3.53	5.64	16.6	2.31	3.44	24.3	13.4	20.6	28.3	10.8	18.7	20.2

Table 4.1: The percentage of bad pixels for random curve-based DP (RSFCDP) on the Middlebury stereo datasets (error threshold of 1.0).

than the error threshold. A constant jump cost of 100 was used across all tests to ensure a fair basis for comparison. The value of the jump cost parameter was chosen based on previous experiments as a "middle ground" jump cost capable of producing acceptable results in a variety of situations.

As shown in the table, adding more random curves and selecting the correct disparity with a median filter is more effective than using a single random curve. The greatest improvement is seen when going from 1 curve to 5 curves, with marginal improvements in accuracy after that. Figure 4.1 shows results for random curves for the Tsukuba and Teddy datasets. We show results for 1 random curve, 5 random curves (the number used in our GPU IBR implementation), and 35 random curves. As shown, adding more random curves creates a significant visual difference by smoothing out the edges corresponding to object boundaries. Since curves can be pre-generated for a given image resolution and the computation for each separate curve is done in a single DP pass (a single curve pass requires 14.3 milliseconds on an 2.41 GHz Athlon 64 processor for the 384x288 Tsukuba images), execution is very fast even when several curves are used for computation.

Next, the effectiveness of random and non-random SFCs is compared to other recent DP approaches to solving the streaking problem. We compare to traditional dynamic programming [38], Veksler's tree-based DP [45], Gong and Yang's reliability DP [19], and Lei et al.'s region-tree DP [29]. The disparity jump penalty for each type of curve-based DP is selected to give best results (experimental parameters are listed in Appendix C).

As shown in Table 4.2, the curve-based algorithms are competitive with recent DP approaches in the Middlebury rankings. For the Tsukuba dataset, all

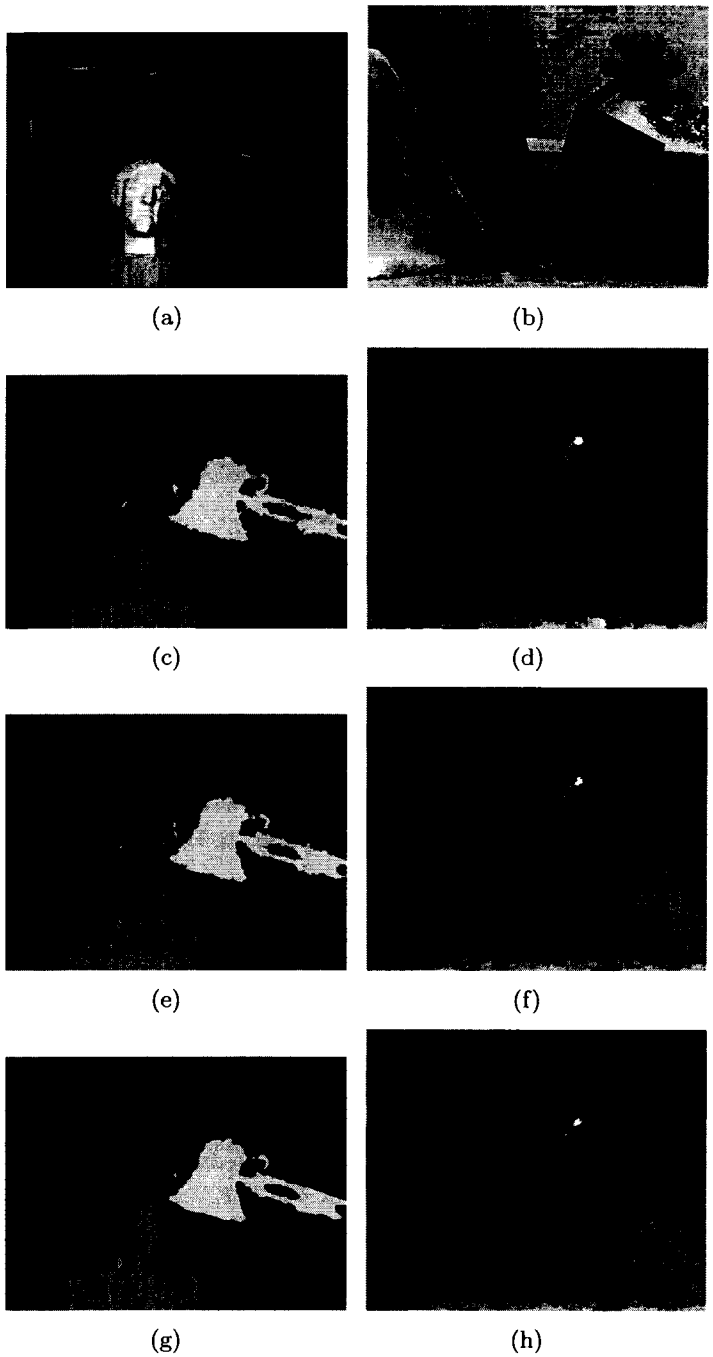


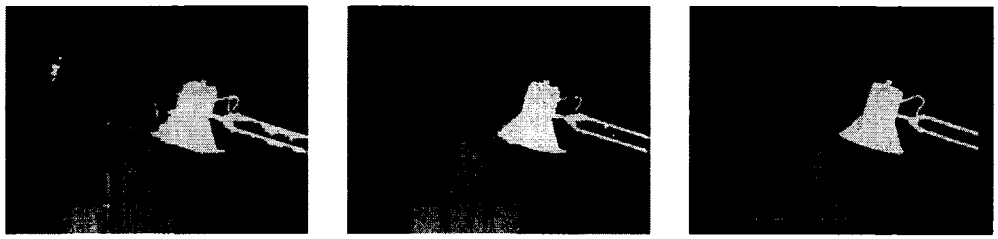
Figure 4.1: Selected results for random curve-based DP with median filtering. (a,b) The source images for the Tsukuba and Teddy datasets, respectively; (c,d) disparity maps for one random curve; (e,f) disparity maps for 5 random curves; and (g,h) disparity maps for 35 random curves.

	Tsukuba			Venus			Teddy			Cones		
	nonocc	all	disc	nonocc	all	disc	nonocc	all	disc	nonocc	all	disc
DP [38]	4.12	5.04	12.00	10.10	11.0	21.0	14.0	21.6	20.6	10.5	19.1	21.1
TreeDP [45]	1.99	2.84	9.96	1.41	2.10	7.74	15.9	23.9	27.1	10.0	18.3	18.9
ReliabilityDP [19]	1.36	3.39	7.25	2.35	3.48	12.2	9.82	16.9	19.5	12.9	19.9	19.7
RegionTreeDP [29]	1.39	1.64	6.85	0.22	0.57	1.93	7.42	11.9	16.8	6.31	11.9	11.8
RSFCDP (35 curves)	3.85	5.93	18.0	2.33	3.43	25.6	13.4	20.6	29.4	9.77	17.7	19.8
RSFCDP (35 curves + x-chk)	2.90	3.92	14.5	1.63	2.35	18.2	12.4	17.7	25.5	9.53	15.7	17.4
SFCDP (lo-res)	2.57	4.58	13.2	3.30	4.44	23.4	15.5	22.3	25.9	17.2	24.7	25.1
SFCDP (hi-res)	2.72	4.74	11.8	2.32	3.46	18.8	15.0	22.1	25.9	17.0	24.2	26.2
SFCDP (hi-res + x-chk)	2.28	2.83	11.10	0.75	1.01	6.13	12.7	17.6	22.5	13.8	19.3	22.2

Table 4.2: The percentage of bad pixels for various curve-based DP approaches, compared to other recent DP algorithms (error threshold of 1.0).

curve-based approaches outperform vanilla dynamic programming, with the high-resolution curves combined with cross-checking outperforming both pixel-tree and reliability-based approaches. Curve-based approaches are even more effective for the Venus dataset, where both random curves and non-random high-resolution curves outperform other DP algorithms (with the exception of region-tree DP). Results are not as good for the more complex Teddy and Cones scenes, but this is common among DP algorithms. For the Teddy dataset, curve-based DP outperforms pixel-tree and regular DP, and for the Cones dataset curve-based DP outperforms reliability-based DP, with random curves significantly outperforming all approaches except for region-tree DP. Also, high resolution curves consistently show better results than Dafner et al.'s original CSFCs. Occlusion handling is particularly effective for high-resolution curves. An example of the effectiveness of cross-checking with our approach is shown in Figure 4.2. Some of the images corresponding to the evaluation data in Table 4.2 are shown in Figure 4.3. As an example, a segment of each testing image with a non-random and sample random curve overlayed is shown in Figure 4.4. As shown, the non-random high-resolution CSFCs minimize object boundary crossings.

For complex datasets such as Teddy and Cones, the random curve-based approach shows significant promise when compared to non-random curves and other approaches, particularly in non-occluded areas. While random curve approaches are not as accurate around object boundaries when compared to approaches such as non-random curves or pixel-trees, they are able to remove error in object interiors and untextured areas. An example of this using the Cones dataset is presented in Figure 4.5. Notice that the result is notably cleaner, especially in the foreground cones and the upper right corner of the image.



(a) (b) (c)

Figure 4.2: An example of the occlusion handling cross-check method. (a) The computed disparity map using a non-random high-resolution CSFC; (b) the same disparity map after cross-checking; and (c) the ground truth disparity.

	Tsukuba			Venus			Teddy			Cones		
	nonocc	all	disc	nonocc	all	disc	nonocc	all	disc	nonocc	all	disc
DP [38]	19.6	20.6	22.8	23.5	24.3	32.8	30.0	36.3	36.1	22.0	29.6	33.7
TreeDP [45]	22.4	23.1	22.3	12.1	12.9	21.7	32.4	38.9	45.6	23.7	30.8	31.7
ReliabilityDP [19]	19.0	20.7	17.5	12.7	14.0	26.1	26.3	32.5	36.8	23.7	29.9	31.5
RegionTreeDP [29]	21.0	21.1	18.3	9.08	9.74	13.8	19.7	24.8	32.1	19.7	24.8	25.4
RSFCDP (35 curves)	16.4	18.2	22.3	7.55	8.76	27.2	22.2	29.7	39.3	21.9	28.9	33.0
RSFCDP (35 curves + x-chk)	16.4	17.3	21.1	6.82	7.68	20.9	22.0	27.8	36.6	22.3	27.5	31.4
SFCDP (lo-res)	23.2	24.8	22.8	11.5	12.7	29.2	25.9	32.9	40.4	28.6	35.1	38.8
SFCDP (hi-res)	23.8	25.4	21.4	10.3	11.5	25.3	25.2	32.3	38.7	28.7	35.0	39.9
SFCDP (hi-res + x-chk)	24.0	24.5	21.0	9.22	9.71	17.0	25.6	30.7	36.0	29.6	33.9	38.0

Table 4.3: The percentage of bad pixels for various curve-based DP approaches, compared to other recent DP algorithms (error threshold of 0.5).

Further confirmation of the merit of a random curve-based approach is found if the Middlebury error threshold is decreased to 0.5. The results of these tests are shown in Table 4.3. In this situation, random SFCs outperform every other algorithm in the Tsukuba and Venus datasets, and every algorithm except for region-tree DP in the Teddy and Cones datasets. This is especially true in non-occluded regions, suggesting that random curves may be combined with a method with better object discontinuity performance to yield a more powerful algorithm. The fact that the performance of random curves improves with a smaller error threshold further confirms the findings of Hirschmüller [23], who uses dynamic programming on several scanline angles for each pixel and then combines each hypothesis into a final result. Similar to our experience with random curves, the performance of Hirschmüller's algorithm increases significantly with a smaller error threshold, suggesting that consensus-based DP approaches are more effective when sub-pixel accuracy is required.

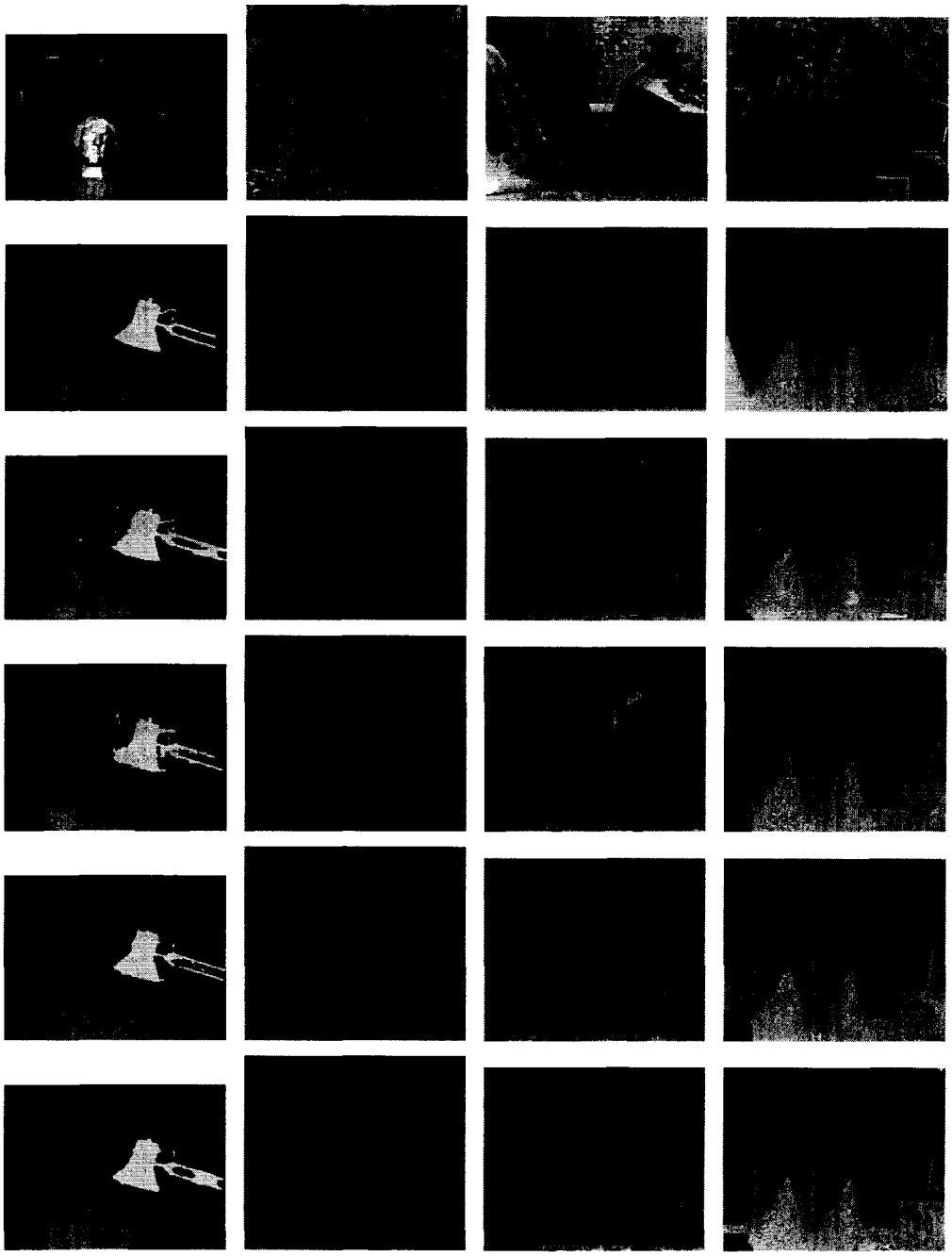
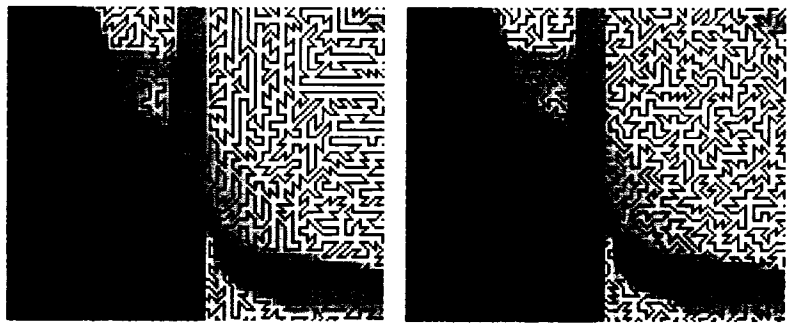
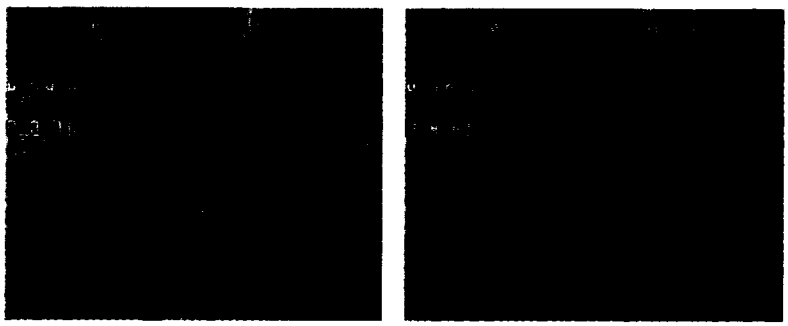


Figure 4.3: Resulting disparity maps. First row: disparity map reference images. Second row: ground truths. Third row: Veksler's pixel-tree DP results [45]. Fourth Row: Gong and Yang's reliability DP results [19]. Fifth Row: Results for 35 filtered random SFCs. Sixth Row: Results for non-random high-resolution SFCs.

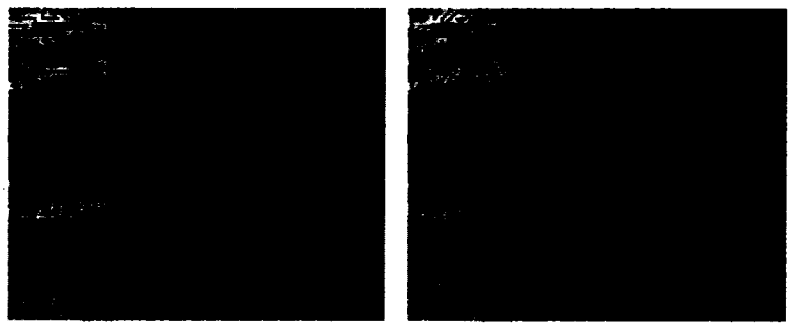


(a) Tsukuba, non-random (b) Tsukuba, random



(c) Venus, non-random (d) Venus, random

(e) Teddy, non-random (f) Teddy, random



(g) Cones, non-random (h) Cones, random

Figure 4.4: A zoomed-in section of each testing image, with the non-random curve used and a sample random curve.

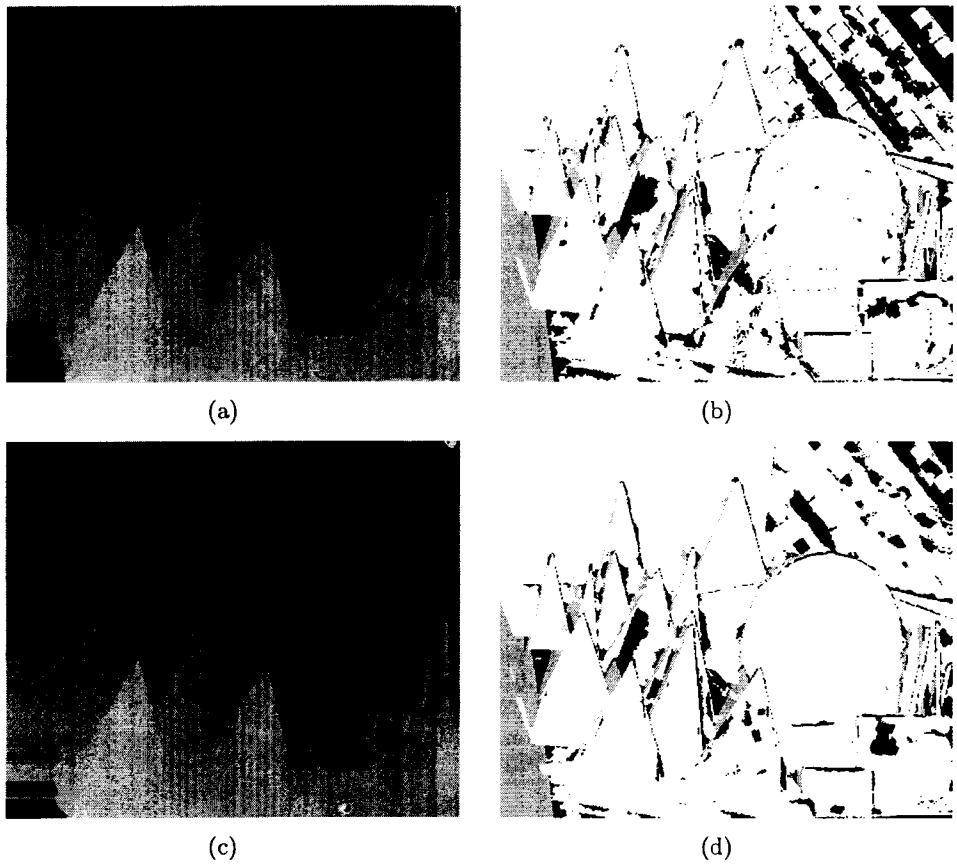


Figure 4.5: Results for the cones dataset, with cross checking enabled. (a,b) The disparity map and erroneous pixels, respectively, for non-random high-resolution curves; and (c,d) the disparity map and erroneous pixels, respectively, for 35 random high-resolution curves. In images (b) and (d), white regions denote pixels without disparity error, black regions denote matchable erroneous pixels (the absolute disparity error is greater than 1.0), and grey regions denote occluded erroneous pixels.

4.2 Image-based Rendering Results

A look at IBR results for a scene involving a person sitting in front of a static background is presented in Figure 4.6. As shown, the random curves with temporal median filtering approach is capable of producing dense depth maps in real-world experiments that accurately reflect the presence of foreground objects for rendering. The rendered result does suffer from some artifacts around foreground object boundaries due to the relative weakness of random curves when dealing with such features. As mentioned previously, random curves will cross image boundaries and drag out "blobs" of disparity in an effect similar to the streaking effect. Instances of these errors that are not removed by the median filter may manifest themselves in the rendered result. In addition, when portions of forgeground objects are only visible in one or two reference cameras, stereo matching may inadvertently match pixels incorrectly. As a result, discontinuities may happen in foreground objects around image boundaries. This can be seen in the torso of the person pictured in figure 4.6.

Error Checking Method

To determine the accuracy of the IBR system, a fifth camera was placed approximately at the center of the experimental rig. The camera was calibrated with respect to the other cameras so that its position and intrinsic parameters were known, and then the virtual camera in the IBR system was set to mimic those parameters. Following rendering, the IBR result is compared to the image captured in the fifth camera using image differencing. A sum of squared differences (SSD) operation is applied to the RGB channels of each pixel in the image, and then the mean SSD is taken across all pixels. To better describe the distribution of error around the mean SSD, we also compute the median SSD and minimum and maximum SSD across the set of all image pixels. To assure invariance to image noise, these operations are performed for 5 consecutive frames of capture and rendering. We then compute the average across those 5 frames for each statistic for final inclusion in the thesis.

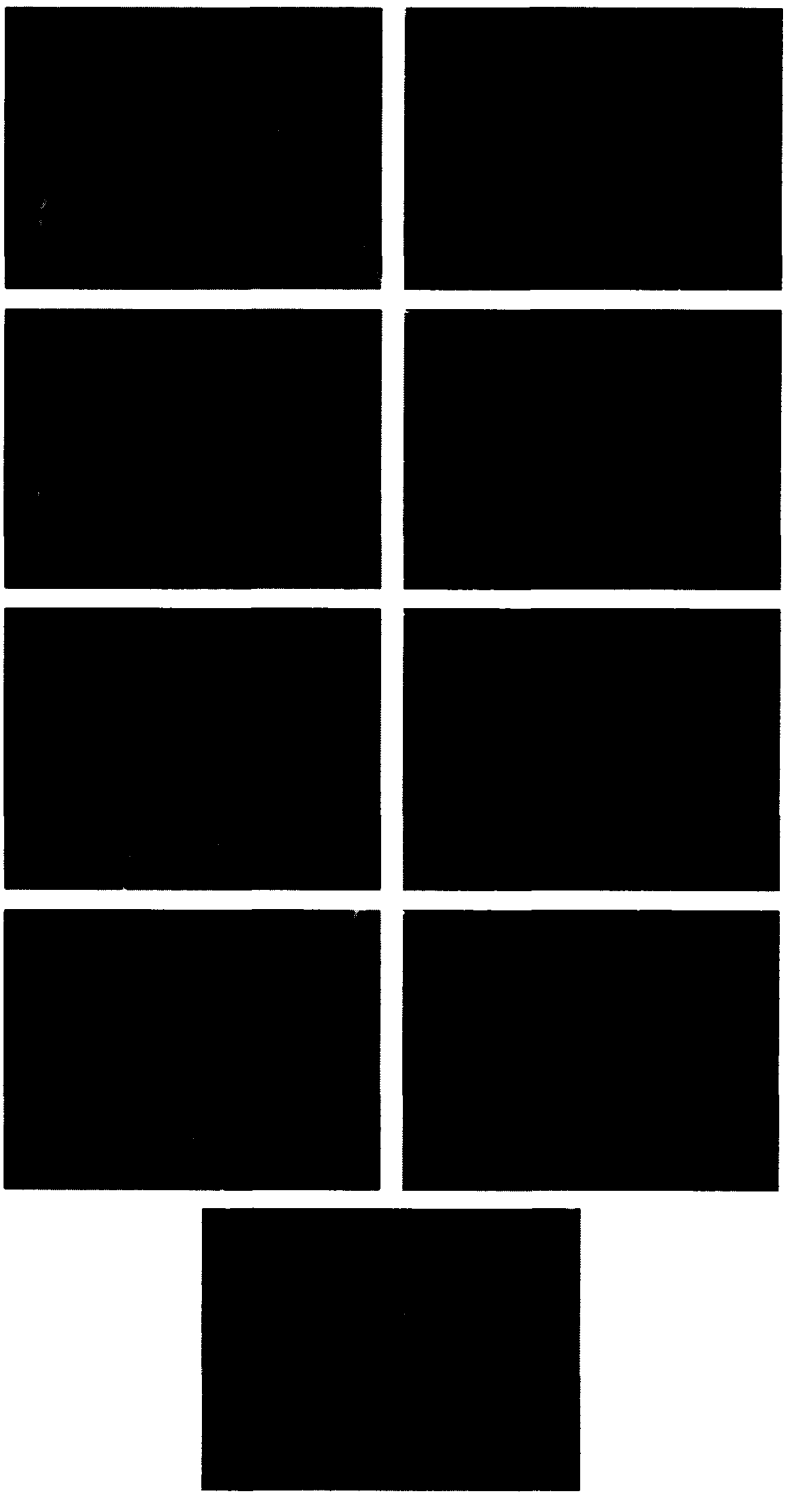


Figure 4.6: Example IBR results. The first four rows display images coming from each of the four reference cameras, with computed depth maps. The final row is the novel view image synthesized from a camera approximately in the center of the rectangle created by the four reference cameras.

Tests and Results

In Figure 4.7, we compare our IBR system stereo results using random and non-random curves with results obtained using Gong and Yang's GPU-based reliability DP [19]. To ensure a fair basis for comparison, the reliability match cost computation is identical to the one used for curve-based results. Match costs are transferred to the CPU where 3 separate DP passes are run: one horizontal pass with the reliability threshold enabled, a vertical pass with the reliability threshold decreased to fill in unreliable pixels, and a final horizontal pass with no reliability threshold to fill in any remaining pixels. Comparison tests are done on a static scene to ensure fairness, and all non-DP experimental parameters are identical. All curve-based results use an identical jump cost. It is worth re-emphasizing that although we evaluate results using non-random curves in this test, the construction time required means that the curve is not suitable for real-time applications.

Because of the extra passes demanded by the reliability DP approach, it runs slower than curve-based DP. In this example, reliability DP achieved 1.62 frames per second while curve-based DP achieved 3.15 frames per second (assuming one curve pass per captured frame). If more than one curve pass is performed per frame, then performance drops (1.6 fps for 5 curve passes, 0.46 fps for 15 curve passes, and 0.28 fps for 25 curve passes).

Figure 4.8 compares rendering results for the same scene presented in Figure 4.7. The mean, median, and max SSD scores over five consecutive frames of rendering for each approach are compared in Table 4.4. The minimum pixel SSD score was 0.0 in all examples, so it is not included in the table. As shown, the filtered random curve approach is competitive with non-random curves and reliability DP. As more curves are added, the error rate tends to decrease (and temporal consistency between frames improves). It should be noted that although the stereo results for reliability DP shown in Figure 4.7 appear more visually pleasing to the human eye, initially unapparent instances of streaking error manifest themselves in the rendered IBR result, causing the overall performance of reliability DP to be about on par with curve-based DP (at a reduced frame rate).

To further demonstrate the importance of median filtering in cleaning up depth maps for rendering and ensuring temporal consistency, we present an example of a

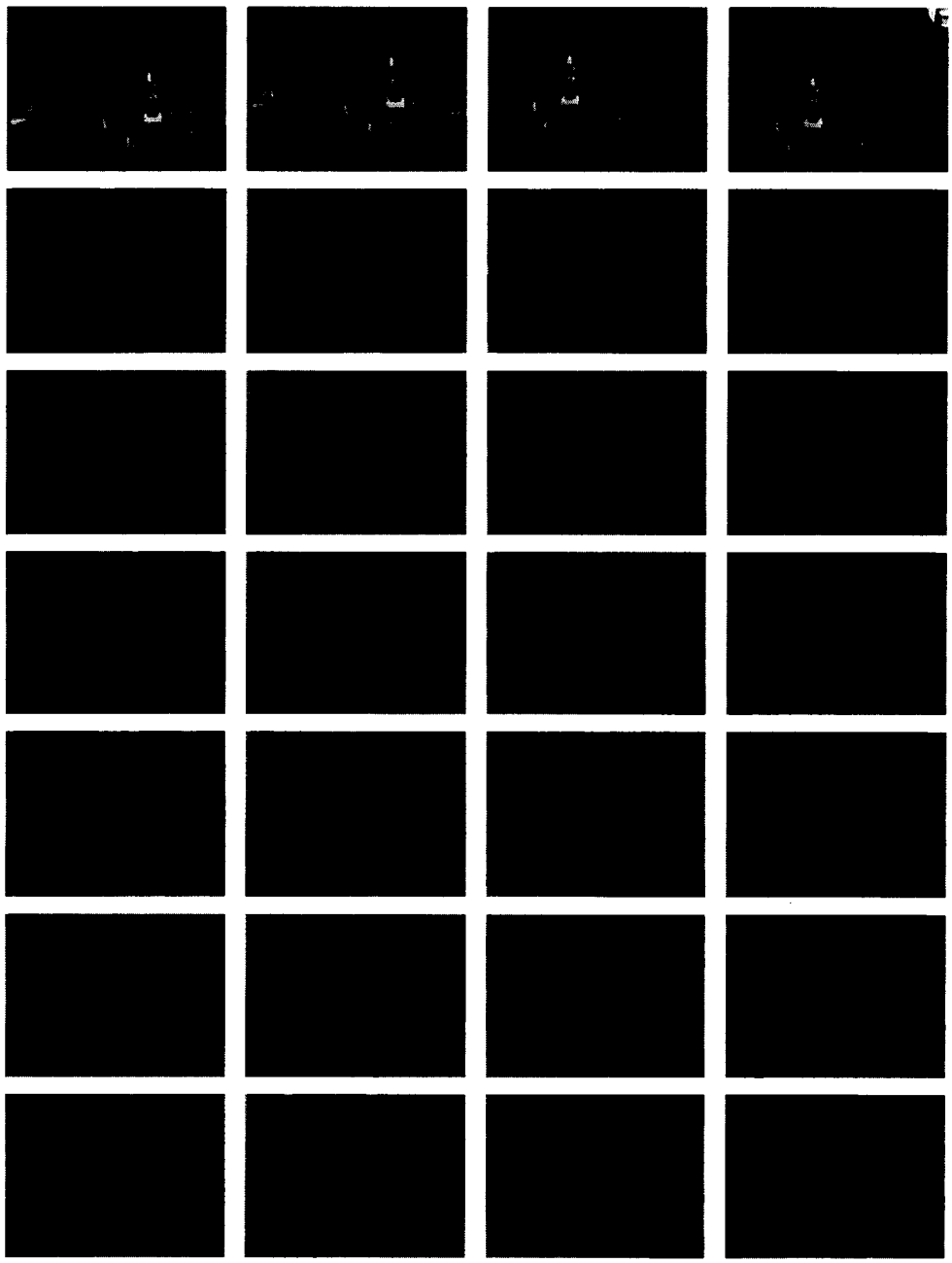


Figure 4.7: Depth maps computed by curve-based and reliability DP. First row: reference images. Second row: 1 random curve depth maps. Third row: 5 random curve depth maps. Fourth row: 15 random curve depth maps. Fifth Row: 25 random curve depth maps. Sixth Row: non-random hi-res CSFC depth maps. Seventh Row: Reliability DP [19] depth maps.

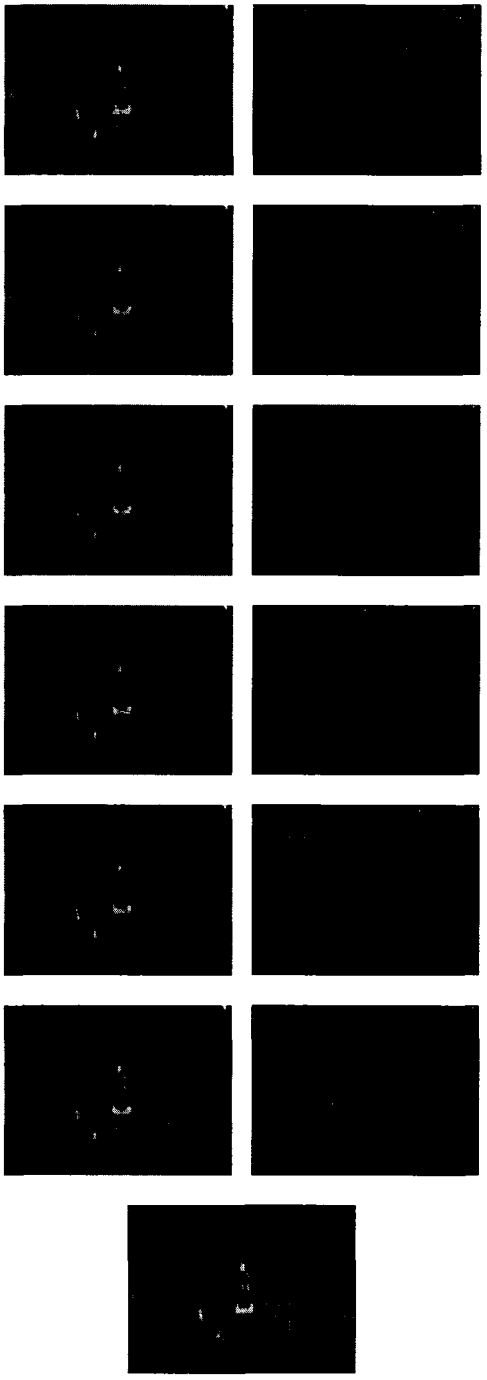


Figure 4.8: A comparison of the rendering results (left) and ground truth differences (right) of curve-based and reliability DP. First row: results for 1 random curve. Second row: results for 5 random curves. Third row: results for 15 random curves. Fourth row: results for 25 random curves. Fifth row: results for non-random hi-res CFSCs. Sixth row: results for reliability DP [19]. Seventh row: ground truth image.

Approach	Mean SSD Score	Median SSD Score	Maximum SSD Score
1 random curve	702.4	96.4	118273.4
5 random curves	668.6	95.2	119571.0
15 random curves	657.0	92.8	119271.8
25 random curves	654.9	96.2	119775.6
non-random curve	680.2	96.8	121251.1
reliability DP [19]	701.0	98.6	118657.4

Table 4.4: A comparison of error rates averaged over 5 consecutive frames for curve-based and reliability-based stereo matching approaches in our IBR system (see Figure 4.8 for corresponding images).

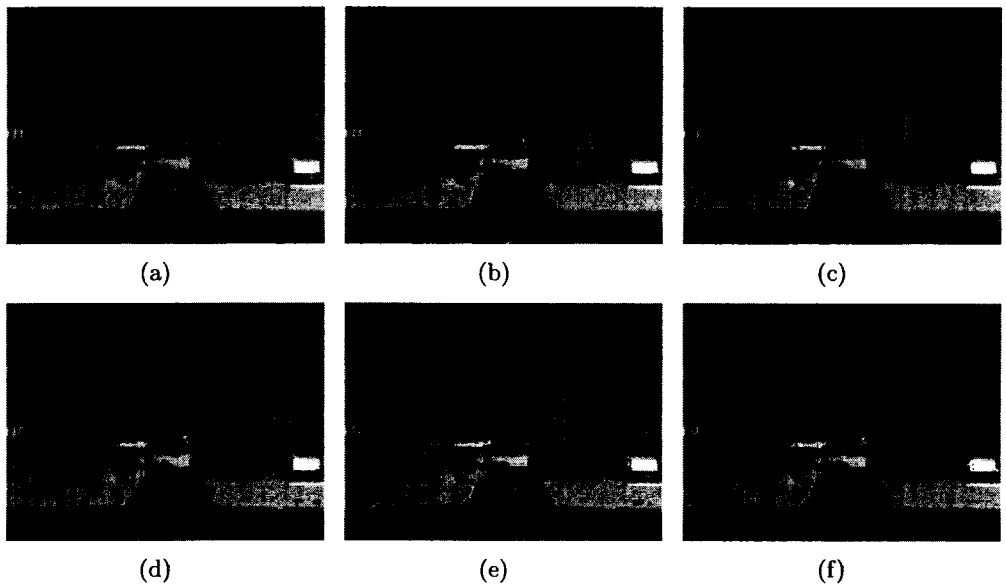
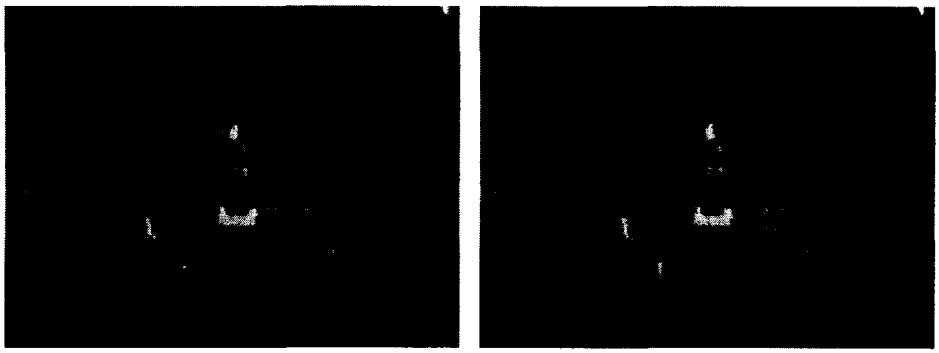


Figure 4.9: A comparison of rendering results with and without median filtering. (a,b,c) median filtering enabled; and (d,e,f) median filtering disabled.

static scene rendered with and without median filtering in Figure 4.9. In the shown example, the SSD for the intensity error when the median-filtered result is compared to the ground truth is 761.86. For the images without median filtering enabled, the SSD is 961.656.

Figure 4.10 demonstrates the use of hole filling in our approach. In the image with holes, the mean SSD when compared to the error-checking camera is 704.42. The number of pixels with an SSD greater than 400 (considered erroneous pixels) is 10,620, not including image holes. The number of hole pixels is 4985. After hole-filling is applied, the same scene has a mean SSD of 678.51 and the number of



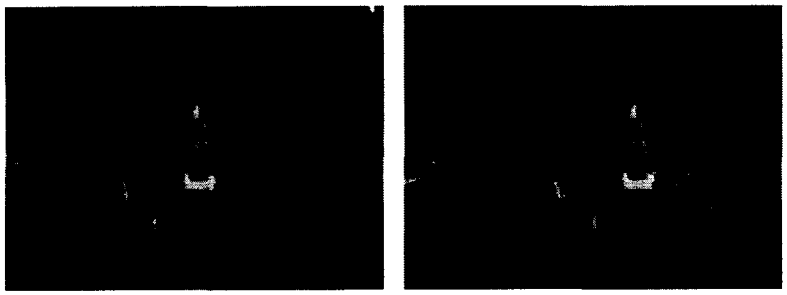
(a) A static scene with image holes. (b) The same scene with holes filled.

Figure 4.10: An example of image hole filling.

erroneous pixels with an SSD greater than 400 is 11,329. This means that roughly 14.2% of the pixels coloured by the hole-filling approach are considered erroneous, compared to the initial results in which 13.8% of pixels are considered erroneous (not including image holes). This demonstrates that our hole-filling approach is able to fill holes with an accuracy comparable to that of the main rendering algorithm. The mean SSD drops slightly when hole filling is turned on because the holes in this scene typically occur in untextured background regions. Properly-filled holes in these regions often have a very low error that contributes to decreasing the average error in the image.

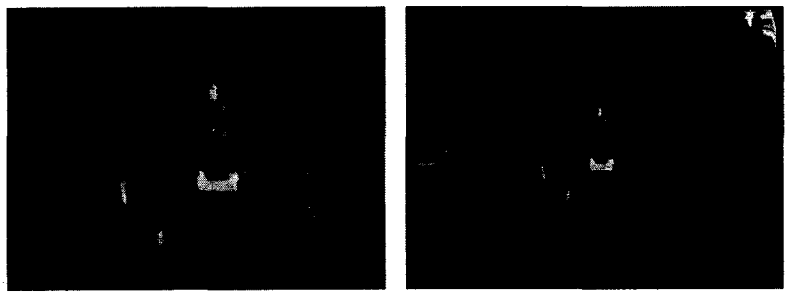
An important benefit of our system's modification of the approach in [48] to use depth is that we now have the innate ability to position the virtual camera at an arbitrary position. Results for a static scene using different camera positions are presented in Figure 4.11. Of course, there are limits to accuracy when the virtual camera moves to an angle containing scene points not visible in any of the reference cameras. Also, as the virtual camera moves further away from the reference cameras, errors in the depth maps are amplified as objects begin to break apart or move to incorrect positions.

Because our system uses discretized depth values instead of disparity, we have the ability to modify and tune the depth resolution for increased performance. This functionality is implemented as a feature of the graphical user interface so users can modify the depth resolution and the front and back clip planes to achieve the best results for the current scene. Visual results using different depth resolutions



(a) The "default" camera position, roughly centered between the four reference cameras.

(b) A horizontally translated position.



(c) A position closer to the scene.

(d) A position further away from the scene.

(e) A rotated position.

Figure 4.11: Examples of different camera positions and orientations.

Number of depth levels	Frame rate (FPS)	Mean SSD error	Median SSD error	Maximum SSD error
128	0.88	658.9	94.6	121886.4
64	1.68	692.8	97.0	118955.6
32	3.15	657.0	94.8	119320.8
16	6.10	710.1	95.2	119268.8
8	10.67	989.1	100.6	120975.8

Table 4.5: Frame rate and error (averaged over five consecutive frames) for the results presented in Figure 4.12.

for a scene of static objects are presented in Figure 4.12. The corresponding frame rate and SSD error statistics are given in Table 4.5. Once again, the minimum SSD error is 0.0 in all tests, so we do not include it. Interestingly, visual quality does not decrease significantly as the depth resolution shrinks, allowing us to achieve much better performance without sacrificing visual quality. Some image detail is lost as the depth resolution decreases, until finally at 8 depth levels the system is unable to resolve objects in the scene properly.

A similar experiment is performed in Figure 4.13 with a person in front of a static background. Once again, image quality remains acceptable down to 16 depth levels. As the depth levels decrease, the frame rate increases. We have found that at increased frame rates reconstruction errors seem to be less visually jarring and more easily forgiven by the human visual system. For this reason, the best visual results for dynamic scenes are often found using very few discrete depth levels (for example, 32 depth levels are often sufficient for impressive visual results). Hence, it is anticipated that the quality of rendering will improve as the hardware performance improves by using more depth levels at a higher speed.

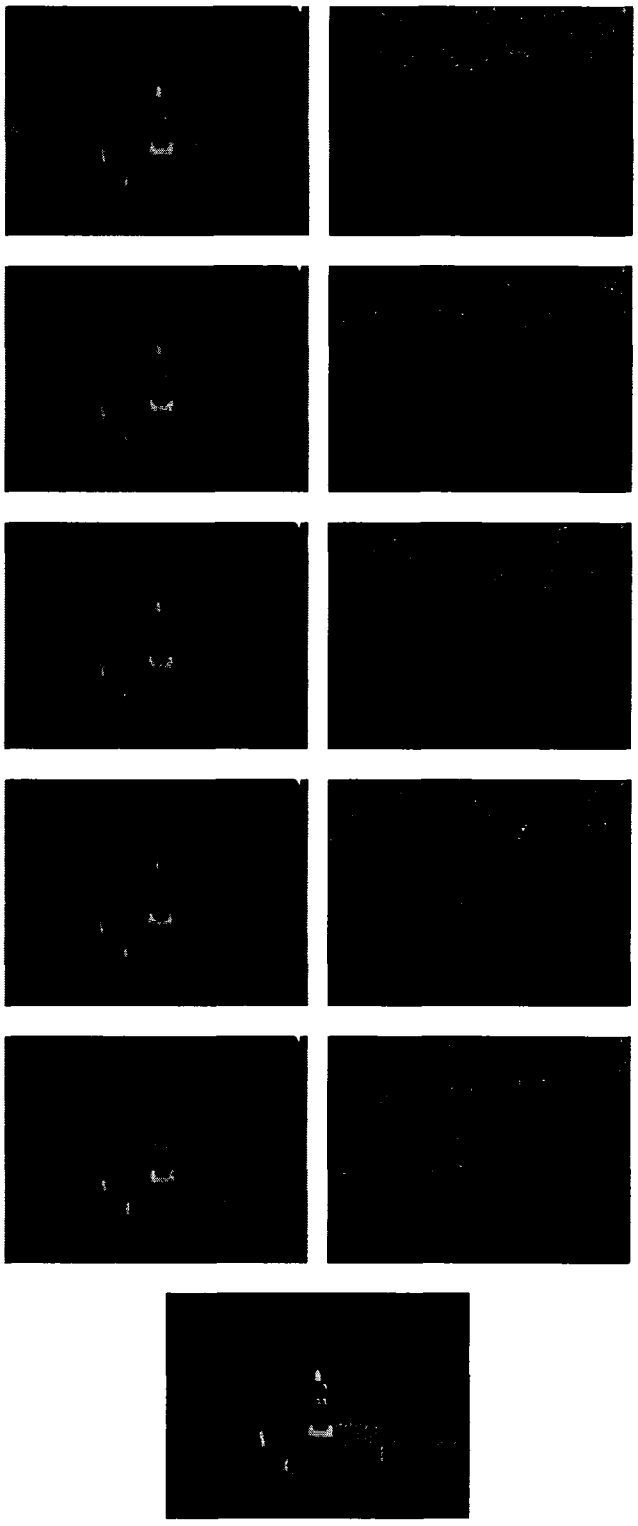
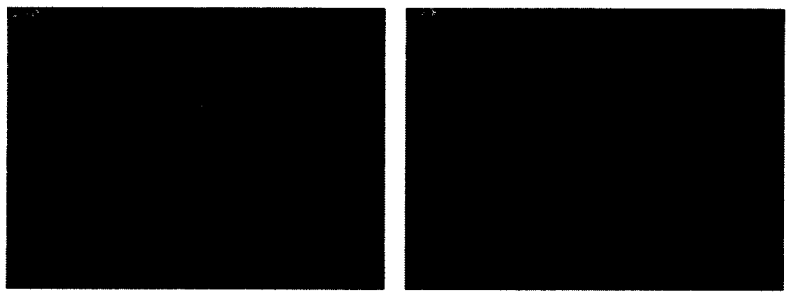


Figure 4.12: Rendering results for a static scene using different depth resolutions. First row: 128 depth levels. Second row: 64 depth levels. Third row: 32 depth levels. Fourth row: 16 depth levels. Fifth row: 8 depth levels. Sixth row: ground truth image.



(a) 128 depth levels

(b) 64 depth levels

(c) 32 depth levels

(d) 16 depth levels

(e) 8 depth levels

Figure 4.13: Rendering results for a non-static scene using different depth resolutions.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this work we have proposed an image-based rendering system capable of capture and rendering at interactive rates on a single consumer-level desktop PC. The system makes heavy use of GPU hardware and can be used with any number of calibrated input cameras. An implementation of the system involving 4 cameras positioned around a 19-inch CRT monitor was demonstrated. This example has shown that the system is capable of dealing with the wide-baseline stereo images acquired in such a situation, and as such would be suitable for applications such as gaze correction for teleconferencing.

In our experimental setup, images from the four calibrated reference cameras are captured and lens distortion is corrected on the GPU. Following this, stereo matching is performed to create depth maps. Cost matching and aggregation are performed on the GPU, and the results are transferred to the CPU for optimization. The depth maps are returned to the GPU for a backwards-rendering view synthesis algorithm. The system uses novel modifications of existing techniques to achieve a high-quality result at interactive frame rates.

5.1.1 Stereo Matching

The cost optimization in our stereo matching system uses a unique implementation of dynamic programming in the same vein as Veksler's pixel trees [45]. In our case, space filling curves are used to provide a global dynamic programming-based solution. The nature of the space-filling curve depends on the application desired.

For offline computations, a custom-generated space filling curve may be created using intensity and/or region information as is done with trees in [45] and [29]. Due to the high cost of curve generation, applications requiring faster performance may use random space-filling curves with results filtered using a post-process such as median filtering as presented in Section 3.1.

Experiments show both random and non-random SFCs to be competitive and in some cases superior to current DP approaches such as reliability-based DP [19] and pixel-tree based DP [45]. When combined with cross-checking, non-random curve-based DP outperforms both approaches in the Middlebury rankings, and shows very good accuracy along object discontinuities in the sample images. On the other hand, random curve-based DP generally performs better in the interior of object regions and non-occluded areas. In fact, random curve-based DP is able to compete with and outperform non-random curves in datasets with a higher disparity range such as the Cones and Teddy datasets. When the error threshold is decreased to 0.5, the random curve approach outperforms both reliability-based and pixel-tree based approaches, and in some instances it outperforms the recently published region-tree based dynamic programming approach [29]. This result lends support to the use of consensus-based dynamic programming approaches for applications where sub-pixel accuracy may be required.

In our GPU-based IBR implementation, DP is applied over a single random curve per frame. Image differencing is performed to selectively apply a median filter to objects that have remained relatively static over the past 5 frames. In addition, depth maps created for our IBR experiments are assumed to have static backgrounds. After several confirming observations a background pixel may be locked in as a ground control point [19], at which point future DP passes will not examine other depth hypotheses for the pixel. This provides a significant speed increase. To prevent possible erroneous depths to refresh after a brief period of time, ground control points will "decay" and reset after a user-defined number of frames.

5.1.2 Image-based Rendering

The image-based rendering algorithm used in our system is the GPU-based approach used in [48], modified to work with depth values. The depth-based modification allows us to remove previous restrictions on the novel camera position to allow for rotation and 3-dimensional translation of the novel view camera. If desired, a dynamically updated background model computed using incoming depth maps may be used to aid rendering.

The image-based rendering system is able to work in a variety of settings and environments. We have found that one can reduce the resolution of the depth discretization to achieve a significant speed increase without having an excessively adverse effect on the quality of the rendered result. In our implementation, we are able to achieve frame rates in excess of 5 frames per second while still maintaining acceptable visual quality.

5.2 Limitations and Future Work

The fact that random curves are able to outperform non-random curves in certain areas while non-random curves maintain better performance along object boundaries lends some credence to the idea of combining the two for a more effective dynamic programming-based stereo matching approach. In the future it may be worthwhile to experiment with using several curves that are random in image areas that would benefit from the use of a random curve, but non-random along object boundaries. There are certain instances in which texture is confusing and intensity information is not the best criteria to use for non-random curve construction. For example, two overlapping objects at different depths may have a very similar texture/colour. In the future it may be worth experimenting with alternatives such as using matching costs or an initial rough disparity estimate to guide curve construction for the best possible results.

With faster hardware and more intelligent data structures, the space-filling curve-based DP presented in this thesis would be be much more accurate for real-time applications. Generating a custom CSFC from scratch may be too slow for current real-time applications, but in the future we wish to experiment with ways

of quickly customizing space-filling curves by refining a predictable structure according to the intensity data in the input images and/or previous depth estimates. This would produce better results by reducing erroneous depth labels caused by the curve crossing object boundaries, and possibly remove the need for the random curve/temporal filter combination in our GPU IBR implementation.

In addition, it would be worthwhile to investigate methods of handling occlusion in wide-baseline binocular stereo so that we could reduce the number of cameras used in our experimental setup. Two cameras in conjunction with an accurate heuristic for filling occluded holes would produce a system capable of much higher frame rates with reduced equipment cost and setup time.

Our IBR system will present lower quality results when dealing with quick movement of objects in the scene. This is due to a number of factors, chief among them being the lack of synchronization among the input cameras. Wang and Yang note that many typical body motions can cause a position offset of as much as 10 pixels in reference cameras capturing at 30 Hz [46]. In this situation, stereo matching and subsequent IBR will fail. Solutions to this include using cameras that support hardware synchronization or correcting the images using a time-consuming software synchronization step [46]. We feel that adding hardware synchronization to our camera setup would improve results for fast-moving objects.

Finally, we have found that the hole filling algorithm used in this work is highly susceptible to the accuracy of the depth maps used. An inaccurate depth map or improper camera selection can result in the "rubber sheet effect" [48] being applied as holes are filled in the image. In the future, it would be worthwhile to investigate alternative methods of image completion to fill holes in the rendered result.

Bibliography

[1] Edward H. Adelson and James R. Bergen. The plenoptic function and the elements of early vision. *Computational Models of Visual Processing*, pages 3–20, 1991.

[2] Jean-Yves Bouguet. Camera calibration toolbox for MATLAB. http://www.vision.caltech.edu/bouguetj/calib_doc/, Viewed on March 10, 2006.

[3] Yuri Boykov, Olga Veksler, and Ramin Zabih. Fast approximate energy minimization via graph cuts. In *Proceedings of International Conference on Computer Vision*, pages 377–384, 1999.

[4] Greg Breinholt and Christoph Schierz. Algorithm 781: Generating hilbert's space-filling curve by recursion. *ACM Transactions on Mathematical Software*, 24(2):184–189, June 1998.

[5] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *Proceedings of SIGGRAPH 2004*, pages 777–786, 2004.

[6] Eric Chen. Quicktime VR: an image-based approach to virtual environment navigation. In *Proceedings of SIGGRAPH 1995*, pages 29–38, 1995.

[7] Antonio Criminisi, Jamie Shotton, Andrew Blake, and Philip Torr. Gaze manipulation for one-to-one teleconferencing. In *Proceedings of the International Conference on Computer Vision*, pages 191–198, 2003.

[8] Revital Dafner, Daniel Cohen-Or, and Yossi Matias. Context-based space filling curves. *Computer Graphics Forum*, 19(3):209–218, 2000.

[9] Yi Deng and Xueyin Lin. A fast line segment based dense stereo algorithm using tree dynamic programming. In *Proceedings of the European Conference on Computer Vision*, pages 201–212, May 2006.

[10] Marc-Antoine Drouin, Martin Trudeau, and Sebastien Roy. Improving border localization of multi-baseline stereo using border-cut. In *Proceedings of IEEE Computer Vision and Pattern Recognition*, pages 511–518, July 2006.

[11] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Efficient belief propagation for early vision. *International Journal of Computer Vision*, 70(1):41–54, October 2006.

[12] Bastian Goldlücke and Marcus Magnor. Joint 3-d reconstruction and background separation in multiple views using graph cuts. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR'03)*, Madison, USA, pages 683–694, June 2003.

[13] Bastian Goldlücke, Marcus Magnor, and Bennett Wilburn. Hardware-accelerated dynamic light field rendering. In *Proceedings of Vision, Modeling and Visualization 2002*, pages 455–462, November 2002.

[14] Minglun Gong. *Rayset and Its Applications to Static and Dynamic Image Synthesis.* PhD thesis, University of Alberta, 2003.

[15] Minglun Gong and Ruigang Yang. Image-gradient-guided real-time stereo on graphics hardware. In *Proceedings of International Conference on 3-D Imaging and Modeling*, pages 548–555, 2005.

[16] Minglun Gong and Yee-Hong Yang. Fast stereo matching using reliability-based dynamic programming and consistency constraints. In *Proceedings of IEEE International Conference on Computer Vision (ICCV '03)*, pages 610–617, 2003.

[17] Minglun Gong and Yee-Hong Yang. Camera field rendering of static and dynamic scenes. *Graphical Models*, 67(2):73–99, March 2005.

[18] Minglun Gong and Yee-Hong Yang. Fast unambiguous stereo matching using reliability-based dynamic programming. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(6):998–1003, June 2005.

[19] Minglun Gong and Yee-Hong Yang. Near real-time reliable stereo matching using programmable graphics hardware. In *Proceedings of IEEE CVPR 2005*, pages 924–931, 2005.

[20] Minglun Gong and Yee-Hong Yang. Real-time stereo matching using orthogonal reliability-based dynamic programming algorithm. Submitted, 2006.

[21] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing.* Prentice-Hall, Inc., 2nd edition, 2002.

[22] Mark Harris and Cliff Woolley (forum moderators). http://www.gpgpu.org. Viewed on June 13, 2006.

[23] Heiko Hirschmüller. Stereo vision in structured environments by consistent semi-global matching. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 2386–2393, June 2006.

[24] Sing Bing Kang and Larry Zitnick. Projecion test and results for Microsoft Research 3d video. http://research.microsoft.com/vision/InteractiveVisualMediaGroup /3DVideoDownload/TestProjection.doc, Viewed on March 10, 2006.

[25] Jae Chul Kim, Kyoung Mu Lee, Byoung Tae Choi, and Sang Uk Lee. A dense stereo matching using two-pass dynamic programming with generalized ground control points. In *Proceedings of CVPR 2005*, pages 1075–1082, 2005.

[26] Vladimir Kolmogorov and Ramin Zabih. Computing visual correspondence with occlusions using graph cuts. In *Proceedings of the International Conference on Computer Vision*, pages 508–515, 2001.

[27] Scott Larsen. Using the graphics processing unit for computer vision. In *IEEE CVPR 2006 (tutorial sessions)*, June 2006.

[28] Jonathan K. Lawder and Peter J.H. King. Using space-filling curves for multi-dimensional indexing. In *Proceedings of BNCOD 17, Lecture Notes in Computer Science*, pages 20–35, 2000.

[29] Cheng Lei, Jason Selzer, and Yee-Hong Yang. Region-tree based stereo using dynamic programming optimization. In *Proceedings of IEEE Computer Vision and Pattern Recognition*, pages 2378–2385, June 2006.

[30] Ming Li, Marcus Magnor, and Hans-Peter Seidel. Hardware-accelerated visual hull reconstruction and rendering. In *Proceedings of Graphics Interface*, pages 65–71, June 2003.

[31] Wojciech Matusik, Chris Buehler, Ramesh Raskar, Steven J. Gortler, and Leonard McMillan. Image-based visual hulls. In *Proceedings of ACM SIGGRAPH 2000*, pages 369–374, 2000.

[32] Leonard McMillan. *An Image-Based Approach to Three-Dimensional Computer Graphics.* PhD thesis, University of North Carolina at Chapel Hill, 1997.

[33] Leonard McMillan and Gary Bishop. Plenoptic modeling: An image-based rendering system. In *Proceedings of SIGGRAPH '95*, pages 39–46, August 1995.

[34] Baback Moghaddam, Kenneth J. Hintz, and Clayton V. Stewart. Space-filling curves for image compression. In *Proceedings of the SPIE*, pages 414–421, August 1991.

[35] Kiyohide Satoh Yuichi Ohta. Occlusion detectable stereo - systematic comparison of detection algorithms. In *Proceedings of International Conference on Pattern Recognition (ICPR '96)*, pages 280–286, 1996.

[36] Manuel M. Oliveira, Gary Bishop, and David McAllister. Relief texture mapping. In *Proceedings of SIGGRAPH 2000*, pages 359–368, 2000.

[37] Peter Rander, PJ Narayanan, and Takeo Kanade. Virtualized reality: Constructing time-varying virtual worlds from real world events. In *Proceedings of IEEE Visualization '97*, pages 277–283, October 1997.

[38] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision*, 47(1-3):7–42, April-June 2002.

[39] Harmut Schirmacher, Li Ming, and Hans-Peter Seidel. On-the-fly processing of generalized lumigraphcs. In *Proceedings of Eurographics 2001*, pages 165–173, 2001.

[40] Steven M. Seitz and Charles R. Dyer. View morphing. In *Proceedings of SIGGRAPH '96*, pages 21–30, 1996.

[41] Jonathan Shade, Steven Gortler, Li wei He, and Richard Szeliski. Layered depth images. In *Proceeding of ACM SIGGRAPH 1998*, pages 231–242, 1998.

[42] Heung-Yeung Shum and Li-Wei He. Rendering with concentric mosaics. In *Proceedings of SIGGRAPH '99*, pages 299–306, 1999.

[43] Heung-Yeung Shum and Sing Bing Kang. A review of image-based rendering techniques. In *Proceedings of IEEE/SPIE Visual Communications and Image Processing (VCIP)*, pages 2–13. Institute of Electrical and Electronics Engineers, Inc., June 2000.

[44] Jian Sun, Heung-Yeung Shum, and Nan-Ning Zheng. Stereo matching using belief propagation. *PAMI*, 25(7):787–800, July 2003.

[45] Olga Veksler. Stereo correspondence by dynamic programming on a tree. In *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR '05)*, pages 384–390, 2005.

[46] Huamin Wang and Ruigang Yang. Towards space-time light field rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, pages 125–132, 2005.

[47] Jan Woetzel and Reinhard Koch. Real-time multi-stereo depth estimation on gpu with approximative discontinuity handling. In *Proceedings of 1st European Conference on Visual Media Production (CVMP 2004)*, pages 245–254, March 2004.

[48] Yi Xu. Hardware-accelerated image-based rendering with depth information. Master's thesis, University of Alberta, 2004.

[49] Ruigang Yang, Marc Pollefeys, and Sifang Li. Improved real-time stereo on commodity graphics hardware. In *IEEE Workshop on Real Time 3D Sensors and Their Use (In conjunction with CVPR 2004)*, page 36, 2004.

[50] Ruigang Yang, Greg Welch, and Gary Bishop. Real-time consensus-based scene reconstruction using commodity graphics hardware. In *Proceedings of Pacific Graphics*, pages 225–235, October 2002.

[51] Lawrence Zitnick, Sing Bing Kang, Matthew Uyttendaele, Simon Winder, and Richard Szeliski. High-quality video view interpolation using a layered representation. *ACM Transactions on Graphics*, 23(3):598–606, August 2004.

Appendix A

Camera Calibration

The cameras were calibrated using Bouguet's Camera Calibration Toolbox for MATLAB [2]. To calibrate the cameras, images of a 15x15 checkerboard pattern were captured simultaneously in all cameras using a separately coded capture application. Care was taken to ensure the calibration pattern was completely visible in every image for each camera.

Following individual calibration of each camera's intrinsic parameters, each camera is extrinsically calibrated against an arbitrarily selected reference camera, which is used as the origin in the world coordinate system. In our experiments the reference camera was the error-checking camera located at the center of the rig. In the absence of this camera, one of the system's four reference cameras can be used without problems. The extrinsic calibration is performed using the stereo pair calibration component of Bouguet's toolbox, and is also used to further refine the intrinsic parameters of the other cameras (the world origin reference camera has its intrinsic parameters fixed for the sake of consistency across the stereo calibration of different pairs).

The parameters obtained from calibration and used in experiments are as follows:

Camera 0 (Error-checking camera, not used in computation)

`focal length` = (256.711019, 255.230759)
`principal point` = (170.942301, 90.221512)
`distortion coefficients` = (−0.368646, 0.167099, 0.002168, −0.001299)
`location` = (0.000000, 0.000000, 0.000000)

$$\texttt{projection matrix} = \begin{pmatrix} 256.711019 & 0.000000 & 170.942301 & 0.000000 \\ 0.000000 & 255.230759 & 90.221512 & 0.000000 \\ 0.000000 & 0.000000 & 1.000000 & 0.000000 \\ 0.000000 & 0.000000 & 0.000000 & 1.000000 \end{pmatrix}$$

Camera 1

`focal length` = (256.958929, 255.135532)
`principal point` = (168.568358, 106.197102)
`distortion coefficients` = (−0.346351, 0.125654, 0.001309, 0.000741)
`location` = (243.617098, 21.925511, 1.249159)

$$\texttt{projection matrix} = \begin{pmatrix} 258.123595 & 10.118744 & 166.472229 & 62810.157312 \\ -5.658007 & 257.305308 & 100.667419 & 5726.633867 \\ 0.007438 & 0.021758 & 0.999736 & 1.249159 \\ 0.000000 & 0.000000 & 0.000000 & 1.000000 \end{pmatrix}$$

Camera 2

$\texttt{focal length} = (258.950469, 257.088971)$
$\texttt{principal point} = (167.597843, 100.770291)$
$\texttt{distortion coefficients} = (-0.355169, 0.141680, -0.000138, 0.000958)$
$\texttt{location} = (244.396565, -73.036413, -7.203100)$

$$\texttt{projection matrix} = \begin{pmatrix} 258.626847 & 6.277193 & 167.979562 & 62079.381196 \\ -3.735549 & 258.647630 & 96.627324 & -19502.714676 \\ -0.001781 & 0.016041 & 0.999870 & -7.203100 \\ 0.000000 & 0.000000 & 0.000000 & 1.000000 \end{pmatrix}$$

Camera 3

$\texttt{focal length} = (257.100745, 255.559471)$
$\texttt{principal point} = (164.629606, 102.397620)$
$\texttt{distortion coefficients} = (-0.355167, 0.133996, 0.001009, -0.003996)$
$\texttt{location} = (-238.885035, -73.856757, 8.429006)$

$$\texttt{projection matrix} = \begin{pmatrix} 249.346030 & 4.872812 & 176.086664 & -60029.856741 \\ -2.932511 & 259.212905 & 92.714539 & -18011.683555 \\ -0.045454 & 0.037463 & 0.998264 & 8.429006 \\ 0.000000 & 0.000000 & 0.000000 & 1.000000 \end{pmatrix}$$

Camera 4

$\texttt{focal length} = (256.777368, 255.584916)$
$\texttt{principal point} = (168.724713, 103.550546)$
$\texttt{distortion coefficients} = (-0.350430, 0.121346, 0.002963, -0.004418)$
$\texttt{location} = (-240.992038, 23.634948, 7.109128)$

$$\texttt{projection matrix} = \begin{pmatrix} 251.730123 & -0.177050 & 176.166282 & -60681.815529 \\ -2.168034 & 255.986186 & 102.531641 & 6776.890314 \\ -0.029254 & 0.003907 & 0.999564 & 7.109128 \\ 0.000000 & 0.000000 & 0.000000 & 1.000000 \end{pmatrix}$$

Appendix B

Image Capture

Due to differences in color gains and offsets in cameras (even when identical models are used), the intensities recorded in images taken of the same scene with different cameras are often different. This can harm the quality of stereo matching and image-based rendering results, so in situations where multiple cameras are used simultaneously for dynamic scenes some colour correction is often applied.

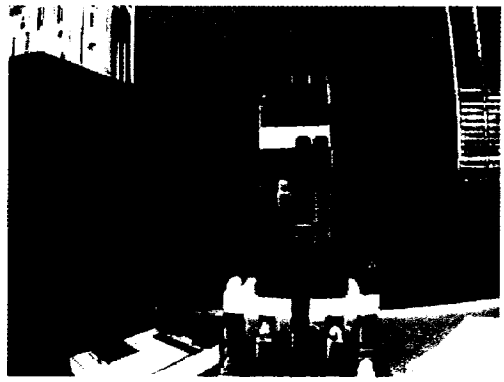
To correct the color differences in our cameras, we use a simple histogram-matching approach. Prior to setting up the cameras in their final positions, we sequentially place each camera in an identical position and capture an identical image of a static scene. One camera is arbitrarily selected as a colour reference, and the RGB histograms of the other cameras are matched to that of the reference camera using histogram matching [21]. The histogram matching process generates a lookup table for each colour channel of each camera which can be used to transform the recorded colours to corrected colours. We perform this operation on the CPU immediately after the images are captured in our IBR system so that the colours are corrected for stereo matching and IBR processing. An example of uncorrected and corrected images of a static scene is shown in Figure B.1.

After colour correction, images are transferred to the GPU for distortion correction and noise removal. Distortion correction is performed using a fragment shader program that corrects according to the distortion parameters computed by Bouget's camera calibration toolkit [2]. An example of an image before and after distortion correction is presented in Figure B.2.

Finally, noise is removed by running a 3x3 median filter over each image using a fragment shader program. The shader computes a median value for each colour channel and outputs the final result. An example of the effects of the median filter are shown in Figure B.3.

(a) Uncorrected camera image

(b) Corrected camera image



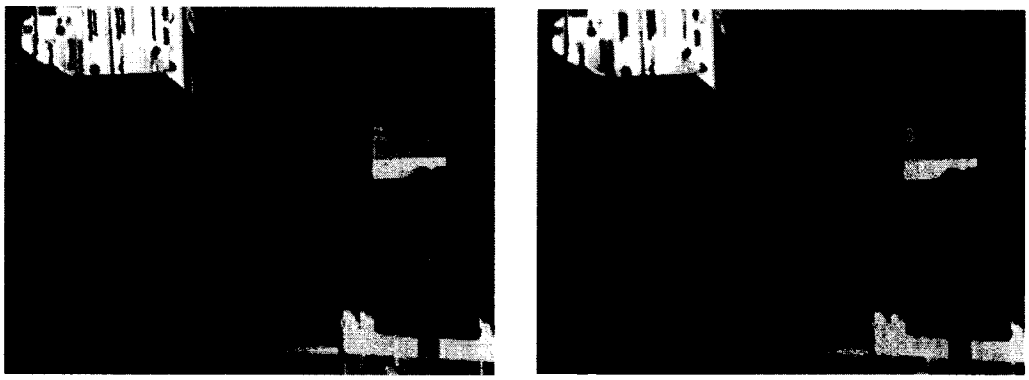
(c) Colour reference camera image

Figure B.1: An example of the colour correction technique applied to raw input images.

(a) Distorted camera image

(b) Undistorted camera image

Figure B.2: An example of distortion correction.



(a) Original camera image (b) Camera image after median filtering

Figure B.3: An example of median filtering for noise removal.

Appendix C

Experimental Parameters

Jump Costs for 1.0 error threshold tests

```
jump cost (35 random curves) = 130
jump cost (35 random curves + cross-check) = 110
jump cost (low-resolution CSFC) = 50
jump cost (high-resolution CSFC) = 60
jump cost (high-resolution CSFC + cross-check) = 150
```

Jump Costs for 0.5 error threshold tests

```
jump cost (35 random curves) = 80
jump cost (35 random curves + cross-check) = 75
jump cost (low-resolution CSFC) = 50
jump cost (high-resolution CSFC) = 55
jump cost (high-resolution CSFC + cross-check) = 150
```

IBR System Parameters

```
SFC jump cost = 80
Euclidean distance threshold for median filter cancelation = 0.05
```